Freescale MQX™ RTOS 4.1.0
i.MX 6SoloX I/O Drivers
User's Guide

MQX410SXIOUG
Rev. 1
05/2015

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQX410SXIOUG
Rev. 1, 05/2015

freescale™

Chapter 1
Before You Begin

Chapter 2
MQX I/O

Chapter 3
Pipe Device Driver

Chapter 4
Serial-Device Families

Chapter 5
SPI Drivers

Chapter 6
SPI Slave Drivers

Chapter 7
I2C Driver

Chapter 8
FlexCAN Driver

Chapter 9
LWGPIO Driver

Chapter 10
Low Power Manager

Chapter 11
LWADC Driver

Chapter 12
HWTIMER Driver

Chapter 13
MMA8451Q Digital Accelerometer Driver

Chapter 14
MAG3110 Digital Magnetometer Driver

Chapter 15
Core_mutex Driver

Chapter 16
Revision History

Chapter 1 Before You Begin

1.1 About This Book

MQX 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

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

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

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

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

1.3 Document Conventions

1.3.1 Notes

Notes point out important information. For example:

NOTE

Non-strict semaphores do not have priority inheritance.

1.3.2 Cautions

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

CAUTION

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

Chapter 2 MQX I/O

2.1 Overview

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

2.2 MQX I/O Layers

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

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

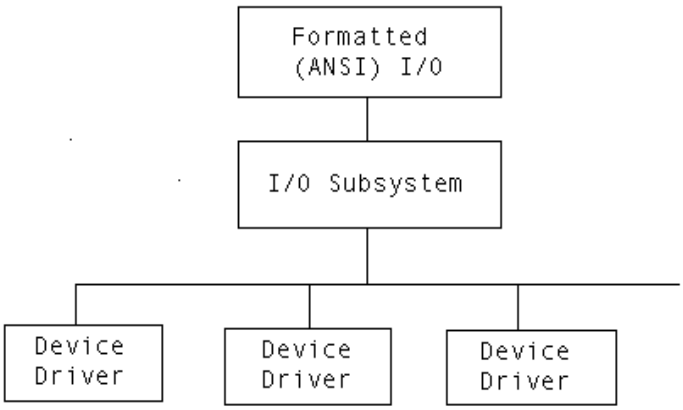

Figure 2-1. MQX I/O Layers

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

2.2.1 I/O Device Structure

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

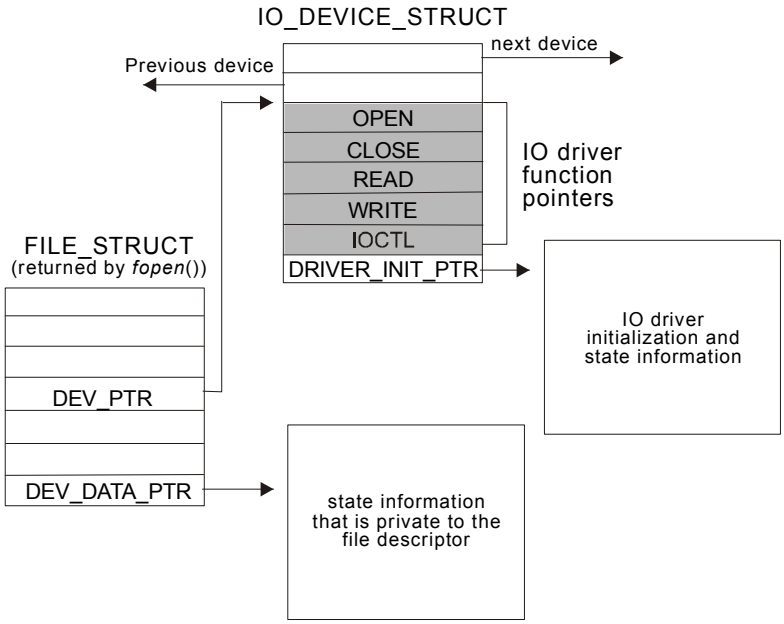

Figure 2-2. I/O Device Structure — I/O Device Drivers

2.2.2 I/O Device Structure for Serial-Device Drivers

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

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

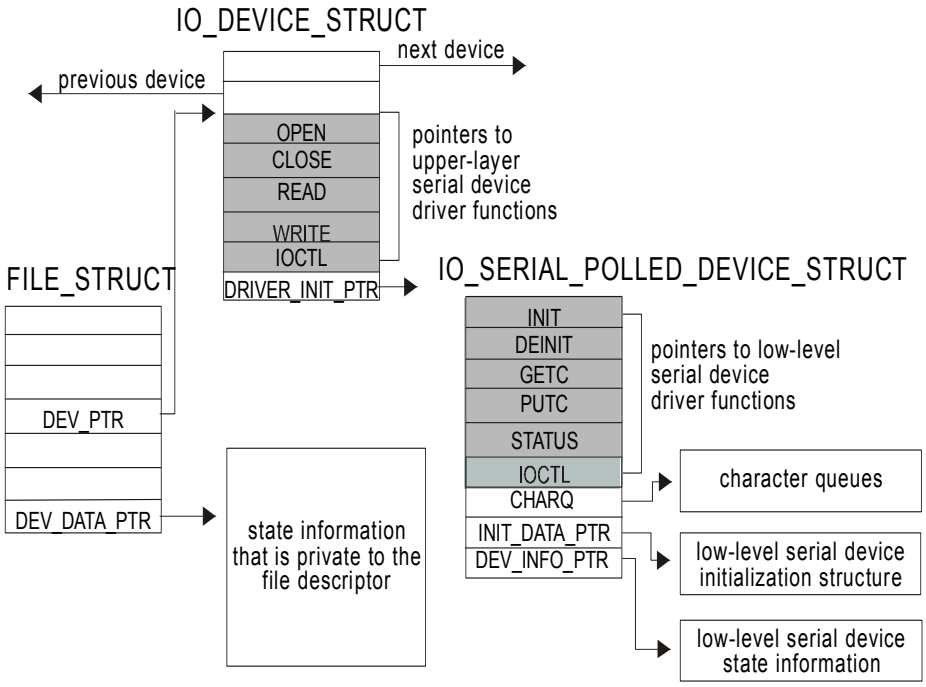

Figure 2-3. I/O Device Structure — Serial-Device Drivers

2.3 Formatted I/O Library

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

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

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

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

2.4 I/O Subsystem

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

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

- _io_dev_install
- _io_dev_install_ext
- _io_dev_uninstall
- _io_get_handle

- _io_init
- _io_set_handle

2.4.1 _io_dev_install

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

Synopsis

```
_mqx_uint _io_dev_install(
    char        *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 Section 2.9, "Device Driver Services" below.

2.7 Device Names

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

For example:

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

installs device `mfs1:`

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

For example:

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

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

2.8 Installing Device Drivers

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

- Call **_io_device_install()** (where **device** is replaced by the name of the driver family) from your application. Usually, the function calls **_io_dev_install()** internally to register the device with MQX 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 Section 2.7, "Device Names" above for restrictions on the string that identifies the name of a device.

2.9 Device Driver Services

A device driver usually provides the following services:

- _io_device_open
- _io_device_close
- _io_device_read
- _io_device_write
- _io_device_ioctl

2.9.1 _io_device_open

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

Synopsis

```
mqx_int _io_device_open(
        FILE_DEVICE_STRUCT_PTR fd_ptr,
        char                   *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 Section 2.10, “I/O Control Commands”).
- *param_ptr [IN/OUT]* — Pointer to the I/O control parameters.

Return Value

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

Remarks

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

2.10 I/O Control Commands

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

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

2.11 Device identification

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

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`

Drivers	Family (device)	Directory in mqx\source\io
PCB (Packet Control Block) drivers (HDLC, I2C, ..)	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
I2C (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 I2C and QSPI drivers. Only character-based master-mode-only I2C and QSPI drivers are supported.

Chapter 3 Pipe Device Driver

3.1 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.

3.2 Source Code Location

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

3.3 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.

3.4 Driver Services

The pipe device driver provides the following services:

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

3.5 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
    )
```

3.6 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.

3.7 I/O Control Commands

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

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

Chapter 4 Serial-Device Families

4.1 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

4.2 Source Code Location

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

4.3 Header Files

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

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

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

4.4 Installing Drivers

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

Driver	Installation Function
Interrupt-driven	_imx_uart_int_install
Polled	_imx_uart_polled_install

4.4.1 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 i.MX family

```
#include <serl_imx_uart.h>
typedef struct imx_uart_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;
} IMX_UART_INIT_STRUCT, *
IMX_UART_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.

Example

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

```
const IMX_UART_INIT_STRUCT _bsp_sci1_init = {
   /* queue size         */ BSPCFG_SCI1_QUEUE_SIZE,
   /* Channel            */ 1,
   /* Clock Speed        */ BSP_PLL3_UART_CLOCK,
   /* Baud rate          */ BSPCFG_SCI1_BAUD_RATE,
   /* RX/TX Int vect     */ NVIC_UART1,
   /* ERR Int vect       */ 0, //INT_UART1_ERR,
   /* RX/TX priority     */ 3,
   /* ERR priority       */ 4,
};
```

4.5 Driver Services

The serial device driver provides these services:

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

4.6 I/O Open Flags

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

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

4.7 I/O Control Commands

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

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

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

4.8 I/O Hardware Signals

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

Signal	Description
IO_SERIAL_CTS	Hardware CTS signal

IO_SERIAL_RTS	Hardware RTS signal
IO_SERIAL_DTR	Hardware DTR signal
IO_SERIAL_DSR	Hardware DSRsignal
IO_SERIAL_DCD	Hardware DCD signal
IO_SERIAL_RI	Hardware RI signal
IO_SERIAL_BRK	Hardware BREAK signal

4.9 I/O Stop Bits

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

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

4.10 I/O Parity

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

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

4.11 Error Codes

No additional error codes are generated.

Chapter 5 SPI Drivers

5.1 Overview

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

5.2 Location of Source Code

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

5.3 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.

5.4 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.

5.5 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.

5.5.1 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, pxs20 and pxs30)

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

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

5.6 Using the Driver

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

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

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

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

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

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.

5.7 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);
```

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");
}
```

5.8 Chip Selects Implemented in Software

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

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

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

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

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

5.9 I/O Open Flags

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

Flag	Description
SPI_FLAG_HALF_DUPLEX or NULL	Read operation on file handle will behave the standard POSIX I/O way.
SPI_FLAG_FULL_DUPLEX	Enables extension to standard POSIX I/O for full-duplex operation.

5.10 I/O Control Commands

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

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

Command	Description	Parameter
IO_IOCTL_SPI_GET_BAUD	Gets the BAUD rate.	uint32_t*
IO_IOCTL_SPI_SET_BAUD	Sets the baud rate. A supported baud rate closest to the given one will be 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_FLAGS	Gets open flags.	uint32_t*

Command	Description	Parameter
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_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.

5.11 Clock Modes

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

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

5.12 Transfer Modes

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

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

5.13 Endian Mode

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

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

5.14 Error Codes

Following the SPI, specific error codes are defined:

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

Chapter 6 SPI Slave Drivers

6.1 ECSPI Slave Driver

This chapter describes the SPI Slave driver which provides interface for ECSPI module.

6.2 Location of Source Code

The source code for ECSPI slave driver is located in source\io\spi_slave\.

6.3 Header Files

To use the ECSPI slave device driver, include the header file *spi_slave_ecspi.h* from source\io\spi_slave\ in your application or in the BSP file bsp.h.

The file *spi_slave_ecspi_prv.h* contains private definitions and data structures that SPI device driver uses. This file is required to compile an SPI slave device driver. There is no need to include this file directly in your application.

6.4 Internal Design of SPI Slave Driver

The SPI slave driver is designed as single layer, low level and non POSIX driver. This driver is device specific and implements functions for basic SPI slave demonstration. It contains these main parts:

- ecspi_slave_info_struct – structure keeping initialization data
- ecspi_slave_init() – initialization function
- ecspi_slave_shutdown() – disables driver and returns resources to the default state
- CALLBACK function – manages incoming and outgoing data and sync mechanism
- ecspi_slave_irq_handler() – passes data to/from callback function.

6.5 Installing SPI Driver

The SPI slave driver is not a POSIX driver and it does not need to be installed.

6.6 Initialization structure

The initialization function ecspi_slave_init() requires a pointer to an initialization structure to be passed to it. This structure is used to initialize the ECSPI device and the driver itself.

Initialization structure contains two sections.

- User section – data must be initialized by user

- System section – data are set by driver

Initialization structure

```
typedef struct ecspi_slave_info_struct
{
    /* -------- User section -------- */
    uint32_t INSTANCE;
    uint32_t FRAME_SIZE;
    uint32_t CS;
    uint32_t SS_POL;
    uint32_t SS_CTL;
    uint32_t MODE;
    _mqx_int (*CALLBACK)(void *app_data_ptr, uint32_t *tx_data_ptr, uint32_t
*rx_data_ptr);
    void *APP_DATA_PTR;
    /* -------- System section -------- */
    ECSPI_MemMapPtr ECSPI_REG_PTR;
} ECSPI_SLAVE_INFO_STRUCT, * ECSPI_SLAVE_INFO_STRUCT_PTR;
```

Parameters

INSTANCE – Instance number

FRAME_SIZE – Size of single SPI data frame in bits.

CS – Chip select signal number for transfer.

SS_POL – Slave select active state: 0-low, 1-high.

SS_CTL – Burst end by: 0-number of bits, 1-SS edge.

MODE – Clock polarity and phase setting.

CALLBACK – ECSPI slave driver callback.

APP_DATA_PTR – Pointer to optional application data for callback.

ECSPI_REG_PTR – Register access pointer (set by driver itself).

Example of initialization structure

```
ECSPI_SLAVE_INFO_STRUCT info =
{
/* User's section */
4,                          // INSTANCE.
FRAME_SIZE,                 // FRAME_SIZE.
3,                          // CS.
0,                          // SS_POL,
0,                          // SS_CTL
SPI_CLK_POL_PHA_MODE1,      // MODE,
data_transfer,              // CALLBACK
NULL,                       // APP_DATA_PTR
/* System section. */
NULL                        // ECSPI_REG_PTR
}:
```

6.7 Using the Driver

SPI slave driver provides initialization function ecspi_slave_init(). This function sets up ECSPI block according to user settings in the initialization structure, installs ISR, configures appropriate pins to ECSPI slave functionality and enables data handling.

SPI slave driver executes ISR ecspi_slave_irq_handler() each time an interrupt occurs. ISR in turn executes the callback function. Callback is defined by user and handles RX/TX buffers and synchronization mechanism between application and driver.

Example of callback function

```
_mqx_int data_transfer(void *app_data_ptr, uint32_t *tx_data_ptr, uint32_t
*rx_data_ptr)
{
    if(rx_data_ptr == NULL)
    {
        /* Fill tx data register by first word before first transfer */
        /* This will be executed in ecspi_slave_init function */
        *(uint8_t *)tx_data_ptr = tx_buffer;
    }
    else
    {
        /* Handle interrupt event */
        /* This will be executed in ISR */
        rx_buffer = *(uint8_t *)rx_data_ptr;
        *(uint8_t *)tx_data_ptr = tx_buffer;
        _lwsem_post(&TRANSFER_COMPLETE);
    }
    return MQX_OK;
}
```

The very first call of the callback function takes place during execution of ecspi_slave_init(). At that point there is no data received yet so the callback function is passed NULL pointer in place of the rx_data_parameter. The callback function shall recognize this case and act accordingly: typically provide feed dummy pattern to the tx buffer and possibly reset its internal state machine (if any).

Be aware that the data which are fed by the callback function to the tx buffer are not sent to the bus instantly. This data is just stored and prepared to be shifted out to the bus once the master starts next burst of clock pulses.

SPI slave can be shut down by function ecspi_slave_shutdown(). This function installs back default ISR, disables ECSPI block and resets related GPIO.

For more information see examples and readme files in:

- \mqx\examples\spi_slave\
- \mqx\examples\spi_master\

Chapter 7 I²C Driver

7.1 Overview

This chapter describes I²C device driver. The driver includes:

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

7.2 Source Code Location

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

7.3 Header Files

To use an I²C device driver, include the header files, *i2c.h,* and device-specific, *i2c_imx.h,* from *source\io\i2c* in your application or in the BSP file *bsp.h*. Use the header files according to the following table.

Driver	Header file
I²C interrupt-driven	• `i2c.h` • `i2c_imx.h`
I²C polled	• `i2c.h` • `i2c_imx.h`

The files *i2c_imx_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.

7.4 Installing Drivers

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

Driver	Installation function
Interrupt-driven	_imx_i2c_int_install()
Polled	_imx_i2c_polled_install()

7.4.1 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 i.MX processor family

```
typedef struct imx_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;
}IMX_I2C_INIT_STRUCT, * IMX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE). For the i.MX family devices, only MASTER MODE is supported.

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 (For legacy usage only, the driver will use buffer assigned in application.).

RX_BUFFER_SIZE - Rx buffer size (For legacy usage only, the driver will use buffer assigned in application.).

Example

The following code is an example for the i.MX processor family as it can be found in the appropriate BSP code. See, for example, the *init_i2c.c* file.

```
const IMX_I2C_INIT_STRUCT _bsp_i2c3_init = {
   3,                      /* I2C channel    */
   BSP_I2C3_MODE,          /* I2C mode       */
   BSP_I2C3_ADDRESS,       /* I2C address    */
```

```
   BSP_I2C3_BAUD_RATE,     /* I2C baud rate  */
   BSP_I2C3_INT_LEVEL,     /* I2C int level  */
   0,                   /* I2C int sublevel not available*/
   BSP_I2C3_TX_BUFFER_SIZE,/* I2C int tx buf */
   BSP_I2C3_RX_BUFFER_SIZE /* I2C int rx buf */
};
```

7.5 Driver Services

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

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

NOTE

The interrupt driven i2c driver for the i.MX family works in synchronous mode. This operation of the new driver is the same as that of the polled driven i2c driver, but more efficient.

7.6 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*.

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

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

NOTE

For the i.MX family device, only MASTER mode is supported.

7.7 Device States

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

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

7.8 Device Modes

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

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

7.9 Bus Availability

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

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

7.10 Error Codes

No additional error codes are generated.

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

Chapter 8 FlexCAN Driver

8.1 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.

8.2 Source Code Location

The source files for the FlexCAN driver are located in the `source\io\can\flexcan` directory.

8.3 Header Files

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

8.4 API Function Reference - FlexCAN Module Related Functions

This section provides function reference for the FlexCAN module driver.

8.4.1 flexcan_set_bitrate()

The function sets up all the time segment values.

Synopsis

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

Parameters

instance – The FlexCAN instance number.

bitrate – FlexCAN bit rate (Bit/s) in the flexcan_bitrate_table_t table.

Description

The function sets up all the time segment values. Those time segment values are from the table `flexcan_bitrate_table_t` and based on the bit rate in Bit/s passed in. Available bitrates supported are:

- 125000, /*!< 125 kBit/s*/
- 250000, /*!< 250 kBit/s*/
- 500000, /*!< 500 kBit/s*/
- 1000000, /*!< 1 MBit/s */

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_FREQUENCY (invalid bitrate)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN bitrate
uint8_t  instance = 1;
uint32_t result = flexcan_set_bitrate(instance, 1000000);
```

8.4.2 flexcan_get_bitrate()

This function gets the FlexCAN bitrate for specified.

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 flexcan_bitrate_table_t table.

Description

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

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_FREQUENCY(invalid bitrate)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Get FlexCAN bitrate
uint8_t instance = 1;
uint32_t bitrate_get;
uint32_t  result = flexcan_get_bitrate(instance, &bitrate_get);
```

8.4.3 flexcan_set_mask_type ()

This function sets mask type for FlexCAN Rx.

Synopsis

```
uint32_t 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 will set operation mode as freeze mode and set mask type, then de-assert freeze mode and wait till exit from freeze mode. Available mask types supported are:

- kFlexCanRxMask_Global
- kFlexCanRxMask_Individual

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN Rx mask type
uint8_t  instance = 1;
flexcan_rx_mask_type_t type = kFlexCanRxMask_Global;
Uint32_t result = flexcan_set_mask_type (instance, type);
```

8.4.4 flexcan_set_rx_fifo_global_mask ()

This function sets global standard or extended mask for FlexCAN Rx FIFO.

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 – Mailbox id type.

mask – Mask value will be set.

Description

This function will set operation mode as freeze mode and set global mask, then de-assert freeze mode and wait till exit from freeze mode. Available mailbox id type supported are:

- kFlexCanMbId_Std
- kFlexCanMbId_Ext

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_ID_TYPE(invalid id type)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN Rx fifo global mask
uint8_t  instance = 1;
flexcan_mb_id_type_t id_type = kFlexCanMbId_Std;
uint32_t mask = 0x7FF;
Uint32_t result = flexcan_set_rx_fifo_global_mask (instance, id_type, mask);
```

8.4.5 flexcan_set_rx_mb_global_mask ()

This function sets global standard or extended mask for FlexCAN Rx Message buffer.

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 – Mailbox id type.

mask – Mask value will be set.

Description

This function will set operation mode as freeze mode and set global mask, then de-assert freeze mode and wait till exit from freeze mode. Available mailbox id types supported are:

- kFlexCanMbId_Std
- kFlexCanMbId_Ext

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_ID_TYPE(invalid id type)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN Rx message buffer global mask
uint8_t  instance = 1;
flexcan_mb_id_type_t id_type = kFlexCanMbId_Std;
uint32_t mask = 0x7FF;
Uint32_t result = flexcan_set_rx_mb_global_mask (instance, id_type, mask);
```

8.4.6 flexcan_set_rx_mb_global_mask ()

This function sets global standard or extended mask for FlexCAN Rx FIFO.

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 – Mailbox id type.

mask – Mask value will be set.

Description

This function sets operation mode as freeze mode and set global mask, then de-assert freeze mode and wait till exit from freeze mode. Available mailbox id types supported are:

- kFlexCanMbId_Std
- kFlexCanMbId_Ext

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_ID_TYPE(invalid id type)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN Rx message buffer global mask
uint8_t  instance = 1;
flexcan_mb_id_type_t id_type = kFlexCanMbId_Std;
uint32_t mask = 0x7FF;
Uint32_t result = flexcan_set_rx_mb_global_mask (instance, id_type, mask);
```

8.4.7 flexcan_set_rx_individual_mask ()

This function sets individual standard or extended mask for FlexCAN Rx.

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 – Mailbox id type.

mb_idx – Index of the message buffer.

mask – Mask value will be set.

Description

This function sets operation mode as freeze mode and set individual mask, then de-assert freeze mode and wait till exit from freeze mode. Available mailbox id types supported are:

- kFlexCanMbId_Std
- kFlexCanMbId_Ext

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_ID_TYPE(invalid id type)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN Rx message buffer global mask
uint8_t  instance = 1;
flexcan_mb_id_type_t id_type = kFlexCanMbId_Std;
uint32_t mask = 0x7FF;
Uint32_t result = flexcan_set_rx_mb_global_mask (instance, id_type, mask);
```

8.4.8 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 – FlexCAN instance number.

data – FlexCAN platform data.

enable_err_interrupts – Enable error interrupt flag, true if enable it, false if not.

Description

This function initializes the FlexCAN device, selects operate mode, enables error and warning interrupt, and sets up an event group.

Return Value

- kFlexCan_OK (success)
- IO_ERROR (Error code returned by I/O functions)
- kFlexCan_UNDEF_ERROR (undefined instance)
- kFlexCan_INVALID_ADDRESS (undefined instance)
- kFlexCan_INIT_FAILED (Error code from software reset function)

Example

```
// Initialize the FlexCAN driver
flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;

uint32_t  result = flexcan_init(instance, &flexcan_data, TRUE);
```

8.4.9 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_index,
    flexcan_mb_code_status_tx_t *cs
uint32_t msg_id)
```

Parameters

instance – The FlexCAN instance number.

data – The FlexCAN platform data.

mb_index – Index of the message buffer.

cs – Tx code and status values.

msg_id – Id of the message to transmit.

Description

This function first configures FlexCAN Tx message buffer. Then enables interrupt for requested mailbox and enables the FlexCAN Message buffer interrupt.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexcan_NOT_SUPPORT (Not support)

Example

```
// Configure Tx buffer fields
flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;

flexcan_mb_code_status_tx_t tx_cs;
tx_cs. code = kFlexCanTX_Data
tx_cs.msg_id_type = kFlexCanMbId_Std;
tx_cs.data_length = 1;
tx_cs.substitute_remote = 0;
tx_cs.remote_transmission = 0;
tx_cs.local_priority_enable = 0;
tx_cs.local_priority_val = 0;

unit32_t mb_idx = 13;
uint32_t TX_identifier = 0x321;

uint32_t result = flexcan_tx_mb_config(instance, &flexcan_data, mb_idx, &tx_cs,
TX_identifier);
```

8.4.10 flexcan_send()

This function starts transmitting 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 – FlexCAN instance number

data – FlexCAN platform data

mb_idx – ID of the message to transmit

cs – Tx code and status values

msg_id – ID of the message to transmit

num_bytes – Number of bytes in message buffer

mb_data – Bytes of the message buffer to be transmitted

Description

This function sets up FlexCAN Tx buffer, copies user's buffer data into the message buffer data field, and wait for the transmission completed interrupt. Available message 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_MAILBOX(invalid mailbox id)
- kFlexcan_MESSAGE_FORMAT_UNKNOWN (invalid message format)
- kFlexcan_UNDEF_ERROR (Not defined)
- kFlexcan_NO_MESSAGE (No message data)

Example

```
// Configure Tx buffer fields
uint8_t instance = 1;
flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;

flexcan_mb_code_status_tx_t tx_cs;
```

```
tx_cs.code = kFlexCanTX_Data
tx_cs.msg_id_type = kFlexCanMbId_Std;
tx_cs.data_length = 1;
tx_cs.substitute_remote = 0;
tx_cs.remote_transmission = 0;
tx_cs.local_priority_enable = 0;
tx_cs.local_priority_val = 0;

uint32_t mb_idx = 13;
uint32_t msg_id = 0x321;
uint8_t mb_data = 1;

uint32_t result = flexcan_send(instance, &flexcan_data, mb_idx, &tx_cs, msg_id, 1,
&mb_data);
```

8.4.11 flexcan_rx_mb_config()

This function configures a FlexCAN 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 – FlexCAN instance number

data – FlexCAN platform data

mb_idx – Index of the message buffer

cs – Rx code and status values

msg_id – ID of the message to transmit

Description

This function will set FlexCAN Rx message buffer data field, install isr, enable interrupt line and enable message buffer interrupt.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX(invalid mailbox id)
- kFlexcan_NOT_SUPPORT (Not support)
- kFlexCan_UNDEF_ERROR(Not defined)
- kFlexCan_MESSAGE_FORMAT_UNKNOWN(invalid message format)

- kFlexCan_INT_INSTALL_FAILED(wrong interrupt vector)
- kFlexCan_INT_ENABLE_FAILED(interrupt enable failed)

Example

```
// Configure RX buffer fields
uint8_t instance = 1;
flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;
uint32_t mb_idx = 9;

flexcan_mb_code_status_tx_t tx_cs;
tx_cs. code = kFlexCanTX_Data
tx_cs.msg_id_type = kFlexCanMbId_Std;
tx_cs.data_length = 1;
tx_cs.substitute_remote = 0;
tx_cs.remote_transmission = 0;
tx_cs.local_priority_enable = 0;
tx_cs.local_priority_val = 0;
uint32_t msg_id = 0x321;

uint32_t result = flexcan_rx_mb_config(instance, &flexcan_data, mb_idx, &tx_cs,
msg_id);
```

8.4.12 flexcan_rx_fifo_config()

This function configures a FlexCAN FIFO fields for receiving data..

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 – FlexCAN instance number.

data – FlexCAN platform data.

Id_format – Format of Rx FIFO ID filter table elements.

id_filter_table – ID filter table, contains RX FITO ID filter elements.

Description

This function will set FlexCAN Rx FIFO data field, install isr, enable interrupt line and enable FIFO interrupt.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX(invalid mailbox id)
- kFlexcan_NOT_SUPPORT (Not support)
- kFlexCan_UNDEF_ERROR(Not defined)
- kFlexCan_MESSAGE_FORMAT_UNKNOWN(invalid message format)
- kFlexCan_INT_INSTALL_FAILED(wrong interrupt vector)
- kFlexCan_INT_ENABLE_FAILED(interrupt enable failed)

Example

```
// Configure RX buffer fields
uint8_t instance = 1;
flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;
flexcan_rx_fifo_id_element_format_t id_format = kFlexCanRxFifoIdElementFormat_A;

flexcan_id_table_t id_table;
id_table.is_extended_mb = 0;
id_table.is_remote_mb = 0;
uint32_t rx_fifo_id[8];
rx_fifo_id[0] = 0x666;
rx_fifo_id[1] = 0x667;
rx_fifo_id[2] = 0x676;
rx_fifo_id[3] = 0x66E;
rx_fifo_id[4] = 0x66F;
for (i = 5; i < 8; i++)
    rx_fifo_id[i] = 0x6E6;
id_table.id_filter = rx_fifo_id;

uint32_t result = flexcan_rx_fifo_config(instance, &flexcan_data,
kFlexCanRxFifoIdElementFormat_A, &id_table);
```

8.4.13 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 – FlexCAN instance number.

data – FlexCAN platform data.

mb_idx – Index of the message buffer.

msg_id – ID of the message to transmit.

receiveDataCount – Number of data to be received.

is_rx_mb_data – Whether data is from message buffer or not.

is_rx_fifo_data – Whether data is from fifo or not.

rx_mb – FlexCAN receive message buffer data.

rx_fifo – FlexCAN receive FIFO data.

Description

This function will set mask bit. Then will wait for event from interrupt handle and lock the Rx message buffer or Rx fifo before get receive data, message buffer or Rx fifo will be unlocked after get the receive data.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_UNDEF_ERROR(Not defined)
- kFlexCan_INVALID_MAILBOX(invalid mailbox id)

Example

```
// Start receiving data
uint8_t instance = 1;
flexcan_mb_t rx_mb;
flexcan_mb_t rx_fifo;
uint8_t instance;
bool is_rx_mb_data;
bool is_rx_fifo_data;

flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
```

```
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;
uint32_t mb_idx = 9;
uint32_t msg_id = 0x321;
uint32_t receiveDataCount = 1;

is_rx_fifo0_data = FALSE;
is_rx_fifo1_data = FALSE;

uint32_t result = flexcan_start_receive(instance, &flexcan_data, mb_idx, msg_id,
&receiveDataCount, &is_rx_fifo0_data, &is_rx_fifo1_data, &rx_mb, &rx_fifo);
```

8.4.14 flexcan_receive()

This function gets ready for receiving data.

Synopsis

```
uint32_t flexcan_receive(
    uint8_t instance,
    flexcan_config_t *data
    uint32_t mb_idx,
    flexcan_mb_code_status_rx_t *cs,
    uint32_t msg_id,
    flexcan_rx_fifo_id_element_format_t id_format,
    flexcan_id_table_t *id_filter_table,
    uint32_t receiveDataCount,
    flexcan_mb_t *rx_mb,
    flexcan_mb_t *rx_fifo)
```

Parameters

instance – FlexCAN instance number
data – FlexCAN platform data
mb_idx – Index of the message buffer
cs – Rx code and status values
msg_id – ID of the message to transmit.
id_format – Format of the Rx FIFO ID filter table elements
id_filter_table – ID filter table, contains RX FITO ID filter elements
receiveDataCount – Number of data to be received
rx_mb – FlexCAN receive message buffer data
rx_fifo – FlexCAN receive FIFO data

Description

This function first configures Rx FIFO fields or Rx message buffer fields, then call flexcan_start_receive() to start receiving data actually.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_UNDEF_ERROR (Not defined)
- kFlexCan_INVALID_MAILBOX (invalid mailbox ID)

Example

```
// Start receiving data
uint8_t instance = 1;
flexcan_mb_t rx_mb;
flexcan_mb_t rx_fifo;
uint8_t instance;
bool is_rx_mb_data;
bool is_rx_fifo_data;

flexcan_config_t flexcan_data;
flexcan_data.num_mb = 16;
flexcan_data.max_num_mb = 16;
flexcan_data.num_rximr = 64;
flexcan_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan_data.is_rx_fifo_needed = TRUE;
flexcan_data.is_rx_mb_needed = TRUE;

flexcan_mb_code_status_tx_t tx_cs;
tx_cs. code = kFlexCanTX_Data
tx_cs.msg_id_type = kFlexCanMbId_Std;
tx_cs.data_length = 1;
tx_cs.substitute_remote = 0;
tx_cs.remote_transmission = 0;
tx_cs.local_priority_enable = 0;
tx_cs.local_priority_val = 0;

uint32_t mb_idx = 9;
uint32_t msg_id = 0x321;
uint32_t receiveDataCount = 1;
flexcan_rx_fifo_id_element_format_t id_format  =  kFlexCanRxFifoIdElementFormat_A;

uint32_t rx_fifo_id[8];
flexcan_id_table_t id_table;
id_table.is_extended_mb = 0;
id_table.is_remote_mb = 0;
uint32_t rx_fifo_id[8];
rx_fifo_id[0] = 0x666;
```

```
rx_fifo_id[1] = 0x667;
rx_fifo_id[2] = 0x676;
rx_fifo_id[3] = 0x66E;
rx_fifo_id[4] = 0x66F;
for (i = 5; i < 8; i++)
    rx_fifo_id[i] = 0x6E6;
id_table.id_filter = rx_fifo_id;

uint32_t result = flexcan_receive(instance, &is_rx_fifo0_data, mb_idx, & tx_cs,
msg_id, id_format, &id_filter_table, &is_rx_fifo1_data, &rx_mb, &rx_fifo);
```

8.4.15 flexcan_shutdown()

The function shuts down a FlexCAN instance.

Synopsis

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

Parameters

instance – FlexCAN instance number

Description

This function will enter disable mode and disable FlexCAN module, then disable the FlexCAN clock.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_UNDEF_ERROR (not defined)

Example

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

8.4.16 flexcan_enter_stop_mode()

The function requests the FlexCAN hardware to enter stop mode.

Synopsis

```
uint32_t flexcan_enter_stop_mode(uint8_t instance)
```

Parameters

instance – FlexCAN instance number

Description

This function will request flexcan to enter stop mode.

Return Value

- kFlexcan_OK (success)
- Other value if not successful

Example

```
/* Request FlexCAN to enter stop mode */
uint8_t instance = 1;
uint32_t result = flexcan_enter_stop_mode(instance);
```

8.4.17 flexcan_exit_stop_mode()

The function requests the FlexCAN hardware to exit stop mode.

Synopsis

```
uint32_t flexcan_exit_stop_mode(uint8_t instance)
```

Parameters

instance – FlexCAN instance number

Description

This function will request flexcan to exit stop mode.

Return Value

- kFlexcan_OK (success)
- Other value if not successful

Example

```
/* Request FlexCAN to exit stop mode */
uint8_t instance = 1;
uint32_t result = flexcan_exit_stop_mode(instance);
```

8.4.18 flexcan_irq_handler()

The function is the interrupt handler for a FlexCAN.

Synopsis

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

Parameters

flexcan_ptr – Point to a FlexCAN instance

Description

The function is the interrupt handler for a FlexCAN. It first reads the interrupt flags. Then it check Tx/Rx interrupt flag and clear all interrupt flags.

Example

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

8.4.19 flexcan_int_enable()

This function enables the interrupt for the specified FlexCAN device and mailbox ID.

Synopsis

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

Parameters

dev_num – FlexCAN device number

mailbox_number – Mailbox index

Description

The function enables the specified FlexCAN interrupt source.

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_MAILBOX (invalid mailbox id)
- kFlexcan_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* Enable FlexCAN interrupt for specific FlexCAN instance and mailbox id */
uint8_t instance = 1;
uint32_t mailbox_number = 13;
uint32_t result = flexcan_int_enable(instance, mailbox_number);
```

8.4.20 flexcan_int_disable()

This function disables the interrupt for the specified FlexCAN device and mailbox ID.

Synopsis

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

Parameters

dev_num – FlexCAN device number.

mailbox_number – Mailbox index.

Description

The function will get the interrupt vector with specified device number and mailbox id, then disables the specified FlexCAN interrupt source.

Return Value

- kFlexcan_OK (success)
- kFlexCan_INVALID_MAILBOX (invalid mailbox id)
- kFlexcan_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* Disable FlexCAN interrupt for specific FlexCAN instance and mailbox id */
uint8_t instance = 1;
uint32_t mailbox_numnber = 13;
uint32_t result = flexcan_int_disable(instance, mailbox_numnber);
```

8.4.21 flexcan_install_isr()

This function installs the interrupt service routine for the specified FlexCAN device and mailbox ID.

Synopsis

```
uint32_t flexcan_install_isr(
uint8_t dev_num,
uint32_t mailbox_number,
INT_ISR_FPTR isr)
```

Parameters

dev_num – FlexCAN device number

mailbox_number – Mailbox index

isr – Interrupt service routine address

Description

The function will get the interrupt vector with specified device number and mailbox id, then install interrupt service routine.

Return Value

- kFlexcan_OK (success)
- kFlexcan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexcan_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void my_isr_function(void *  can_reg_base_ptr);
/* install interrupt service routine for FlexCAN 1*/
uint8_t  instance = 1;
INT_ISR_FPTR isr = my_isr_function;
uint32_t result = flexcan_install_isr(instance, isr);
```

8.5 Data Types

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

8.5.1 flexcan_time_segment

This structure can be used to set up time segments.

```
typedef struct flexcan_time_segment {
    uint32_t progseg;          //!< 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;
```

8.5.2 flexcan_mb

This structure can be used to configure the FlexCAN message buffer.

```
typedef struct flexcan_mb {
uint32_t cs;     //!< code and status
uint32_t msg_id;     //!< message buffer id
    uint8_t data[kFlexCanMessageSize];     //!< data of the FlexCAN message
} flexcan_mb_t;

kFlexCanMessageSize is defined as:
enum _flexcan_constants
{
    kFlexCanMessageSize = 8
};
```

8.5.3 flexcan_config

This structure can be used to configure FlexCAN device.

```
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
uint32_t num_rximr;             //!< The number of total RXIMR registers
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_rx_fifo_id_filter_num_t is defined as:
typedef enum _flexcan_rx_fifo_id_filter_number {
    kFlexCanRxFifoIDFilters_8   = 0x0,         //!<   8 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_16  = 0x1,         //!<  16 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_24  = 0x2,         //!<  24 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_32  = 0x3,         //!<  32 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_40  = 0x4,         //!<  40 Rx FIFO Filters
```

```
    kFlexCanRxFifoIDFilters_48  = 0x5,          //!<  48 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_56  = 0x6,          //!<  56 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_64  = 0x7,          //!<  64 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_72  = 0x8,          //!<  72 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_80  = 0x9,          //!<  80 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_88  = 0xA,          //!<  88 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_96  = 0xB,          //!<  96 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_104 = 0xC,          //!< 104 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_112 = 0xD,          //!< 112 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_120 = 0xE,          //!< 120 Rx FIFO Filters
    kFlexCanRxFifoIDFilters_128 = 0xF           //!< 128 Rx FIFO Filters
} flexcan_rx_fifo_id_filter_num_t;
```

8.5.4 flexcan_rx_fifo_config

This structure can be used to configure the FlexCAN Rx FIFO.

```
typedef struct flexcan_rx_fifo_config {
    flexcan_mb_id_type_t msg_id_type;  //!< Type of message ID
    uint32_t data_length;              //!< Length of Data in Bytes
uint32_t substitute_remote;        //!< bytes of the FlexCAN message
uint32_t remote_transmission;      // !< Remote transmission request
flexcan_rx_fifo_id_element_format_t id_filter_number;    //! < The number
                                   //! of RX FIFO ID filters
} flexcan_rx_fifo_config;

flexcan_rx_fifo_id_element_format_t id defined as:
typedef enum _flexcan_rx_fifo_id_element_format {
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.
} flexcan_rx_fifo_id_element_format_t;
```

8.6 Error Codes

The FlexCAN driver defines the following error codes:

Error code	Description
kFlexcan_OK	Success
kFlexcan_UNDEF_ERROR	Unknown error
kFlexcan_NOT_SUPPORT	Not support

Error code	Description
kFlexcan_NO_MESSAGE	No message received
kFlexcan_INVALID_ADDRESS	Wrong device specified
kFlexcan_INVALID_BITRATE	Wrong bitrate setting
kFlexcan_INT_ENABLE_FAILED	MQX interrupt enabling failed
kFlexcan_INT_DISABLE_FAILED	MQX interrupt disabling failed
kFlexcan_INT_INSTALL_FAILED	MQX interrupt installation failed
kFlexcan_DATA_SIZE_ERROR	Data length not in range 0..8
KFlexcan_MESSAGE_FORMAT_UNKNOWN	Wrong message format specified
kFlexcan_INVALID_ID_TYPE	Invalid ID type

8.7 Example

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

Chapter 9 LWGPIO Driver

9.1 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.

9.2 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.

9.3 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*.

9.4 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.

9.4.1 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);
```

9.4.2 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);
}
```

9.4.3 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);
```

9.4.4 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);
```

9.4.5 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);
```

9.4.6 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);
```

9.4.7 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);
```

9.4.8 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);
```

9.4.9 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);
```

9.4.10 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);
}
```

9.4.11 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);
```

9.4.12 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 */
}
```

9.4.13 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);
```

9.4.14 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);
```

9.5 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.

9.5.1 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);
```

9.5.2 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);
```

9.5.3 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);
}
```

9.6 Data Types Used by the LWGPIO API

The following data types are used within the LWGPIO driver.

9.6.1 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.

9.6.2 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.

9.6.3 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.

9.6.4 LWGPIO_VALUE

This enumerated value specifies the voltage value of the pin. The value is generic.

```
typedef enum {
    LWGPIO_VALUE_LOW,
    LWGPIO_VALUE_HIGH,
    LWGPIO_VALUE_NOCHANGE
} LWGPIO_VALUE;
```

The LWGPIO_VALUE enum type is used to set or get the voltage value of the specified pin. The special value of LWGPIO_VALUE_NOCHANGE can be passed to a function if the change of the value is undesirable or it is returned in special case if the value can not be obtained.

9.6.5 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.

9.7 Example

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

Chapter 10 Low Power Manager

10.1 Overview

This chapter describes the M4 low power management (LPM) driver. i.MX 6SoloX has two asynchronous cores. The ARM® Cortex®-A9 (A9 for short) processor is the main low power controller, the ARM® Cortex®-M4 (M4 for short) processor cannot control the system-level low power mode, but it should maintain the correct power status in the M4 domain and report it to A9. The coordination of the two cores achieves system-level power saving. The M4 LPM driver provides an interface for the user to achieve this.

10.2 Mechanism

In the i.MX 6SoloX, A9 and M4 can run different operating systems (OSs). Different on chip peripheral works under one of the two cores, forming two virtual domains -- the A9 virtual domain and the M4 virtual domain. Considering system-level low power management, the M4 virtual domain, including attached peripherals, becomes a virtual-peripheral to A9. A9 supervise all its peripheral's running status (including the M4 virtual-peripheral) to make low power management decisions and perform operations accordingly.

The M4 LPM driver's responsibility is to keep track of all the attached peripherals on its virtual domain. When all its peripherals are in idle state, it can release occupied power resources and inform A9 about this. A9 can then power down these resources to achieve system level power saving. During this period, M4 core keeps in WFI state, the core stops running.

When an interrupt happens, which requires M4 to return to normal running mode, the M4 LPM driver first requires A9 to prepare all the required power resources. After that, M4 returns to normal operational mode, until the next time it can enter low power state again.

The mechanism contains two handshakes between M4 and A9. The handshake is implemented through the MU module. M4 initiates the handshake and wait for A9's acknowledgment. During the handshake, M4 must respond to nothing except A9's MU message, this ensures when A9 do low power operations. M4 sticks to WFI state and executes no instructions.

It is also required that during this phase, M4 must execute its code in memory that has no requirement for power resources. In a typical situation, M4 runs its program in QSPI with a XIP manner. When A9 does low power operations, XIP needed resources, such as power supply and clock input to QSPI, may be unstable. To make sure that M4 runs code safely when A9 does such operations, M4 should migrate to run code from TCM.

Figure 10-1. shows the procedure of low power handshake.

The low power handshake mechanism is designed to be executed automatically in idle task. The driver provides an API for the user to turn on or turn off the logic. When it is turned off, M4 core will simply execute a WFI. A9 will not be notified and therefore does not perform low power operations.

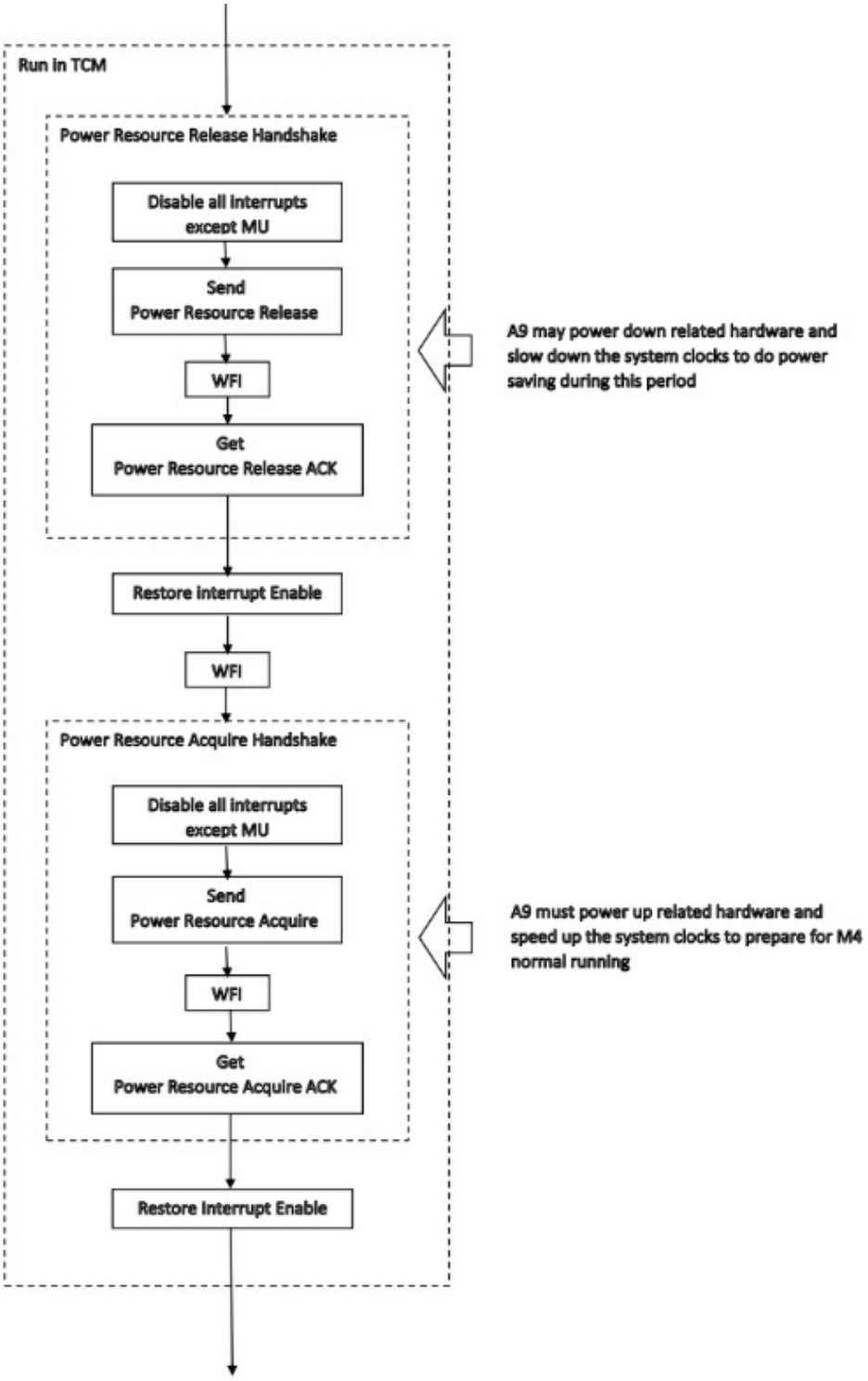

Figure 10-1. M4 Low Power Handshake Mechanism

10.3 Source Code Location

The source code of the M4 LPM driver is located in the mqx\source\io\lpm_mcore directory.

10.4 Header Files

To use an M4 LPM driver, include the header files lpm_mcore.h from mqx\source\io\lpm_mcore in your application or in the BSP file bsp.h.

10.5 Installing Driver

The M4 LPM device driver provides an installation function _io_mcore_lpm_init() that the BSP calls. The function makes a runtime code copy from ROM to TCM. It puts all the low power handshake mechanism codes into TCM to meet the requirement described in the Mechanism.

M4 LPM device driver installation:

```
#if MQX_ENABLE_MCORE_LPM
   _io_mcore_lpm_init();
#endif
```

This code is located in the mqx\source\bsp\<board>\init_bsp.c file.

10.6 Driver Services

The table below describes the M4 LPM device driver services:

API	Description
_io_mcore_lpm_get_status()	Gets current M4 domain power status
_io_mcore_lpm_set_status()	Sets current M4 domain power status
_io_mcore_lpm_register_peer_wakeup()	Register speripheral in M4 domain as a direct wakeup source to A9

10.6.1 Macro Definition

Macro	Description
STATUS_NORMAL_RUN	M4 platform is in normal running mode, and A9 cannot power down necessary resources.
STATUS_LOWPOWER_RUN	M4 platform is in low power idle mode, and A9 can power down released resources.
WAKEUP_ENABLE	Used by _io_mcore_lpm_register_peer_wakeup as input parameter. Enable an M4 peripheral as direct wakeup source to A9.
WAKEUP_DISABLE	Used by _io_mcore_lpm_register_peer_wakeup as input parameter. Disable an M4 peripheral as direct wakeup source to A9.
WAKEUP_REGISTER_SUCCESS	_io_mcore_lpm_register_peer_wakeup return value. It indicates the registration of wakeup source to A9 success.
WAKEUP_REGISTER_FAILURE	_io_mcore_lpm_register_peer_wakeup return value. Mean the registration of wakeup source to A9 fails. In this case M4 cannot enter low power running mode.

10.6.2 _io_mcore_lpm_get_status

```
uint32_t _io_mcore_lpm_get_status(void)
```

Parameters

NA

Return Value

The function returns STATUS_NORMAL_RUN, if current M4 virtual-domain is in full-running mode. It returns STATUS_LOWPOWER_RUN, if current M4 virtual-domain is in low power running mode.

10.6.3 _io_mcore_lpm_set_status

```
void _io_mcore_lpm_set_status(uint32_t status)
```

Parameters

status - If the parameter of STATUS_NORMAL_RUN is passed, the function sets the M4 virtual platform in normal running mode. In this case, the driver will bypass the low power handshake mechanism with A9 and prevent A9 from powering down necessary resources. If the parameter of STATUS_LOWPOWER_RUN is passed, the function sets the M4 virtual platform in low power running mode. In this case, the low power handshake mechanism with A9 will be performed with A9 to allow A9 to power down the resources for power saving.

Return Value

NA

Remarks

M4 platform developer should keep aware of the current M4 virtual-platform power status. Only when all the peripheral is in idle state, it can call this function with STATUS_LOWPOWER_RUN parameter to turn on the low power handshake logic.

10.6.4 _io_mcore_lpm_register_peer_wakeup

```
uint32_t _io_mcore_lpm_register_peer_wakeup(
     uint32_t int_no,
     uint32_t enable
)
```

Parameters

int_no - wakeup interrupt number to be registered

enable - pass WAKEUP_ENABLE to enable the wakeup interrupt, pass WAKEUP_DISABLE to disable the wakeup interrupt.

Return value

The function returns WAKE UP_REGISTER_SUCCESS to indicate that the registration to A9 succeeds. In this case, the M4 can continue to call _io_mcore_lpm_set_status() with STATUS_LOWPOWER_RUN to begin the low power handshake mechanism with A9. The function returns WAKEUP_REGISTER_SUCCESS to indicate that the registration to A9 fails. In this case, M4 cannot call _io_mcore_lpm_set_status() with STATUS_LOWPOWER_RUN to do low power handshake mechanism with A9.

Remarks

After M4 calls the _io_mcore_lpm_set_status() with STATUS_LOWPOWER_RUN, A9 may go into deep sleep mode, which will hold M4 in stop mode. In this case, M4 will not respond to any interrupt. Therefore, this API should be invoked to register a M4 peripheral as wakeup source to A9. The interrupt will first wakeup A9. A9 will then stop holding M4 in stop mode. After that, normal handshake mechanism will be resumed.

10.7 Example

The source code of the M4 LPM driver example is located in the mqx\examples\randwfi directory.

Chapter 11 LWADC Driver

11.1 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.

11.2 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.

11.3 Header Files

To use LWADC driver, include the *lwadc.h* header file and the platform specific header file (e.g. *lwadc_mpxs30.h*) in your application or in the BSP header file (*bsp.h*). The platform specific header should be included before *lwadc.h.*

11.4 API Function Reference

This section contains the function reference for the LWADC driver.

11.4.1 _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.

11.4.2 _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.

11.4.3 _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.

11.4.4 _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.

11.4.5 _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.

11.4.6 _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.

ATTRIBUTE	Used to set or obtain:
LWADC_RESOLUTION	ADC Device resolution in steps.
LWADC_REFERENCE	ADC Reference voltage in millivolts.
LWADC_ FREQUENCY	ADC module base frequency, actual relation between this parameter and sampling rate parameter is device specific.
LWADC_DIVIDER	The input divider.
LWADC_DIFFERENTIAL	Enables channel as a differential input.
LWADC_POWER_DOWN	Power up or down the ADC Device.
LWADC_NUMERATOR	Numerator to be used on this channel for channel scaling.
LWADC_DENOMINATOR	Denominator to be used on this channel for channel scaling.
LWADC_FORMAT	Channel data format (such as left/right aligned).
LWADC_INPUT_CONVERSION_ENABLE	Enable or disable conversion for the input.

NOTE

Not all ADC devices will support all attributes, nor will all ADCs support a per-input setting of the attributes. Setting an attribute on one input may affect other or all inputs on a device.

11.4.7 _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().

11.4.8 _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.

11.5 Data Types Used by the LWADC API

The following data types are used within the LWADC driver.

11.5.1 LWADC_INIT_STRUCT

This device-specific structure contains necessary parameters for initialization of ADC module on a particular platform.

Synopsis for MPXSxx family:

```
typedef struct lwadc_init_struct {
    uint32_t device;
    uint32_t format;
    uint32_t clock;
    uint32_t reference;
} LWADC_INIT_STRUCT, * LWADC_INIT_STRUCT_PTR;
```

Parameters

device — Device number to initialize.
format — Preset data format, see LWADC_FORMAT attribute.
clock — ADC module clock frequency.
reference — Preset reference voltage in millivolts, see LWADC_REFERENCE attribute.

11.5.2 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.

11.5.3 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

11.6 Example

An example application demonstrating LWADC usage is provided. The example application can be found in `\mqx\examples\lwadc` directory.

Chapter 12 HWTIMER Driver

12.1 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.

12.2 Source Code Location

The source code for HWTIMER drivers is located in `source\io\hwtimer` directory.

12.3 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*.

12.4 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.

12.4.1 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, please refer to the chapter about the low layer driver below.

The meaning of the numerical identifier varies depending on the low layer driver used. Typically, it identifies a particular timer channel to initialize.

The initialization function has to be called prior to using any other HWTIMER driver API function.

12.4.2 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.

12.4.3 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.

12.4.4 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.

12.4.5 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.

12.4.6 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.

12.4.7 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.

12.4.8 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.

12.4.9 hwtimer_stop()

Synopsis

```
_mqx_int hwtimer_stop
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Hwtimer stop succeeded)
- Error - Otherwise

Description

The timer stops counting after this function is called. Pending interrupts and callbacks are canceled.

12.4.10 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).

12.4.11 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().

12.4.12 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.

12.4.13 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.

12.4.14 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.

12.4.15 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.

12.5 Data Types Used by the HWTIMER API

The following data types are used within the HWTIMER driver.

12.5.1 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.

12.5.2 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.

12.5.3 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.

12.6 Low Level Drivers Specifications

This chapter describes features related to various low level driver implementation. Currently only PIT timer module is supported. The implementation will be extended to other timer modules in the upcoming MQX releases.

12.6.1 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.

12.7 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 13 MMA8451Q Digital Accelerometer Driver

13.1 Overview

This chapter describes the MMA8451Q device driver. The driver defines interface for MMA8451Q Three-Axis Digital Accelerometer and accompanies the MQX release.

13.2 Source Code Location

The source code of the MMA8451Q driver is located in `mqx\source\io\sensor\mma8451q` directory.

13.3 Header Files

- To use the MMA8451Q device driver, include the header file *mma8451q.h* in your application.
- The file *mma8451q_basic.h* contains basic level I/O driver declarations and useful macros.
- The file *mma8451q_generic.h* contains generic usage functional level I/O driver function declarations.
- The file *mma8451q_ff_mt.h* contains motion and freefall detection feature related function declarations and useful macros.
- The file *mma8451q_lapo.h* contains Portrait/Landscape detection feature related function declarations and useful macros.
- The file *mma8451q_pulse.h* contains pulse detection feature related function declarations and useful macros.
- The file *mma8451q_tran.h* contains transient detection feature related function declarations and useful macros.
- The file *mma8451q_reg.h* contains register definitions and bit field masks. User can use this file with mma8451q basic level driver.
- The file *mma8451q_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.

13.4 MMA8451Q Driver API Description

The following section lists the various functions of the MMA8451Q library.

The mma8451q driver is divided into two layers: basic level driver and functional level driver. Functional level driver is also divided into five parts: generic function part, Freefall & Motion detection part, Orientation detection part, Tap (pulse) detection part, and Transient detection part. Using the part needed in your application can reduce total code size.

13.4.1 How To Use This Driver

1. Open an i2c connection by using the **fopen()** function and store the i2c pointer in a MQX_FILE_PTR type variable for the need of **mma8451q_init()** function.
2. Set the i2c bus to master mode and set the bus speed according to your application requirement.
3. Program the slave address, output data rate, full scale range, active power scheme and burst read mode using the **mma8451q_init()** function.
4. Optionally you can configure parameters without re-initialization using Initialization and Configuration Functions. (i.e there is no need to call again **mma8451q_init()** function).
5. Set data correction offset using **mma8451q_set_user_offset()** function.
6. Configure mma8451q internal FIFO data buffer and other embedded functions, if needed. For more information about how to configure the data FIFO and embedded function, please refer to correlative Freescale Application Note or take the example as a reference. All the examples available are listed in Section “Example” of this chapter.
7. Configure the interrupt function, if interrupt driven operation is needed.
   Interrupt configuration progress:
   1) Set interrupt polarity.
   2) Set interrupt output mode.
   3) Set interrupt pin route configurations.
   4) Enable interrupt for specified interrupt source.
   5) Enable and configure the corresponding GPIO pin interrupt which connected to mma8451q INT pin.
8. Switch the device to active mode using **mma8451q_set_operating_mode()** function. After that, the sensor will acquire data and store the data into internal register continuously, and all the embedded functions enabled start to work.

13.4.2 Initialization and Configuration Functions

This section provides a set of functions which can re-configure the generic features after device initialization.
Some of the functions can be only used in STANDBY mode, make sure that MMA8451Q is working in STANDBY mode before call these functions. For further information, please refer to function’s description.

- **mma8451q_init()**
- **mma8451q_deinit()**
- **mma8451q_set_slave_address()**
- **mma8451q_set_output_data_rate()**
- **mma8451q_set_power_scheme()**
- **mma8451q_set_full_scale_range()**
- **mma8451q_set_burst_read_mode()**
- **mma8451q_set_user_offset()**
- **mma8451q_set_self_test_state()**

- **mma8451q_reset_sensor()**
- **mma8451q_set_operating_mode()**
- **mma8451q_set_aslp_output_data_rate()**
- **mma8451q_set_aslp_power_scheme()**
- **mma8451q_set_wake_up_bypass()**
- **mma8451q_set_aslp_count()**
- **mma8451q_set_aslp_state()**
- **mma8451q_set_low_noise_state()**
- **mma8451q_set_hpf_cutoff()**
- **mma8451q_set_hpf_state()**

13.4.3 Basic I/O Functions

This section provides a set of functions which can be used to access MMA8451Q internal register through i2c bus.
Basic I/O level driver aims at provide user a fundamental interface with MMA8451Q. User can write their own high level driver based on it. To use basic I/O functions, just include *mma8451q.h* in your application and call basic level driver after **mma8451q_init()**. *mma8451q_basic.h* include basic level driver function declarations and many useful macros which can reduce user programming difficulty.

NOTE

User can use basic I/O functions and functional driver functions interlaced in their application.

- **mma8451q_write_reg()**
- **mma8451q_read_reg()**
- **mma8451q_write_single_reg()**
- **mma8451q_read_single_reg()**
- **mma8451q_modify_bitField()**
- **mma8451q_get_bitField()**

13.4.4 Data Acquisition Functions

This section provides a set of functions which can be used to get acceleration output data from the sensor. For more information, please refer to the function description.

- **mma8451q_get_acc_data()**
- **mma8451q_get_acc_from_fifo()**

13.4.5 FIFO Data Buffer Configuration Functions

This section provides a set of functions which are related to mma8451q internal FIFO data buffer configuration.

MMA8451Q contains a 32 sample internal FIFO data buffer minimizing traffic across the I2C bus. The FIFO can also provide power savings of the system by allowing the host processor/MCU to go into a SLEEP mode while the accelerometer independently stores the data, up to 32 samples per axis.
For details on the configurations for the FIFO buffer as well as more specific examples and application benefits, refer to Freescale application note, AN4073.

- **mma8451q_set_fifo_watermark()**
- **mma8451q_get_fifo_watermark()**
- **mma8451q_set_fifo_mode()**
- **mma8451q_get_fifo_mode()**
- **mma8451q_set_fifo_trigger_source()**
- **mma8451q_get_fifo_trigger_source()**
- **mma8451q_set_fifo_gate()**
- **mma8451q_get_fifo_gate()**
- **mma8451q_get_fifo_status()**
- **mma8451q_get_fifo_count()**

13.4.6 Interrupt Configuration Functions

This section provides a set of functions which are related to mma8451q internal interrupt controller configuration.
There are seven configurable interrupts in the MMA8451Q: Data Ready, Motion/Freefall, Tap (Pulse), Orientation, Transient, FIFO and Auto-SLEEP events. These seven interrupt sources can be routed to one of two interrupt pins. The interrupt source must be enabled and configured. If the event flag is asserted because the event condition is detected, the corresponding interrupt pin, INT1 or INT2, will assert.

- **mma8451q_set_int_polarity()**
- **mma8451q_get_int_polarity()**
- **mma8451q_set_int_output_mode()**
- **mma8451q_get_int_output_mode()**
- **mma8451q_set_int_pin_route()**
- **mma8451q_get_int_pin_route()**
- **mma8451q_set_int_state()**
- **mma8451q_get_int_state()**
- **mma8451q_get_int_source()**

13.4.7 Motion and Freefall Detection Functions

This section provides a set of functions which are related to mma8451q motion and freefall detection configuration.
MMA8451Q has flexible interrupt architecture for detecting either a Freefall or a Motion. Freefall can be enabled where the set threshold must be less than the configured threshold, or motion can be enabled where the set threshold must be greater than the threshold.

For details on the Freefall and Motion detection with specific application examples and recommended configuration settings, refer to Freescale application note, AN4070.

- **mma8451q_set_ff_mt_db_cnt_mode()**
- **mma8451q_get_ff_mt_db_cnt_mode()**
- **mma8451q_set_ff_mt_db_cnt()**
- **mma8451q_get_ff_mt_db_cnt()**
- **mma8451q_set_ff_mt_threshold()**
- **mma8451q_get_ff_mt_threshold()**
- **mma8451q_set_ff_mt_event_latch_state()**
- **mma8451q_get_ff_mt_event_latch_state()**
- **mma8451q_set_ff_mt_selection()**
- **mma8451q_get_ff_mt_selection()**
- **mma8451q_set_ff_mt_state()**
- **mma8451q_get_ff_mt_state()**
- **mma8451q_get_ff_mt_status()**

13.4.8 Portrait/Landscape detection Functions

This section provides a set of functions which are related to mma8451q portrait/landscape detection. The MMA8451Q incorporates an advanced algorithm for orientation detection (ability to detect all 6 orientations) with configurable trip points. The embedded algorithm allows the selection of the mid point with the desired hysteresis value.

For further information on the configuration settings of the orientation detection function, including recommendations for configuring the device to support various application use cases, refer to Freescale application note, AN4068.

- **mma8451q_set_lapo_db_cnt_mode()**
- **mma8451q_get_lapo_db_cnt_mode()**
- **mma8451q_set_lapo_db_cnt()**
- **mma8451q_get_lapo_db_cnt()**
- **mma8451q_set_back_front_threshold()**
- **mma8451q_get_back_front_threshold()**
- **mma8451q_set_lapo_threshold()**
- **mma8451q_get_lapo_threshold()**
- **mma8451q_set_z_lock_threshold()**
- **mma8451q_get_z_lock_threshold()**
- **mma8451q_set_lapo_trip_hys()**
- **mma8451q_get_lapo_trip_hys()**
- **mma8451q_set_lapo_state()**
- **mma8451q_get_lapo_state()**
- **mma8451q_get_lapo_status()**

13.4.9 Pulse Detection Functions

This section provides a set of functions which are related to mma8451q tap(pulse) detection.
The MMA8451Q has embedded single/double and directional tap detection. This function has various customizing timers for setting the pulse time width and the latency time between pulses. There are programmable thresholds for all three axes.
For more information on how to configure the device for tap detection please refer to Freescale application note, AN4072.

- **mma8451q_set_double_pulse_abort()**
- **mma8451q_get_double_pulse_abort()**
- **mma8451q_set_pulse_threshold()**
- **mma8451q_get_pulse_threshold()**
- **mma8451q_set_pulse_time_limit()**
- **mma8451q_get_pulse_time_limit()**
- **mma8451q_set_pulse_latency()**
- **mma8451q_get_pulse_latency()**
- **mma8451q_set_pulse_time_window()**
- **mma8451q_get_pulse_time_window()**
- **mma8451q_set_pulse_event_latch_state()**
- **mma8451q_get_pulse_event_latch_state()**
- **mma8451q_set_pulse_hpf_state()**
- **mma8451q_get_pulse_hpf_state()**
- **mma8451q_set_pulse_lpf_state()**
- **mma8451q_get_pulse_lpf_state()**
- **mma8451q_set_pulse_detect_state()**
- **mma8451q_get_pulse_detect_state()**
- **mma8451q_get_pulse_detect_status()**

13.4.10 Transient Detection Functions

This section provides a set of functions which are related to mma8451q transient detection.
The embedded Transient Detection function uses the high-pass filtered data allowing the user to set the threshold and debounce counter. The transient detection feature can be used in the same manner as the motion detection by bypassing the high-pass filter.
For details on the benefits of the embedded Transient Detection function along with specific application examples and recommended configuration settings, please refer to Freescale application note, AN4071.

- **mma8451q_set_transient_db_cnt_mode()**
- **mma8451q_get_transient_db_cnt_mode()**
- **mma8451q_set_transient_db_cnt()**
- **mma8451q_get_transient_db_cnt()**
- **mma8451q_set_transient_threshold()**

- **mma8451q_get_transient_threshold()**
- **mma8451q_set_transient_event_latch_state()**
- **mma8451q_get_transient_event_latch_state()**
- **mma8451q_set_transient_bypass_hpf()**
- **mma8451q_get_transient_bypass_hpf()**
- **mma8451q_set_transient_state()**
- **mma8451q_get_transient_state()**
- **mma8451q_get_transient_status()**

13.4.11 Status Inquiry Functions

This section provides a set of functions which can be used to get the status of current generic configuration or useful information (for example device id).

- **mma8451q_get_slave_address()**
- **mma8451q_get_dr_status()**
- **mma8451q_get_device_id()**
- **mma8451q_get_system_mode()**
- **mma8451q_get_output_data_rate()**
- **mma8451q_get_power_scheme()**
- **mma8451q_get_full_scale_range()**
- **mma8451q_get_burst_read_mode()**
- **mma8451q_get_user_offset()**
- **mma8451q_get_self_test_state()**
- **mma8451q_get_senor_reset_state()**
- **mma8451q_get_operating_mode()**
- **mma8451q_get_aslp_output_data_rate()**
- **mma8451q_get_aslp_power_scheme()**
- **mma8451q_get_wake_up_bypass()**
- **mma8451q_get_aslp_count()**
- **mma8451q_get_aslp_state()**
- **mma8451q_get_low_noise_state()**
- **mma8451q_get_hpf_cutoff()**
- **mma8451q_get_hpf_state()**

13.4.12 Function Descriptions

This section describes the MMA8451Q driver functions in details.

13.4.12.1 Initialization and Configuration Functions

13.4.12.1.1 mma8451q_init

Function Name:
void * mma8451q_init

```
(
    MMA8451Q_INIT_STRUCT   *mma8451q_init_ptr,
    MQX_FILE_PTR            fd
)
```

Function Description:

Initialize MMA8451Q with parameter set in MMA8451Q initialize structure. This function should be called after i2c driver initialization and before other mma8451q driver be called. It will initialize the mma8451q slave address, output data rate, full scale range, active power scheme and burst read mode defined in **MMA8451Q_INIT_STRUCT**. After initialization, the mma8451q will stay in **STANDBY** operation mode.

Parameters:

- *mma8451q_init_ptr[IN]*: MMA8451Q Init structure pointer.
- *fd[IN]*: File pointer for the I2C channel connected to mma8451q.

Return Value:

- **MMA8451Q_handle**: If initialize successful.
- **NULL**: If mma8451q initialize failed.

Note:

None.

13.4.12.1.2 mma8451q_deinit

Function Name:

```
bool mma8451q_deinit
(
    void    *mma8451q_handle
)
```

Function Description:

mma8451q_deinit() function will force the device operation mode back into **STANDBY** mode and recover all the mma8451q register content to the default value.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.1.3 mma8451q_set_slave_address

Function Name:

```
bool mma8451q_set_slave_address
(
    void      *mma8451q_handle,
    uint8_t    slave_address
)
```

Function Description:

Configure the mma8451q driver's internal data structure slave address field. The slave address is used when MCU communicate with mma8451q during address cycle.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *slave_address[IN]*: Slave address to be set.

Return Value:

- **TRUE** if successful.

Note:

Slave address should be set according to mma8451q **SA0** Pin connection. If **SA0** connect to logic 0, slave address will be 0x1C. If **SA0** connect to logic 1, slave address will be 0x1D. For more information, please refer to MMA8451Q datasheet.

13.4.12.1.4 mma8451q_set_output_data_rate

Function Name:

```
bool mma8451q_set_output_data_rate
(
    void       *mma8451q_handle,
    uint8_t     output_rate
)
```

Function Description:

This function configures the output data rate of accelerometer. This value is closely related to power consumption and resolution and should be set according to application requirement.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *output_rate[IN]*: Output data rate to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with output data rate macro in mma8451q_basic.h.

13.4.12.1.5 mma8451q_set_power_scheme

Function Name:

```
bool mma8451q_set_power_scheme
(
    void       *mma8451q_handle,
    uint8_t     power_scheme
)
```

Function Description:

This function configures the oversample modes of the accelerometer. This value is closely related to power consumption and resolution and should be set according to application requirement.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *power_scheme [IN]*: Power scheme to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with active power scheme macro in mma8451q_basic.h.

13.4.12.1.6 mma8451q_set_full_scale_range

Function Name:

```
bool mma8451q_set_full_scale_range
(
    void       *mma8451q_handle,
    uint8_t     full_scale
)
```

Function Description:

This function is used to set the full scale range of the accelerometer. The full scale range can be set to 2G, 4G and 8G and should be set according to application requirement.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *full_scale[IN]*: Full scale range to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with full scale range macro in mma8451q_basic.h.

13.4.12.1.7 mma8451q_set_burst_read_mode

Function Name:

```
bool mma8451q_set_burst_read_mode
(
    void       *mma8451q_handle,
    uint8_t     read_mode
)
```

Function Description:

This function is used to enable or disable the burst read mode of the sensor. This field should be set according to application requirement. For application need precise output data, burst read mode should be set to **NORMAL_MODE**. In this mode, accelerometer output data will be 14-bit width. For application need higher i2c bus access speed, burst read mode should be set to **BURST_READ_MODE**. In this mode, accelerometer output data will be 8-bit width.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *read_mode[IN]*: Burst read mode to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with burst read mode macro in mma8451q_basic.h.

13.4.12.1.8 mma8451q_set_user_offset

Function Name:

```
bool mma8451q_set_user_offset
(
    void       *mma8451q_handle,
    int8_t      offset_x,
    int8_t      offset_y,
    int8_t      offset_z
)
```

Function Description:

This function is used to calibrate the 0g offset. The 2's complement offset correction values are used to realign the Zero-g position of the X, Y, and Z-axis after device board mount. The resolution of the offset registers is 2 mg per LSB. The 2's complement 8-bit value would result in an offset compensation range ±256 mg.

For more information on how to calibrate the 0g offset, refer to application note AN4069.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *offset_x[IN]*: User Offset Correction on x axis to be set.
- *offset_y[IN]*: User Offset Correction on y axis to be set.
- *offset_z[IN]*: User Offset Correction on z axis to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- The resolution of the user offset is 2 mg per LSB. The 2's complement 8-bit value would result in an offset compensation range ±256 mg.

13.4.12.1.9 mma8451q_set_self_test_state

Function Name:

```
bool mma8451q_set_self_test_state
(
    void       *mma8451q_handle,
    uint8_t     st_enabled
)
```

Function Description:

This function configures the self test enable/disable state. When Self-Test is activated, the sensor outputs will exhibit a change in their DC levels which are related to the selected full scale through the device sensitivity. The device output level is given by the algebraic sum of the signals produced by the acceleration acting on the sensor and by the electrostatic test-force.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *st_enabled[IN]*: Self test function enable state.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with self test state macro in mma8451q_basic.h.

13.4.12.1.10 mma8451q_reset_sensor

Function Name:

```
bool mma8451q_reset_sensor
(
    void       *mma8451q_handle
)
```

Function Description:

Calling this function will activate the software reset. When this function is called, all registers are rest and are loaded with default values. This function can be used, no matter whether it is in **ACTIVE/WAKE**, **ACTIVE/SLEEP**, or **STANDBY** mode.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.

Return Value:

- **TRUE** if successful.

Note:

The I2C communication system is reset to avoid accidental corrupted data access during reset progress.

13.4.12.1.11 mma8451q_set_operating_mode

Function Name:

```
bool mma8451q_set_operating_mode
(
```

```
    void        *mma8451q_handle,
    uint8_t      operating_mode
)
```

Function Description:

This function is used to set the operation mode of the sensor. **ACTIVE** mode will make periodic measurements based on values programmed in the output data rate and power scheme fields.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *operating_mode[IN]*: Operating mode to be set.

Return Value:

- **TRUE** if successful.

Note:

- Should be used with operating mode macro in mma8451q_basic.h.
- Both **SLEEP** and **WAKE** modes are **ACTIVE** modes.

13.4.12.1.12 mma8451q_set_aslp_output_data_rate

Function Name:

```
bool mma8451q_set_aslp_output_data_rate
(
    void        *mma8451q_handle,
    uint8_t      output_rate
)
```

Function Description:

This function set the output data rate of SLEEP mode. For more information about output data rate, please refer to **mma8451q_set_output_data_rate()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *output_rate[IN]*: Sleep mode data output rate to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with auto sleep output data rate macro in mma8451q_basic.h.

13.4.12.1.13 mma8451q_set_aslp_power_scheme

Function Name:

```
bool mma8451q_set_aslp_power_scheme
(
    void        *mma8451q_handle,
    uint8_t      power_scheme
)
```

Function Description:

This function set the power scheme of SLEEP mode. For more information about power scheme, please refer to **mma8451q_set_power_scheme()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *power_scheme[IN]*: Sleep mode power scheme to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with auto sleep power scheme macro in mma8451q_basic.h.

13.4.12.1.14 mma8451q_set_wake_up_bypass

Function Name:

```
bool mma8451q_set_wake_up_bypass
(
    void       *mma8451q_handle,
    uint8_t     wake_up_bypass
)
```

Function Description:

This function configures which embedded function is bypassed in SLEEP mode. The function which is not bypassed in SLEEP mode can wake up the sensor from SLEEP mode.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *wake_up_bypass[IN]*: Wake up bypass event to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with auto sleep wake up source macro in mma8451q_basic.h.

13.4.12.1.15 mma8451q_set_aslp_count

Function Name:

```
bool mma8451q_set_aslp_count
(
void       *mma8451q_handle,
uint8_t     aslp_count
)
```

Function Description:

This function set the countdown value of the minimum time period of inactivity required to switch the sensor from WAKE mode to SLEEP mode. How to set this value is specified in mma8451q datasheet.

Parameters:

- **mma8451q_handle**[IN]: MMA8451Q device instance handler.

- **aslp_count**[IN]: Auto sleep count to be set.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when the device is in "**STANDBY**" mode.
- For more information about how to choose **aslp_count** value, please refer to **Table. ASLP_COUNT Relationship with ODR** in mma8451q datasheet.

13.4.12.1.16 mma8451q_set_aslp_state

Function Name:
```
bool mma8451q_set_aslp_state
(
    void       *mma8451q_handle,
    uint8_t     aslp_enabled
)
```
Function Description:
This function set the enable/disable state of Auto Sleep feature. Enable this feature can reduce power dissipation.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *aslp_enabled[IN]*: Auto sleep function enable state.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with auto-sleep state macro in mma8451q_basic.h.

13.4.12.1.17 mma8451q_set_low_noise_state

Function Name:
```
bool mma8451q_set_low_noise_state
(
void      *mma8451q_handle,
uint8_t    lnoise_enabled
)
```
Function Description:

This function set the enable/disable state of the low noise feature. In Low Noise mode, the maximum signal that can be measured is ±4g. This feature should be set according to application requirement.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *lnoise_enabled[IN]*: Low noise enable state.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with low noise state macro in mma8451q_basic.h.
- Any thresholds set above 4g will not be reached, if low noise feature enabled.

13.4.12.1.18 mma8451q_set_hpf_cutoff

Function Name:
```
bool mma8451q_set_hpf_cutoff
(
    void      *mma8451q_handle,
    uint8_t    hpf_cutoff
)
```
Function Description:
This function set the cutoff frequency of mma8451q build-in high pass filter. This value is closely related to current output data rate and power scheme. How to set this value is specified in mma8451q datasheet.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **hpf_cutoff**[IN]: High pass filter cutoff configuration.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when the device is in "**STANDBY**" mode.
- For how to choose **hpf_cutoff** value, see **Table. High-Pass Filter Cutoff Options** in mma8451q data sheet.

13.4.12.1.19 mma8451q_set_hpf_state

Function Name:
```
bool mma8451q_set_hpf_state
(
    void      *mma8451q_handle,
    uint8_t    hpf_enabled
)
```
Function Description:
This function set the enable/disable state of mma8451q build-in high pass filter. The output data go through the high-pass filter is eliminated the offset (DC) and low frequencies (well below the cutoff). For details of implementation on the high-pass filter, please refer to Freescale application note, AN4071.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *hpf_enabled[IN]*: High pass filter enable state.

Return Value:
- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be used with high pass filter state macro in mma8451q_basic.h.

13.4.12.2 Basic I/O Functions

13.4.12.2.1 mma8451q_write_reg

Function Name:

```
bool mma8451q_write_reg
(
    void         *mma8451q_handle,
    uint8_t       addr,
    uint8_t      *buffer,
    uint16_t      n
)
```

Function Description:

MMA8451Q basic multi-byte write function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *addr[IN]*: MMA8451Q register address.
- *buffer[IN]*: Buffer for write function.
- *n[IN]*: Number of bytes to be sent.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.2.2 mma8451q_read_reg

Function Name:

```
bool mma8451q_read_reg
(
    void        *mma8451q_handle,
    uint8_t      addr,
    uint8_t     *buffer,
    uint16_t     n
)
```

Function Description:

MMA8451Q basic multi-byte read function.

Parameters:

- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **addr**[IN]: MMA8451Q register address.
- **buffer**[OUT]: Buffer for read function.
- **n**[IN]: Number of bytes to be read.

Return Value:
- **TRUE** if successful.

Note:
None.

13.4.12.2.3 mma8451q_write_single_reg

Function Name:
```
bool mma8451q_write_single_reg
(
    void        *mma8451q_handle,
    uint8_t      addr,
    uint8_t      data
)
```
Function Description:
MMA8451Q basic single byte write function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *addr[IN]*: MMA8451Q register address.
- *data[IN]*: Data to be write into MMA8451Q.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.2.4 mma8451q_read_single_reg

Function Name:
```
bool mma8451q_read_single_reg
(
    void        *mma8451q_handle,
    uint8_t      addr,
    uint8_t     *buffer
)
```
Function Description:
MMA8451Q basic single byte read function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *addr[IN]*: MMA8451Q register address.
- *buffer[OUT]*: Buffer for storing single register value.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.2.5 mma8451q_modify_bitField

Function Name:

```
bool mma8451q_modify_bitField
(
    void         *mma8451q_handle,
    uint8_t       reg_address,
    uint8_t       bit_field_mask,
    uint8_t       bit_field_value
)
```

Function Description:

This function is used to modify the bit field that indicated by **bit_field_mas**k with **bit_field_value.**

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *reg_address[IN]*: Address of the register to be modified.
- *bit_field_mask[IN]*: Bit field mask.
- *bit_field_value[IN]*: Bit field value to be written into register.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.2.6 mma8451q_get_bitField

Function Name:

```
bool mma8451q_get_bitField
(
    void        *mma8451q_handle,
    uint8_t      reg_address,
    uint8_t      bit_field_mask,
    uint8_t     *buffer
)
```

Function Description:

This function is used to get the content from bit field that indicated by **bit_field_mas**k.

Parameters:

- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **reg_address**[IN]: Address of the register to be read.
- **bit_field_mask**[IN]: Bit field mask.
- **buffer**[OUT]: Buffer to store register value.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.3 Data Acquisition Functions

13.4.12.3.1 mma8451q_get_acc_data

Function Name:

```
bool mma8451q_get_acc_data
(
    void        *mma8451q_handle,
    int16_t     *data_x,
    int16_t     *data_y,
    int16_t     *data_z
)
```

Function Description:

This function is used to get acceleration in 3 axes and store them into given buffer. The output data will be 14-bit width, if **BUREST READ MODE** is disabled. The output data will be 8-bit width (the MSB 8-bit data of **NORAMAL MODE**), if **BUREST READ MODE** is enabled. For burst read mode selection method, please refer to **mma8451q_set_burst_read_mode()** function description. The sensitivity of the acceleration is listed in mma8451q data sheet **Table. Mechanical Characteristics**.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *data_x[OUT]*: Buffer to store acceleration data in x axis.
- *data_y[OUT]*: Buffer to store acceleration data in y axis.
- *data_z[OUT]*: Buffer to store acceleration data in z axis.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.3.2 mma8451q_get_acc_from_fifo

Function Name:

```
bool mma8451q_get_acc_from_fifo
(
    void        *mma8451q_handle,
    uint8_t     *buffer,
    uint8_t      n
)
```

Function Description:

This function is used to read n samples (not n bytes) of acceleration data per axis from the mma8451q build-in FIFO. The FIFO depth is 32 samples per axis. The order of data read from the FIFO is described in mma8451q data sheet.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store acc data.

- *n[IN]*: Number of FIFO count to be read from FIFO.

Return Value:
- **TRUE** if successful.

Note:
- Read n FIFO count will read n measure data on each axis.
- The function will return FALSE, if the number of samples
- expected is greater than the actual one.

13.4.12.4 FIFO Data Buffer Configuration Functions

13.4.12.4.1 mma8451q_set_fifo_watermark

Function Name:
```
bool mma8451q_set_fifo_watermark
(
    void       *mma8451q_handle,
    uint8_t     watermark
)
```
Function Description:
This function is used to set the watermark of mma8451q build-in FIFO. A FIFO watermark event flag is raised when FIFO sample count ⩾ watermark. Setting the watermark to 0 will disable the FIFO watermark event flag generation.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *watermark[IN]*: FIFO watermark to be set.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.4.2 mma8451q_get_fifo_watermark

Function Name:
```
bool mma8451q_get_fifo_watermark
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```
Function Description:
This function is used to get the current FIFO watermark value. For more information about FIFO watermark, please refer to **mma8451q_set_fifo_watermark()** function description.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **buffer**[OUT]: Buffer to store FIFO watermark.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.4.3 mma8451q_set_fifo_mode

Function Name:
```
bool mma8451q_set_fifo_mode
(
    void         *mma8451q_handle,
    uint8_t       fifo_mode
)
```
Function Description:
This function is used to set the FIFO working mode. The mma8451q build-in FIFO can work in 4 modes: DISABLE mode, CIRCULAR mode, FULL-FILL mode and TRIGGER mode. The description of each FIFO mode is list in mma8451q datasheet. User should set FIFO mode according to application requirement. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **fifo_mode**[IN]: FIFO mode to be set.

Return Value:
- **TRUE** if successful.

Note:

Should be used with FIFO status macro in mma8451q_basic.h.

13.4.12.4.4 mma8451q_get_fifo_mode

Function Name:
```
bool mma8451q_get_fifo_mode
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```
Function Description:
This function is used to get the current fifo mode. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **buffer**[OUT]: Buffer to store FIFO mode.

Return Value:
- **TRUE** if successful.

Note:

Should be used with FIFO status macro in mma8451q_basic.h.

13.4.12.4.5 mma8451q_set_fifo_trigger_source

Function Name:

```
bool mma8451q_set_fifo_trigger_source
(
    void        *mma8451q_handle,
    uint8_t      trigger_source
)
```

Function Description:

This function is used to set which function may trigger the FIFO to its interrupt. The bits are rising edge sensitive, and are set by a low to high state change and reset by reading the appropriate source register. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *trigger_source[IN]*: FIFO trigger source to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with FIFO trigger configuration macro in mma8451q_basic.h.

13.4.12.4.6 mma8451q_get_fifo_trigger_source

Function Name:

```
bool mma8451q_get_fifo_trigger_source
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```

Function Description:

This function is used to get the current fifo trigger souce value.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store FIFO trigger source value.

Return Value:

- **TRUE** if successful.

Note:

Should be used with FIFO trigger configuration macro in mma8451q_basic.h. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.

13.4.12.4.7 mma8451q_set_fifo_gate

Function Name:

```
bool mma8451q_set_fifo_gate
(
    void          *mma8451q_handle,
    uint8_t        fifo_gate
)
```

Function Description:

This function is used to set the fifo gate enable/disable state. The FIFO input buffer is blocked when transitioning from WAKE to SLEEP mode or from SLEEP to WAKE mode until the FIFO is flushed, if this feature is enabled. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *fifo_gate[IN]*: FIFO gate to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with FIFO gate macro in mma8451q_basic.h.

13.4.12.4.8 mma8451q_get_fifo_gate

Function Name:

```
bool mma8451q_get_fifo_gate
(
    void       *mma8451q_handle,
    uint8_t    *buffer
)
```

Function Description:

This function is used to get current fifo gate enable/disable state. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store FIFO gate value.

Return Value:

- **TRUE** if successful.

Note:

- Should be used with FIFO gate macro in mma8451q_basic.h.

13.4.12.4.9 mma8451q_get_fifo_status

Function Name:

```
bool mma8451q_get_fifo_status
```

```
(
void          *mma8451q_handle,
uint8_t       *buffer
)
```

Function Description:

This function is used to get the current fifo working status including fifo overflow flag, fifo watermark flag, and current fifo count. For more information about how to use the FIFO, please refer to Freescale application note, AN4073.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store current FIFO status.

Return Value:

- **TRUE** if successful.

Note:

Should be used with FIFO status macro in mma8451q_basic.h.

13.4.12.4.10 mma8451q_get_fifo_count

Function Name:

```
bool mma8451q_get_fifo_count
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get the current number of samples stored in the FIFO. For more information about how to use the FIFO, see the Freescale application note, AN4073.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store fifo count.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.5 Interrupt Configuration Functions

13.4.12.5.1 mma8451q_set_int_polarity

Function Name:

```
bool mma8451q_set_int_polarity
(
    void         *mma8451q_handle,
    uint8_t       polarity
```

```
)
```

Function Description:

This function is used to set the interrupt output polarity on INT1 and INT2 pin of mma8451q. This value should be set according to hardware connection.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *polarity[IN]*: MMA8451Q interrupt polarity to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with interrupt polarity macro in mma8451q_basic.h.

13.4.12.5.2 mma8451q_get_int_polarity

Function Name:

```
bool mma8451q_get_int_polarity
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get the current interrupt output polarity.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store interrupt polarity.

Return Value:

- **TRUE** if successful.

Note:

Should be used with interrupt polarity macro in mma8451q_basic.h.

13.4.12.5.3 mma8451q_set_int_output_mode

Function Name:

```
bool mma8451q_set_int_output_mode
(
    void         *mma8451q_handle,
    uint8_t       output_mode
)
```

Function Description:

This function is used to set the interrupt output mode on INT1and INT2 pin of the sensor. The interrupt output mode can be set to Push-Pull or Open Drain mode. This value should be set according to hardware connection.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *output_mode[IN]*: MMA8451Q interrupt output mode to be set.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with interrupt output mode macro in mma8451q_basic.h.

13.4.12.5.4 mma8451q_get_int_output_mode

Function Name:
```
bool mma8451q_get_int_output_mode
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```
Function Description:
This function is used to get current interrupt output mode on INT1 and INT2 pin of mma8451q.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store interrupt output mode.

Return Value:
- **TRUE** if successful.

Note:

Should be used with interrupt output mode macro in mma8451q_basic.h.

13.4.12.5.5 mma8451q_set_int_pin_route

Function Name:
```
bool mma8451q_set_int_pin_route
(
    void          *mma8451q_handle,
    uint8_t        pin_route
)
```
Function Description:
This function is used to set the pin route of specified interrupt source. There are 7 interrupt sources that can be route to INT1 or INT2 pin. This value should be set according to application requirement.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **pin_route**[IN]: MMA8451Q interrupt pin route to be set.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.

- Should be used with interrupt pin route macro in mma8451q_basic.h.

13.4.12.5.6 mma8451q_get_int_pin_route

Function Name:

```
bool mma8451q_get_int_pin_route
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get the current interrupt pin route configuration.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store interrupt pin route.

Return Value:

- **TRUE** if successful.

Note:

Should be used with interrupt pin route macro in mma8451q_basic.h.

13.4.12.5.7 mma8451q_set_int_state

Function Name:

```
bool mma8451q_set_int_state
(
    void         *mma8451q_handle,
    uint8_t       int_enabled
)
```

Function Description:

This function is used to set the enable/disable state of specified interrupt source.
There are 7 interrupt sources that can be enabled / disabled independently.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *int_enabled[IN]*: MMA8451Q interrupt state to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with interrupt enable/disable macro in mma8451q_basic.h.

13.4.12.5.8 mma8451q_get_int_state

Function Name:

```
bool mma8451q_get_int_state
(
```

```
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get the current interrupt enable/disable state.

Parameters:

- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **buffer**[OUT]: Buffer to store interrupt state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with interrupt enable/disable macro in mma8451q_basic.h.

13.4.12.5.9 mma8451q_get_int_source

Function Name:

```
bool mma8451q_get_int_source
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to look up the interrupt source when interrupt is detected.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store interrupt source flag.

Return Value:

- **TRUE** if successful.

Note:

- Should be used with interrupt source macro in mma8451q_basic.h.
- For more information about how to clear interrupt flag of the sensor, please refer to mma8451q datasheet or take the example as a reference.

13.4.12.6 Motion and Freefall Detection Functions

13.4.12.6.1 mma8451q_set_ff_mt_db_cnt_mode

Function Name:

```
bool mma8451q_set_ff_mt_db_cnt_mode
(
    void          *mma8451q_handle,
    uint8_t        cnt_mode
)
```

Function Description:

This function is used to set freefall / motion detection debounce counter mode. The debounce counter is used to filter out irregular spurious events which might impede the detection of inertial events.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *cnt_mode[IN]*: Free Fall/Motion function debounce counter mode configuration.

Return Value:

- **TRUE** if successful.

Note:

Should be used with debounce counter mode macro in mma8451q_ff_mt.h.

13.4.12.6.2 mma8451q_get_ff_mt_db_cnt_mode

Function Name:

```
bool mma8451q_get_ff_mt_db_cnt_mode
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get current debounce counter mode of freefall / motion detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store debounce counter configuration.

Return Value:

- **TRUE** if successful.

Note:

Should be used with debounce counter mode macro in mma8451q_ff_mt.h.

13.4.12.6.3 mma8451q_set_ff_mt_db_cnt

Function Name:

```
bool mma8451q_set_ff_mt_db_cnt
(
    void          *mma8451q_handle,
    uint8_t        cnt_value
)
```

Function Description:

This function is used to set the debounce counter value of freefall / motion detection function. This value should be set according to application requirement. For more information about how to set this value, please refer to Freescale application note, AN4070.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *cnt_value[IN]*: Free Fall/Motion function debounce counter value to be set.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.6.4 mma8451q_get_ff_mt_db_cnt

Function Name:
```
bool mma8451q_get_ff_mt_db_cnt
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```
Function Description:
This function is used to get the current debounce counter value of freefall / motion detection function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store Free Fall/Motion function debounce counter value.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.6.5 mma8451q_set_ff_mt_threshold

Function Name:
```
bool mma8451q_set_ff_mt_threshold
(
    void         *mma8451q_handle,
    uint8_t       threshold
)
```
Function Description:
This function is used to set the freefall / motion detection threshold value. When acceleration measured exceeds the threshold value and freefall / motion event will be generated and interrupt will be generated if freefall / motion interrupt is enabled. For more information about how to set this value, please refer to Freescale application note, AN4070.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *threshold[IN]*: Free Fall/Motion function threshold value.

Return Value:
- **TRUE** if successful.

Note:

The threshold value cannot exceed 0x7F.

13.4.12.6.6 mma8451q_get_ff_mt_threshold

Function Name:

```
bool mma8451q_get_ff_mt_threshold
(
    void       *mma8451q_handle,
    uint8_t    *buffer
)
```

Function Description:

This function is used to get the current freefall / motion detection threshold value.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store Free Fall/Motion function threshold.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.6.7 mma8451q_set_ff_mt_event_latch_state

Function Name:

```
bool mma8451q_set_ff_mt_event_latch_state
(
    void            *mma8451q_handle,
    uint8_t          latch_enable
)
```

Function Description:

This function is used to set the event latch state of freefall / motion detection function. Enable this feature will latch event flag in FF_MT_SRC register; otherwise FF_MT_SRC will indicate the real-time status of the event.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *latch_enable[IN]*: Free Fall/Motion function event latch state to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with event latch enable macro in mma8451q_ff_mt.h.

13.4.12.6.8 mma8451q_get_ff_mt_event_latch_state

Function Name:

```
bool mma8451q_get_ff_mt_event_latch_state
(
    void          *mma8451q_handle,
```

```
    uint8_t       *buffer
)
```

Function Description:

This function is used to get current event latch state configuration of freefall / motion detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store Free Fall/Motion function event latch state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with event latch enable macro in mma8451q_ff_mt.h.

13.4.12.6.9 mma8451q_set_ff_mt_selection

Function Name:

```
bool mma8451q_set_ff_mt_selection
(
    void         *mma8451q_handle,
    uint8_t       selection
)
```

Function Description:

This function is used to selection the working mode of freefall / motion detection function. The freefall / motion detection module can work in freefall detection mode and motion detection mode.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *selection[IN]*: Free Fall/Motion function selection.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with freefall & motion detection selection macro in mma8451q_ff_mt.h.

13.4.12.6.10 mma8451q_get_ff_mt_selection

Function Name:

```
bool mma8451q_get_ff_mt_selection
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get the current freefall / motion detection function working mode.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.

- *buffer[OUT]*: Buffer to store Free Fall/Motion selection.

Return Value:
- **TRUE** if successful.

Note:

Should be used with freefall & motion detection selection macro in mma8451q_ff_mt.h.

13.4.12.6.11 mma8451q_set_ff_mt_state

Function Name:
```
bool mma8451q_set_ff_mt_state
(
    void          *mma8451q_handle,
    uint8_t        enable_state
)
```
Function Description:
This function is used to set the enable/disable state of freefall / motion detection function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *enable_state[IN]*: Free Fall/Motion function enable state to be set.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with freefall/motion detection enable macro in mma8451q_ff_mt.h.

13.4.12.6.12 mma8451q_get_ff_mt_state

Function Name:
```
bool mma8451q_get_ff_mt_state
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```
Function Description:
This function is used to get the current enable / disable state of freefall / motion detection function.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **buffer**[OUT]: Buffer to store Free Fall/Motion function enable state.

Return Value:
- **TRUE** if successful.

Note:

Should be used with freefall/motion detection enable macro in mma8451q_ff_mt.h.

13.4.12.6.13 mma8451q_get_ff_mt_status

Function Name:

```
bool mma8451q_get_ff_mt_status
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get current working status of freefall / motion detection function including event active flag, motion flags and motion polarity on each axis.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store current Free Fall/Motion function status.

Return Value:

- **TRUE** if successful.

Note:

Should be used with freefall and motion status macro in mma8451q_ff_mt.h.

13.4.12.7 Portrait/Landscape detection Functions

13.4.12.7.1 mma8451q_set_lapo_db_cnt_mode

Function Name:

```
bool mma8451q_set_lapo_db_cnt_mode
(
    void          *mma8451q_handle,
    uint8_t        cnt_mode
)
```

Function Description:

This function is used to set portrait/landscape detection debounce counter mode. The debounce counter is used to filter out irregular spurious events which might impede the detection of inertial events.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *cnt_mode[IN]*: Lapo function debounce counter mode configuration.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with debounce counter mode macro in mma8451q_lapo.h.

13.4.12.7.2 mma8451q_get_lapo_db_cnt_mode

```
bool mma8451q_get_lapo_db_cnt_mode
```

```
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get current debounce counter mode of portrait/landscape detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store debounce counter mode configuration.

Return Value:

- **TRUE** if successful.

Note:

Should be used with debounce counter mode macro in mma8451q_lapo.h.

13.4.12.7.3 mma8451q_set_lapo_db_cnt

```
bool mma8451q_set_lapo_db_cnt
(
    void         *mma8451q_handle,
    uint8_t       cnt_value
)
```

Function Description:

This function is used to set the debounce counter value of portrait/landscape detection function. This value should be set according to application requirement. For more information about how to set this value, please refer to Freescale application note, AN4068.

Parameters:

- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **cnt_value**[IN]: Lapo function debounce counter value to be set.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.7.4 mma8451q_get_lapo_db_cnt

```
bool mma8451q_get_lapo_db_cnt
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get the current debounce counter value of portrait/landscape detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.

- *buffer[OUT]*: Buffer to store debounce counter value.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.7.5 mma8451q_set_back_front_threshold

```
bool mma8451q_set_back_front_threshold
(
    void         *mma8451q_handle,
    uint8_t       threshold
)
```

Function Description:
This function is used to set the back front threshold of portrait/landscape detection function. This value is for back / front detection feature of portrait/landscape detection function. This value should be set according to application requirement. For more information about how to set this value, please refer to Freescale application note, AN4068.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *threshold[IN]*: Threshold of back/front trip angle.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with back/front trip angle threshold macro in mma8451q_lapo.h.

13.4.12.7.6 mma8451q_get_back_front_threshold

```
bool mma8451q_get_back_front_threshold
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:
This function is used to get the current back / front detection threshold of portrait/landscape detection function.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store back front threshold.

Return Value:
- **TRUE** if successful.

Note:

Should be used with back/front trip angle threshold macro in mma8451q_lapo.h.

13.4.12.7.7 mma8451q_set_lapo_threshold

```
bool mma8451q_set_lapo_threshold
(
    void         *mma8451q_handle,
    uint8_t       threshold
)
```

Function Description:
This function is used to set the portrait / landscape detection threshold. This value should be set according to application requirement. For more information about how to set this value, please refer to Freescale application note, AN4068.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *threshold[IN]*: Threshold of Portrait/Landscape trip angle.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with portrait/landscape threshold macro in mma8451q_lapo.h.

13.4.12.7.8 mma8451q_get_lapo_threshold

```
bool mma8451q_get_lapo_threshold
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:
This function is used to get current portrait / landscape detection threshold value.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store threshold of Portrait/Landscape trip angle.

Return Value:

- **TRUE** if successful.

Note:

Should be used with portrait/landscape threshold macro in mma8451q_lapo.h.

13.4.12.7.9 mma8451q_set_z_lock_threshold

```
bool mma8451q_set_z_lock_threshold
(
    void         *mma8451q_handle,
    uint8_t  threshold
)
```

Function Description:
This function is used to set the z-lock threshold of portrait / landscape detection function. For more information about z-lock meaning and how to set this value, please refer to Freescale application note, AN4068.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *threshold[IN]*: Threshold of z-lock angle.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with z-lock angle threshold macro in mma8451q_lapo.h.

13.4.12.7.10 mma8451q_get_z_lock_threshold

```
bool mma8451q_get_z_lock_threshold
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:
This function is used to get current z-lock threshold value of portrait / landscape detection function.
Parameters:
- *mReturn Value:ma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store threshold of z-lock angle.

Return Value:
- **TRUE** if successful.

Note:

Should be used with z-lock angle threshold macro in mma8451q_lapo.h.

13.4.12.7.11 mma8451q_set_lapo_trip_hys

```
bool mma8451q_set_lapo_trip_hys
(
    void          *mma8451q_handle,
    uint8_t        hysteresis
)
```

Function Description:
This function is used to set the portrait / landscape switch hysteresis angle. This value is used to reduce misinformation of portrait / landscape switch. For more information about how to set this value, please refer to Freescale application note, AN4068.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *hysteresis[IN]*: Hysteresis of Portrait/Landscape trip.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with hysteresis of portrait/landscape trip macro in mma8451q_lapo.h.

13.4.12.7.12 mma8451q_get_lapo_trip_hys

bool mma8451q_get_lapo_trip_hys
```
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```
Function Description:
This function is used to get the curret hysteresis of portrait / landscape switch.
Parameters:
- **mma8451q_handle**[IN]: MMA8451Q device instance handler.
- **buffer**[OUT]: Buffer to store hysteresis of Portrait/Landscape trip.

Return Value:
- **TRUE** if successful.

Note:

Should be used with hysteresis of portrait/landscape trip macro in mma8451q_lapo.h.

13.4.12.7.13 mma8451q_set_lapo_state

```
bool mma8451q_set_lapo_state
(
    void           *mma8451q_handle,
    uint8_t         lapo_enabled
)
```
Function Description:
This function is used to set the enable / disable state of portrait/landscape detection function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *lapo_enabled[IN]*: Portrait/Landscape function enable configuration.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with portrait/landscape function enable macro in mma8451q_lapo.h.

13.4.12.7.14 mma8451q_get_lapo_state

```
bool mma8451q_get_lapo_state
(
```

```
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:
This function is used to get current enable / disable state of portrait/landscape detection function.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store Portrait/Landscape function enable configuration.

Return Value:

- **TRUE** if successful.

Note:

Should be used with portrait/landscape function enable macro in mma8451q_lapo.h.

13.4.12.7.15 mma8451q_get_lapo_status

```
bool mma8451q_get_lapo_status
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```

Function Description:
This function is used to get current portrait/landscape detection function working status including landscape/portrait status change flag, z-tilt angle lockout status, landscape / portrait orientation status and back or front orientation status.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store Portrait/Landscape function status.

Return Value:

- **TRUE** if successful.

Note:

- Should be used with portrait/landscape status macro in mma8451q_lapo.h.

13.4.12.8 Pulse Detection Functions

13.4.12.8.1 mma8451q_set_double_pulse_abort

```
bool mma8451q_set_double_pulse_abort
(
    void            *mma8451q_handle,
    uint8_t          abort_sel
)
```

Function Description:
This function is used to set the enable / disable state of portrait double pulse abort feature of pulse detection function. Enable this feature suspends the double tap detection if the start of a pulse is detected during the time period specified by the pulse latency value and the pulse ends before the end of the time

period specified by the pulse latency value. For more information about how to set this value, please refer to Freescale application note, AN4072.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *abort_sel[IN]*: Pulse detect function double pulse abort selection.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with double pulse abort macro in mma8451q_pulse.h.

13.4.12.8.2 mma8451q_get_double_pulse_abort

```
bool mma8451q_get_double_pulse_abort
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```

Function Description:

This function is used to get current double pulse abort feature of pulse detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store double pulse abort selection.

Return Value:

- **TRUE** if successful.

Note:

Should be used with double pulse abort macro in mma8451q_pulse.h.

13.4.12.8.3 mma8451q_set_pulse_threshold

```
bool mma8451q_set_pulse_threshold
(
    void           *mma8451q_handle,
    uint8_t         threshold,
    uint8_t         axis
)
```

Function Description:

This function is used to set threshold for pulse detection in specified axis. When acceleration measured exceeds the threshold value and pulse detection event will be generated and interrupt will be generated if pulse detection interrupt is enabled. For more information about how to set this value, please refer to Freescale application note, AN4072.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *threshold[IN]*: Pulse detect function threshold in single axis.

- *axis[IN]*: Axis selection with mask.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with axis define in mma8451q_pulse.h.
- The threshold value can't exceed 0x7F.

13.4.12.8.4 mma8451q_get_pulse_threshold

```
bool mma8451q_get_pulse_threshold
(
    void             *mma8451q_handle,
    uint8_t          *buffer,
    uint8_t           axis
)
```

Function Description:
This function is used to get current pulse detect threshold in specified axis.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect function threshold in single axis.
- *axis[IN]*: Axis selection with mask.

Return Value:
- **TRUE** if successful.

Note:

Should be used with axis define in mma8451q_pulse.h.

13.4.12.8.5 mma8451q_set_pulse_time_limit

```
bool mma8451q_set_pulse_time_limit
(
    void         *mma8451q_handle,
    uint8_t       time_limit
)
```

Function Description:
This function is used to set pulse time limit. The pulse time limit value is closely related to output data rate and power scheme. For more information about how to set this value, please refer to Freescale application note, AN4072.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *time_limit[IN]*: Pulse detect function time limit.

Return Value:
- **TRUE** if successful.

Note:

This function can only be used when device in "**STANDBY**" mode.

13.4.12.8.6 mma8451q_get_pulse_time_limit

```
bool mma8451q_get_pulse_time_limit
(
      void            *mma8451q_handle,
      uint8_t         *buffer
)
```

Function Description:
This function is used to get current pulse time limit value.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect function time limit.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.8.7 mma8451q_set_pulse_latency

```
bool mma8451q_set_pulse_latency
(
    void          *mma8451q_handle,
    uint8_t        latency
)
```

Function Description:
This function is used to set pulse latency. The pulse latency value is closely related to output data rate and power scheme. For more information about how to set this value, please refer to Freescale application note, AN4072.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *latency[IN]*: Pulse detect function latency.

Return Value:

- **TRUE** if successful.

Note:

This function can only be used when device in "**STANDBY**" mode.

13.4.12.8.8 mma8451q_get_pulse_latency

```
bool mma8451q_get_pulse_latency
(
    void            *mma8451q_handle,
    uint8_t         *buffer
)
```

Function Description:
This function is used to get current pulse latency value.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect function latency.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.8.9 mma8451q_set_pulse_time_window

```
bool mma8451q_set_pulse_time_window
(
    void         *mma8451q_handle,
    uint8_t       time_window
)
```

Function Description:
This function is used to set pulse time window. The pulse time window value is closely related to output data rate and power scheme. For more information about how to set this value, please refer to Freescale application note, AN4072.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *time_window[IN]*: Pulse detect function time window.

Return Value:
- **TRUE** if successful.

Note:

This function can only be used when device in "**STANDBY**" mode.

13.4.12.8.10 mma8451q_get_pulse_time_window

```
bool mma8451q_get_pulse_time_window
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:
This function is used to get current pulse time window value.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect function time window.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.8.11 mma8451q_set_pulse_event_latch_state

```
bool mma8451q_set_pulse_event_latch_state
(
    void           *mma8451q_handle,
    uint8_t         latch_state
)
```

Function Description:

This function is used to the event latch state of pulse detection function. Enable this feature will latch event flag in PULSE_SRC register; otherwise PULSE_SRC will indicate the real-time status of the event.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *latch_state[IN]*: Pulse detect function event latch_state to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with event latch enable macro in mma8451q_pulse.h.

13.4.12.8.12 mma8451q_get_pulse_event_latch_state

```
bool mma8451q_get_pulse_event_latch_state
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get current pulse event latch state.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect function event latch_state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with event latch enable macro in mma8451q_pulse.h.

13.4.12.8.13 mma8451q_set_pulse_hpf_state

```
bool mma8451q_set_pulse_hpf_state
(
    void          *mma8451q_handle,
```

```
    uint8_t        bypass_state
)
```

Function Description:
This function is used to set the enable / disable state of high pass filter for pulse detection. For more information about how to set this value, please refer to Freescale application note, AN4072.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *bypass_state[IN]*: Pulse detect function high pass filter bypass state.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with high pass filter macro in mma8451q_pulse.h.

13.4.12.8.14 mma8451q_get_pulse_hpf_state

```
bool mma8451q_get_pulse_hpf_state
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:
This function is used to get enable / disable state of high pass filter for pulse detection function.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect high pass filter bypass state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with high pass filter macro in mma8451q_pulse.h.

13.4.12.8.15 mma8451q_set_pulse_lpf_state

```
bool mma8451q_set_pulse_lpf_state
(
    void           *mma8451q_handle,
    uint8_t         bypass_state
)
```

Function Description:
This function is used to set the enable / disable state of low pass filter for pulse detection. For more information about how to set this value, please refer to Freescale application note, AN4072.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *bypass_state[IN]*: Pulse detect function low pass filter bypass state.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with low pass filter macro in mma8451q_pulse.h.

13.4.12.8.16 mma8451q_get_pulse_lpf_state

```
bool mma8451q_get_pulse_lpf_state
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:
This function is used to get enable / disable state of low pass filter for pulse detection function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect low pass filter bypass state.

Return Value:
- **TRUE** if successful.

Note:

Should be used with low pass filter macro in mma8451q_pulse.h.

13.4.12.8.17 mma8451q_set_pulse_detect_state

```
bool mma8451q_set_pulse_detect_state
(
    void          *mma8451q_handle,
    uint8_t        tap_state
)
```

Function Description:
This function is used to set the enable / disable state pulse detection function.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *tap_state[IN]*: Pulse detect function enable state to be set.

Return Value:
- **TRUE** if successful.

Note:
- This function can only be used when device in "**STANDBY**" mode.
- Should be used with tap detection enable macro in mma8451q_pulse.h.

13.4.12.8.18 mma8451q_get_pulse_detect_state

```
bool mma8451q_get_pulse_detect_state
(
```

```
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:
This function is used to get the enable / disable state pulse detection function.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect function enable state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with tap detection enable macro in mma8451q_pulse.h.

13.4.12.8.19 mma8451q_get_pulse_detect_status

```
bool mma8451q_get_pulse_detect_status
(
    void            *mma8451q_handle,
    uint8_t         *buffer
)
```

Function Description:
This function is used to get current pulse detection function working status including: event active flag, single / double pulse detection flag on each axis.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store pulse detect status.

Return Value:

- **TRUE** if successful.

Note:

Should be used with pulse event status macro in mma8451q_pulse.h.

13.4.12.9 Transient Detection Functions

13.4.12.9.1 mma8451q_set_transient_db_cnt_mode

```
bool mma8451q_set_transient_db_cnt_mode
(
    void               *mma8451q_handle,
    uint8_t             cnt_mode
)
```

Function Description:
This function is used to set transient detection debounce counter mode. The
debounce counter is used to filter out irregular spurious events which might impede the detection of inertial events.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *cnt_mode[IN]*: Transient detect function debounce counter configuration.

Return Value:
- **TRUE** if successful.

Note:

Should be used with debounce counter mode macro in mma8451q_tran.h.

13.4.12.9.2 mma8451q_get_transient_db_cnt_mode

```
bool mma8451q_get_transient_db_cnt_mode
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:
This function is used to get current transient detection debounce counter mode
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store debounce counter enable config.

Return Value:
- **TRUE** if successful.

Note:

Should be used with debounce counter mode macro in mma8451q_tran.h.

13.4.12.9.3 mma8451q_set_transient_db_cnt

```
bool mma8451q_set_transient_db_cnt
(
    void          *mma8451q_handle,
    uint8_t        cnt_value
)
```

Function Description:
This function is used to set the debounce counter value of transient detection function. This value should be set according to application requirement. For more information about how to set this value, please refer to Freescale application note, AN4071.
Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *cnt_value[IN]*: Transient detect function debounce counter value.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.9.4 mma8451q_get_transient_db_cnt

```
bool mma8451q_get_transient_db_cnt
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get the current debounce counter value of transient detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store debounce counter value.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.9.5 mma8451q_set_transient_threshold

```
bool mma8451q_set_transient_threshold
(
    void          *mma8451q_handle,
    uint8_t        threshold
)
```

Function Description:

This function is used to set the transient detection threshold value. When acceleration measured exceeds the threshold value and transient event will be generated and interrupt will be generated if transient interrupt is enabled. For more information about how to set this value, please refer to Freescale application note, AN4071.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *threshold[IN]*: Transient detect function threshold.

Return Value:

- **TRUE** if successful.

Note:

The threshold value can’t exceed 0x7F.

13.4.12.9.6 mma8451q_get_transient_threshold

```
bool mma8451q_get_transient_threshold
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get the current transient detection threshold value.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store transient detect function threshold.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.9.7 mma8451q_set_transient_event_latch_state

```
bool mma8451q_set_transient_event_latch_state
(
    void            *mma8451q_handle,
    uint8_t          latch_state
)
```

Function Description:

This function is used to the event latch state of transient detection function.
Enable this feature will latch event flag in TRANSIENT_SRC register; otherwise TRANSIENT_SRC will indicate the real-time status of the event.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *latch_state[IN]*: Transient detect function event latch state to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with event latch enable macro in mma8451q_tran.h.

13.4.12.9.8 mma8451q_get_transient_event_latch_state

```
bool mma8451q_get_transient_event_latch_state
(
    void             *mma8451q_handle,
    uint8_t          *buffer
)
```

Function Description:

This function is used to get current event latch state configuration of transient detection function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store transient detect function event latch state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with event latch enable macro in mma8451q_tran.h.

13.4.12.9.9 mma8451q_set_transient_bypass_hpf

```
bool mma8451q_set_transient_bypass_hpf
(
    void           *mma8451q_handle,
    uint8_t         bypass_state
)
```

Function Description:

This function is used to set the enable / disable state of high pass filter bypass feature. The transient function will act the same as motion detection function, if this feature is enabled.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *bypass_state[IN]*: Transient detect function high pass filter bypass state to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with bypass high-pass filter macro in mma8451q_tran.h.

13.4.12.9.10 mma8451q_get_transient_bypass_hpf

```
bool mma8451q_get_transient_bypass_hpf
(
    void            *mma8451q_handle,
    uint8_t         *buffer
)
```

Function Description:

This function is used to get current enable / disable state of bypass high pass filter feature.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store transient detect high pass filter bypass state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with bypass high-pass filter macro in mma8451q_tran.h.

13.4.12.9.11 mma8451q_set_transient_state

```
bool mma8451q_set_transient_state
(
    void          *mma8451q_handle,
    uint8_t        enable_state
```

```
)
```

Function Description:
This function is used to set the enable / disable state of transient detection function.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *enable_state[IN]*: Transient detect function enable configuration.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when device in "**STANDBY**" mode.
- Should be used with transient detection enable macro in mma8451q_tran.h.

13.4.12.9.12 mma8451q_get_transient_state

```
bool mma8451q_get_transient_state
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:
This function is used to get current enable / disable state of transient detection function.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *mma8451q_handle[IN]*: Buffer to store transient detect function enable state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with transient detection enable macro in mma8451q_tran.h.

13.4.12.9.13 mma8451q_get_transient_status

```
bool mma8451q_get_transient_status
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:
This function is used to get current transient detection status including Event Active Flag, transient event flag and polarity of transient event flag on each axis.
Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store transient detect function status.

Return Value:

- **TRUE** if successful.

Note:

Should be used with transient status macro in mma8451q_tran.h.

13.4.12.10 Status Inquiry Functions

13.4.12.10.1 mma8451q_get_slave_address

```
bool mma8451q_get_slave_address
(
    void            *mma8451q_handle,
    uint8_t         *buffer
)
```

Function Description:

This function is used to store current mma8451q slave address configuration into given buffer. For more information about slave address, please refer to **mma8451q_set_slave_address()** and mma8451q initialization structure SLAVE_ADDRESS field.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store slave address.

Return Value:

- **TRUE** if successful.

Note:

Slave address should be set according to mma8451q **SA0** Pin connection. If **SA0** connect to logic 0, slave address will be 0x1C. If **SA0** connect to logic 1, slave address will be 0x1D. For more information, please refer to MMA8451Q datasheet.

13.4.12.10.2 mma8451q_get_dr_status

```
bool mma8451q_get_dr_status
(
    void            *mma8451q_handle,
    uint8_t         *buffer
)
```

Function Description:

This function is used to store current data ready status into given buffer. The data ready status can be one of the data ready status macro defined in mma8451q_basic.h or their combination.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store Data Ready Status Register value.

Return Value:

- **TRUE** if successful.

Note:

Should be used with data ready status macro in mma8451q_basic.h.

13.4.12.10.3 mma8451q_get_device_id

```
bool mma8451q_get_device_id
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get mma8451q Device ID Number and stores the number into given buffer. The Device ID Number read back should match MMA8451Q_DEVICE_ID.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store WHO_AM_I Register value.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.10.4 mma8451q_get_system_mode

```
bool mma8451q_get_system_mode
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get mma8451q system mode and stores it into given buffer. System mode should match one of the system mode macro defined in mma8451q_biasic.h

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store SYSMOD value.

Return Value:

- **TRUE** if successful.

Note:

Should be used with system mode macro in mma8451q_basic.h

13.4.12.10.5 mma8451q_get_output_data_rate

```
bool mma8451q_get_output_data_rate
(
void           *mma8451q_handle,
uint8_t        *buffer
)
```

Function Description:

This function is used to get current output data rate and stores it into given buffer. For more information about output data rate, please refer to **mma8451q_set_output_data_rate()** function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store output data rate.

Return Value:

- **TRUE** if successful.

Note:

Should be used with output data rate macro in mma8451q_basic.h

13.4.12.10.6 mma8451q_get_power_scheme

```
bool mma8451q_get_power_scheme
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get current active power and stores it into given buffer. For more information about output data rate, please refer to **mma8451q_set_power_scheme()** function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store power scheme.

Return Value:

- **TRUE** if successful.

Note:

Should be used with active power scheme macro in mma8451q_basic.h.

13.4.12.10.7 mma8451q_get_full_scale_range

```
bool mma8451q_get_full_scale_range
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get current full scale range and stores it into given buffer. For more information about output data rate, please refer to **mma8451q_set_full_scale_range()** function.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store full scale range.

Return Value:

- **TRUE** if successful.

Note:
- Should be used with full scale range macro in mma8451q_basic.h.

13.4.12.10.8 mma8451q_get_burst_read_mode

```
bool mma8451q_get_burst_read_mode
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:
This function is used to get current burst read mode and stores it into given buffer. For more information about burst read mode, please refer to **mma8451q_set_burst_read_mode()** function.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store burst read mode.

Return Value:
- **TRUE** if successful.

Note:

Should be used with burst read mode macro in mma8451q_basic.h.

13.4.12.10.9 mma8451q_get_user_offset

```
bool mma8451q_get_user_offset
(
    void           *mma8451q_handle,
    int8_t         *offset_x,
    int8_t         *offset_y,
    int8_t         *offset_z
)
```

Function Description:
This function gets current data output correction configuration and stores it into given buffer. For more information about data output correction configuration, please refer to **mma8451q_set_user_offset()** function description.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *offset_x[OUT]*: buffer to store User Offset Correction in x axis from OFF_X reg.
- *offset_y[OUT]*: buffer to store User Offset Correction in y axis from OFF_Y reg.
- *offset_z[OUT]*: buffer to store User Offset Correction in z axis from OFF_Z reg.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.10.10mma8451q_get_self_test_state

```
bool mma8451q_get_self_test_state
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get current self test state and stores it into given buffer. For more information about data output correction configuration, please refer to **mag3110_set_self_test_state()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store self test state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with self test state macro in mma8451q_basic.h.

13.4.12.10.11mma8451q_get_senor_reset_state

```
bool mma8451q_get_senor_reset_state
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get current reset progress and stores it into given buffer. For more information about reset progress, please refer to **mma8451q_reset_sensor()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store reset state.

Return Value:

- **TRUE** if successful.

Note:

- Should be used with sensor reset status macro in mma8451q_basic.h.

13.4.12.10.12mma8451q_get_operating_mode

```
bool mma8451q_get_operating_mode
(
    void           *mma8451q_handle,
    uint8_t        *buffer
)
```

Function Description:

This function is used to get current operating mode and stores it into given buffer. For more information about operating mode, please refer to **mma8451q_set_operating_mode()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store current operating mode.

Return Value:

- **TRUE** if successful.

Note:

Should be used with operating mode macro in mma8451q_basic.h.

13.4.12.10.13mma8451q_get_aslp_output_data_rate

```
bool mma8451q_get_aslp_output_data_rate
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get current auto sleep output data rate and stores it into given buffer. For more information about auto sleep output data rate, please refer to **mma8451q_set_aslp_output_data_rate()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store auto sleep output data rate.

Return Value:

- **TRUE** if successful.

Note:

Should be used with auto sleep output data rate macro in mma8451q_basic.h.

13.4.12.10.14mma8451q_get_aslp_power_scheme

```
bool mma8451q_get_aslp_power_scheme
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get current auto sleep power scheme and stores it into given buffer. For more information about auto sleep power scheme, please refer to **mma8451q_set_aslp_power_scheme()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store asleep power scheme.

Return Value:
- **TRUE** if successful.

Note:

Should be used with auto sleep power scheme macro in mma8451q_basic.h.

13.4.12.10.15mma8451q_get_wake_up_bypass

```
bool mma8451q_get_wake_up_bypass
(
    void            *mma8451q_handle,
    uint8_t         *buffer
)
```

Function Description:
This function is used to get current wake up bypass configuration and stores it into given buffer. For more information about wake up bypass configuration, please refer to **mma8451q_set_wake_up_bypass()** function description.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store wake up bypass event.

Return Value:
- **TRUE** if successful.

Note:

Should be used with auto sleep wake up source macro in mma8451q_basic.h.

13.4.12.10.16mma8451q_get_aslp_count

```
bool mma8451q_get_aslp_count
(
    void          *mma8451q_handle,
    uint8_t       *buffer
)
```

Function Description:
This function is used to get current auto sleep count value and stores it into given buffer. For more information about auto sleep count value, please refer to **mma8451q_set_aslp_count()** function description.

Parameters:
- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store auto sleep count.

Return Value:
- **TRUE** if successful.

Note:

None.

13.4.12.10.17mma8451q_get_aslp_state

```
bool mma8451q_get_aslp_state
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```

Function Description:

This function is used to get current auto sleep state and stores it into given buffer. For more information about auto sleep state, please refer to **mma8451q_set_aslp_state()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store auto sleep enable state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with auto sleep state macro in mma8451q_basic.h.

13.4.12.10.18mma8451q_get_low_noise_state

```
bool mma8451q_get_low_noise_state
(
    void         *mma8451q_handle,
    uint8_t      *buffer
)
```

Function Description:

This function is used to get current low noise state and stores it into given buffer. For more information about low noise state, please refer to **mma8451q_set_low_noise_state()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store low noise enable state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with low noise state macro in mma8451q_basic.h.

13.4.12.10.19mma8451q_get_hpf_cutoff

```
bool mma8451q_get_hpf_cutoff
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```

Function Description:

This function is used to get current high pass filter cutoff value and stores it into given buffer. For more information about high pass filter cutoff, please refer to **mma8451q_set_hpf_cutoff()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store high pass filter cutoff configuration.

Return Value:

- **TRUE** if successful.

Note:

None.

13.4.12.10.20mma8451q_get_hpf_state

```
bool mma8451q_get_hpf_state
(
    void        *mma8451q_handle,
    uint8_t     *buffer
)
```

Function Description:

This function is used to get current high pass filter state and stores it into given buffer. For more information about high pass filter state, please refer to **mma8451q_set_hpf_state()** function description.

Parameters:

- *mma8451q_handle[IN]*: MMA8451Q device instance handler.
- *buffer[OUT]*: Buffer to store high pass filter enable state.

Return Value:

- **TRUE** if successful.

Note:

Should be used with high pass filter state macro in mma8451q_basic.h.

13.5 MMA8451Q Driver Defines

13.5.1 Generic Function Macro

13.5.1.1 I2C Slave Address Macro

These macros are used with **mma8451q_set_slave_address()** and **mma8451q_get_slave_address()** function to set or identify current I2C slave address of mma8451q device.
MMA8451Q_DEFAULT_ADDRESS is equal to MMA8451Q_ADDRESS_SA0_LOW.

- MMA8451Q_ADDRESS_SA0_LOW
- MMA8451Q_ADDRESS_SA0_HIGH
- MMA8451Q_DEFAULT_ADDRESS

13.5.1.2 Device ID Number Macro

This macro is used with **mma8451q_get_device_id()** function to distinguish mma8451q device from other I2C slave device.

- MMA8451Q_DEVICE_ID

13.5.1.3 Data Ready Status Macro

These macros are used with the **mma8451q_get_dr_status()** function.

- MMA8451Q_DATA_READY_ZYXOW
  - Previous X, Y, or Z data was overwritten by new X, Y, or Z data before it was read.
- MMA8451Q_DATA_READY_ZOW
  - Previous Z-axis data was overwritten by new Z-axis data before it was read.
- MMA8451Q_DATA_READY_YOW
  - Previous Y-axis data was overwritten by new Y-axis data before it was read.
- MMA8451Q_DATA_READY_XOW
  - Previous X-axis data was overwritten by new X-axis data before it was read.
- MMA8451Q_DATA_READY_ZYXDR
  - X, Y, Z-axis new Data Ready.
- MMA8451Q_DATA_READY_ZDR
  - Z-axis new Data Available.
- MMA8451Q_DATA_READY_YDR
  - Y-axis new Data Available.
- MMA8451Q_DATA_READY_XDR
  - X-axis new Data Available.

13.5.1.4 System Mode Macro

These macros are used with the **mma8451q_get_system_mode()** function. The system mode should equals to one of the following macros.

- MMA8451Q_SYSMOD_STANDBY
  - STANDBY mode
- MMA8451Q_SYSMOD_WAKE
  - WAKE mode
- MMA8451Q_SYSMOD_SLEEP
  - SLEEP mode

13.5.1.5 Output Data Rate Macro

These macros are used with the **mma8451q_set_output_data_rate()** and **mma8451q_get_output_data_rate()** function. For more information about how to choose output data rate, please refer to mma8451q data sheet.

- MMA8451Q_OUTPUT_DATA_RATE_800HZ
- MMA8451Q_OUTPUT_DATA_RATE_400HZ
- MMA8451Q_OUTPUT_DATA_RATE_200HZ
- MMA8451Q_OUTPUT_DATA_RATE_100HZ
- MMA8451Q_OUTPUT_DATA_RATE_50HZ
- MMA8451Q_OUTPUT_DATA_RATE_12HZ5
- MMA8451Q_OUTPUT_DATA_RATE_6HZ25
- MMA8451Q_OUTPUT_DATA_RATE_1HZ56

13.5.1.6 Active Power Scheme Macro

These macros are used with the **mma8451q_set_power_scheme()** and **mma8451q_get_power_scheme()** function. For more information about how to choose active power scheme, please refer to mma8451q data sheet.

- MMA8451Q_ACTIVE_POWER_SCHEME_NORMAL
- MMA8451Q_ACTIVE_POWER_SCHEME_LOW_NOISE_LOW_POWER
- MMA8451Q_ACTIVE_POWER_SCHEME_HIGH_RESOLUTION
- MMA8451Q_ACTIVE_POWER_SCHEME_LOW_POWER

13.5.1.7 Full Scale Range Macro

These macros are used with the **mma8451q_set_full_scale_range()** and **mma8451q_get_full_scale_range()** functions. This value should be chosen according to application requirement.

- MMA8451Q_FULL_SCALE_RANGE_2G
- MMA8451Q_FULL_SCALE_RANGE_4G
- MMA8451Q_FULL_SCALE_RANGE_8G

13.5.1.8 Burst Read Mode Macro

These macros are used with the **mma8451q_set_burst_read_mode()** and **mma8451q_get_burst_read_mode()** functions. For more information about how to choose burst read mode, please refer to **mma8451q_set_burst_read_mode()** function description.

- MMA8451Q_BURST_READ_MODE_NORMAL
- MMA8451Q_BURST_READ_MODE_FAST

13.5.1.9 Self Test State Macro

These macros are used with the **mma8451q_set_self_teset_state()** and **mma8451q_get_self_test_state()** functions.

- MMA8451Q_SELF_TEST_DISABLE
- MMA8451Q_SELF_TEST_ENABLE

13.5.1.10 Sensor Reset Status Macro

This macro is used with **mma8451q_get_sensor_reset_state()** function to indicate current reset status.

- MMA8451Q_SENOR_RESET_STATUS

13.5.1.11 Operating Mode Macro

These macros are used with the **mma8451q_set_operating_mode()** and **mma8451q_get_operating_mode()** functions. For more information about how to choose operating mode, please refer to **mma8451q_set_operating_mode()** function description.

- MMA8451Q_OPERATING_MODE_STANDBY
- MMA8451Q_OPERATING_MODE_ACTIVE

13.5.2 FIFO Function Macro

13.5.2.1 FIFO Mode Macro

These macros are used with the **mma8451q_set_fifo_mode()** and **mma8451q_get_fifo_mode()** functions. For more information about how to choose this value, please refer to **mma8451q_set_fifo_mode()** function description.

- MMA8451Q_FIFO_MODE_DISABLE
- MMA8451Q_FIFO_MODE_CIRCULAR
- MMA8451Q_FIFO_MODE_FULL_FILL
- MMA8451Q_FIFO_MODE_TRIGGER

13.5.2.2 FIFO Trigger Configuration Macro

These macros are used with the **mma8451q_set_fifo_trigger_source()** and **mma8451q_get_fifo_trigger_source()** functions. For more information about how to choose this value, please refer to **mma8451q_set_fifo_trigger_source()** function description.

- MMA8451Q_TRIG_CFG_TRANS
- MMA8451Q_TRIG_CFG_LNDPRT
- MMA8451Q_TRIG_CFG_PULSE
- MMA8451Q_TRIG_CFG_FF_MT

13.5.2.3 FIFO Gate Macro

These macros are used with the **mma8451q_set_fifo_gate()** and **mma8451q_get_fifo_gate()** functions. For more information about how to choose this value, please refer to **mma8451q_set_fifo_gate()** function description.

- MMA8451Q_FIFO_GATE_DISABLE
- MMA8451Q_FIFO_GATE_ENABLE

13.5.2.4 FIFO Status Macro

These macros are used with the **mma8451q_get_fifo_status()** function to indicate current working status of mma8451q build-in FIFO.

- MMA8451Q_FIFO_EVENT_OVERFLOW
  - FIFO has overflowed.
- MMA8451Q_FIFO_EVENT_WATERMARK
  - FIFO watermark events detected.

13.5.3 Interrupt Function Macro

13.5.3.1 Interrupt Polarity Macro

These macros are used with **mma8451q_set_int_polarity()** and **mma8451q_get_int_polarity()** function. For more information about how to choose this value, please refer to **mma8451q_set_int_polarity()** function description.

- MMA8451Q_INT_POLARITY_ACTIVE_LOW
- MMA8451Q_INT_POLARITY_ACTIVE_HIGH

13.5.3.2 Interrupt Output Mode Macro

These macros are used with the **mma8451q_set_int_output_mode()** and **mma8451q_get_int_output_mode()** functions. For more information about how to choose this value, please refer to **mma8451q_set_int_output_mode()** function description.

- MMA8451Q_INT_MODE_PUSH_PULL
- MMA8451Q_INT_MODE_OPEN_DRAIN

13.5.3.3 Interrupt Pin Route Macro

These macros are used with the **mma8451q_set_int_pin_route()** and **mma8451q_get_int_pin_route()** functions. For more information about how to choose this value, please refer to **mma8451q_set_int_pin_route()** function description.

- MMA8451Q_ASLP_INT_ROUTE_TO_INT1
- MMA8451Q_ASLP_INT_ROUTE_TO_INT2
- MMA8451Q_FIFO_INT_ROUTE_TO_INT1
- MMA8451Q_FIFO_INT_ROUTE_TO_INT2
- MMA8451Q_TRANS_INT_ROUTE_TO_INT1
- MMA8451Q_TRANS_INT_ROUTE_TO_INT2
- MMA8451Q_LNDPRT_INT_ROUTE_TO_INT1
- MMA8451Q_LNDPRT_INT_ROUTE_TO_INT2
- MMA8451Q_PULSE_INT_ROUTE_TO_INT1
- MMA8451Q_PULSE_INT_ROUTE_TO_INT2

- MMA8451Q_FF_MT_INT_ROUTE_TO_INT1
- MMA8451Q_FF_MT_INT_ROUTE_TO_INT2
- MMA8451Q_DRDY_INT_ROUTE_TO_INT1
- MMA8451Q_DRDY_INT_ROUTE_TO_INT2

13.5.3.4 Interrupt Enable/Disable Macro

These macros are used with the **mma8451q_set_int_state()** and **mma8451q_get_int_state()** functions.

- MMA8451Q_ASLP_INT_ENABLE
- MMA8451Q_ASLP_INT_DISABLE
- MMA8451Q_FIFO_INT_ENABLE
- MMA8451Q_FIFO_INT_DISABLE
- MMA8451Q_TRANS_INT_ENABLE
- MMA8451Q_TRANS_INT_DISABLE
- MMA8451Q_LNDPRT_INT_ENABLE
- MMA8451Q_LNDPRT_INT_DISABLE
- MMA8451Q_PULSE_INT_ENABLE
- MMA8451Q_PULSE_INT_DISABLE
- MMA8451Q_FF_MT_INT_ENABLE
- MMA8451Q_FF_MT_INT_DISABLE
- MMA8451Q_DRDY_INT_ENABLE
- MMA8451Q_DRDY_INT_DISABLE

13.5.3.5 Interrupt Source Macro

These macros are used with the **mma8451q_get_int_source()** function to indicate interrupt trigger source.

- MMA8451Q_INT_SOURCE_ASLP
  - A WAKE to SLEEP or SLEEP to WAKE system mode transition has occurred.
- MMA8451Q_INT_SOURCE_FIFO
  - A FIFO interrupt event occurred.
- MMA8451Q_INT_SOURCE_TRANS
  - A transient event has occurred.
- MMA8451Q_INT_SOURCE_LNDPRT
  - A change in orientation status event has occurred.
- MMA8451Q_INT_SOURCE_PULSE
  - A single and/or double pulse event has occurred.
- MMA8451Q_INT_SOURCE_FF_MT
  - A freefall or motion event has occurred.

- MMA8451Q_INT_SOURCE_DRDY
  - A data ready event has occurred.

13.5.4 Auto Sleep and low Noise Function Macro

13.5.4.1 Auto Sleep Output Data Rate Macro

These macros are used with the **mma8451q_set_aslp_output_data_rate()** and **mma8451q_get_aslp_output_data_rate()** functions. For more information about how to choose this value, please refer to **mma8451q_set_aslp_output_data_rate()** function description.

- MMA8451Q_ASLP_OUTPUT_DATA_RATE_50HZ
- MMA8451Q_ASLP_OUTPUT_DATA_RATE_12HZ5
- MMA8451Q_ASLP_OUTPUT_DATA_RATE_6HZ25
- MMA8451Q_ASLP_OUTPUT_DATA_RATE_1HZ56

13.5.4.2 Auto Sleep Power Scheme Macro

These macros are used with the **mma8451q_set_aslp_power_scheme()** and **mma8451q_get_aslp_power_scheme()** functions. For more information about how to choose this value, please refer to **mma8451q_set_aslp_power_scheme()** function description.

- MMA8451Q_ASLP_POWER_SCHEME_NORMAL
- MMA8451Q_ASLP_POWER_SCHEME_LOW_NOISE_LOW_POWER
- MMA8451Q_ASLP_POWER_SCHEME_HIGH_RESOLUTION
- MMA8451Q_ASLP_POWER_SCHEME_LOW_POWER

13.5.4.3 Auto sleep Wake Up Source Macro

These macros are used with the **mma8451q_set_wake_up_bypass()** and **mma8451q_get_wake_up_bypass()** functions.

- MMA8451Q_ASLP_WAKE_TRANS_DISABLE
- MMA8451Q_ASLP_WAKE_TRANS_ENABLE
- MMA8451Q_ASLP_WAKE_LNDPRT_DISABLE
- MMA8451Q_ASLP_WAKE_LNDPRT_ENABLE
- MMA8451Q_ASLP_WAKE_PULSE_DISABLE
- MMA8451Q_ASLP_WAKE_PULSE_ENABLE
- MMA8451Q_ASLP_WAKE_FF_MT_DISABLE
- MMA8451Q_ASLP_WAKE_FF_MT_ENABLE

13.5.4.4 Auto Sleep State Macro

These macros are used with the **mma8451q_set_aslp_state()** and **mma8451q_get_aslp_state()** functions.

- MMA8451Q_AUTO_SLEEP_DISABLE
- MMA8451Q_AUTO_SLEEP_ENABLE

13.5.4.5 Low Noise State Macro

These macros are used with the **mma8451q_set_low_noise_state()** and **mma8451q_get_low_noise_state()** functions.

- MMA8451Q_LOW_NOISE_DISABLE
- MMA8451Q_LOW_NOISE_ENABLE

13.5.4.6 High Pass Filter State Macro

These macros are used with the **mma8451q_set_hpf_state()** and **mma8451q_set_hpf_state()** functions.

- MMA8451Q_HIGH_PASS_FILTER_DISABLE
- MMA8451Q_HIGH_PASS_FILTER_ENABLE

13.5.5 Motion and Freefall Detection Macro

13.5.5.1 Debounce Counter Mode Macro

These macros are used with the **mma8451q_set_ff_mt_db_cnt_mode()** and **mma8451q_get_ff_mt_db_cnt_mode()** functions. For more information about how to choose this value, please refer to **mma8451q_set_ff_mt_db_cnt_mode()** function description.

- MMA8451Q_FF_MT_CFG_DBCNTM_DECREMENT
- MMA8451Q_FF_MT_CFG_DBCNTM_CLEAR

13.5.5.2 Event Latch Enable Macro

These macros are used with the **mma8451q_set_ff_mt_event_latch_state()** and **mma8451q_get_ff_mt_event_latch_state()** functions. For more information about how to choose this value, please refer to **mma8451q_set_ff_mt_event_latch_state()** function description.

- MMA8451Q_FF_MT_CFG_ELE_DISABLE
- MMA8451Q_FF_MT_CFG_ELE_ENABLE

13.5.5.3 Freefall & Motion Detection Selection Macro

These macros are used with the **mma8451q_set_ff_mt_selection()** and **mma8451q_get_ff_mt_selection()** functions. For more information about how to choose this value, please refer to **mma8451q_set_ff_mt_selection()** function description.

- MMA8451Q_FF_MT_SELECT_FREEFALL
- MMA8451Q_FF_MT_SELECT_MOTION

13.5.5.4 Freefall & Motion Detection Enable Macro

These macros are used with the **mma8451q_set_ff_mt_state()** and **mma8451q_get_ff_mt_state()** functions.

- MMA8451Q_FF_MT_ENABLE_Z
- MMA8451Q_FF_MT_DISABLE_Z
- MMA8451Q_FF_MT_ENABLE_Y
- MMA8451Q_FF_MT_DISABLE_Y
- MMA8451Q_FF_MT_ENABLE_X
- MMA8451Q_FF_MT_DISABLE_X

13.5.5.5 Freefall & Motion Status Macro

These macros are used with the **mma8451q_get_ff_mt_status()** function.

- MMA8451Q_FF_MT_EVENT_ACTIVE
- MMA8451Q_FF_MT_EVENT_Z_DETECT
- MMA8451Q_FF_MT_EVENT_Y_DETECT
- MMA8451Q_FF_MT_EVENT_X_DETECT
- MMA8451Q_FF_MT_EVENT_Z_POSITIVE
- MMA8451Q_FF_MT_EVENT_Z_NEGATIVE
- MMA8451Q_FF_MT_EVENT_Y_POSITIVE
- MMA8451Q_FF_MT_EVENT_Y_NEGATIVE
- MMA8451Q_FF_MT_EVENT_X_POSITIVE
- MMA8451Q_FF_MT_EVENT_X_NEGATIVE

13.5.6 Portrait/Landscape detection Macro

13.5.6.1 Debounce Counter Mode Macro

These macros are used with the **mma8451q_set_lapo_db_cnt_mode()** and **mma8451q_get_lapo_db_cnt_mode()** functions. For more information about how to choose this value, please refer to **mma8451q_set_lapo_db_cnt_mode()** function description.

- MMA8451Q_PL_CFG_DBCNTM_DECREMENT
- MMA8451Q_PL_CFG_DBCNTM_CLEAR

13.5.6.2 Back/Front Trip Angle Threshold Macro

These macros are used with the **mma8451q_set_back_front_threshold()** and **mma8451q_get_back_front_threshold()** functions. For more information about how to choose this value, please refer to **mma8451q_set_back_front_threshold()** function description.

- MMA8451Q_PL_BACK_FRONT_THRESHOLD_65_DEGREE
- MMA8451Q_PL_BACK_FRONT_THRESHOLD_70_DEGREE

- MMA8451Q_PL_BACK_FRONT_THRESHOLD_75_DEGREE
- MMA8451Q_PL_BACK_FRONT_THRESHOLD_80_DEGREE

13.5.6.3 Portrait/Landscape Threshold Macro

These macros are used with the **mma8451q_set_lapo_threshold()** and **mma8451q_get_lapo_threshold()** functions. For more information about how to choose this value, please refer to **mma8451q_set_lapo_threshold()** function description.

- MMA8451Q_PL_THRESHOLD_15_DEGREE
- MMA8451Q_PL_THRESHOLD_20_DEGREE
- MMA8451Q_PL_THRESHOLD_30_DEGREE
- MMA8451Q_PL_THRESHOLD_35_DEGREE
- MMA8451Q_PL_THRESHOLD_40_DEGREE
- MMA8451Q_PL_THRESHOLD_45_DEGREE
- MMA8451Q_PL_THRESHOLD_55_DEGREE
- MMA8451Q_PL_THRESHOLD_60_DEGREE
- MMA8451Q_PL_THRESHOLD_70_DEGREE
- MMA8451Q_PL_THRESHOLD_75_DEGREE

13.5.6.4 Z-Lock Angle Threshold Macro

These macros are used with the **mma8451q_set_z_lock_threshold()** and **mma8451q_get_z_lock_threshold()** functions. For more information about how to choose this value, please refer to **mma8451q_set_z_lock_threshold()** function description.

- MMA8451Q_PL_Z_LOCK_THRESHOLD_14_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_18_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_21_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_25_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_29_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_33_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_37_DEGREE
- MMA8451Q_PL_Z_LOCK_THRESHOLD_42_DEGREE

13.5.6.5 Hysteresis of Portrait/Landscape Trip Macro

These macros are used with the **mma8451q_set_lapo_trip_hys()** and **mma8451q_get_lapo_trip_hys()** functions. For more information about how to choose this value, please refer to **mma8451q_set_lapo_trip_hys()** function description.

- MMA8451Q_PL_HYSTERESIS_0_DEGREE
- MMA8451Q_PL_HYSTERESIS_4_DEGREE
- MMA8451Q_PL_HYSTERESIS_7_DEGREE

- MMA8451Q_PL_HYSTERESIS_11_DEGREE
- MMA8451Q_PL_HYSTERESIS_14_DEGREE
- MMA8451Q_PL_HYSTERESIS_17_DEGREE
- MMA8451Q_PL_HYSTERESIS_21_DEGREE
- MMA8451Q_PL_HYSTERESIS_24_DEGREE

13.5.6.6 Portrait/Landscape Function Enable Macro

These macros are used with the **mma8451q_set_lapo_state()** and **mma8451q_get_lapo_state()** functions.

- MMA8451Q_PL_DISABLE
- MMA8451Q_PL_ENABLE

13.5.6.7 Portrait/Landscape Status Macro

These macros are used with the **mma8451q_get_lapo_status()** function.

- MMA8451Q_PL_STATUS_NEWLP
  - BAFRO and/or LAPO and/or Z-Tilt lockout value has changed.
- MMA8451Q_PL_STATUS_LO
  - Z-Tilt lockout trip angle has been exceeded. Lockout has been detected.
- MMA8451Q_PL_STATUS_LAPO
  - Landscape/Portrait orientation field.
- MMA8451Q_PL_STATUS_PU
  - Equipment standing vertically in the normal orientation.
- MMA8451Q_PL_STATUS_PD
  - Equipment standing vertically in the inverted orientation.
- MMA8451Q_PL_STATUS_LR
  - Equipment isin landscape mode to the right.
- MMA8451Q_PL_STATUS_LL
  - Equipment is in landscape mode to the left.
- MMA8451Q_PL_STATUS_BAFRO
  - Back or Front orientation field.
- MMA8451Q_PL_STATUS_FRONT
  - Equipment is in the front facing orientation.
- MMA8451Q_PL_STATUS_BACK
- Equipment is in the back facing orientation.

13.5.7 Pulse Detection Macro

13.5.7.1 Axis Define Macro

These macros are used with the **mma8451q_set_pulse_threshold()** and **mma8451q_set_pulse_threshold()** functions.

- MMA8451Q_PULSE_AXIS_X
- MMA8451Q_PULSE_AXIS_Y
- MMA8451Q_PULSE_AXIS_Z

13.5.7.2 Double Pulse Abort Macro

These macros are used with the **mma8451q_set_double_pulse_abort()** and **mma8451q_get_double_pulse_abort()** functions. For more information about how to choose this value, please refer to **mma8451q_set_double_pulse_abort()** function description.

- MMA8451Q_PULSE_CFG_DPA_DISABLE
- MMA8451Q_PULSE_CFG_DPA_ENABLE

13.5.7.3 Event Latch Enable Macro

These macros are used with the **mma8451q_set_pulse_event_latch_state()** and **mma8451q_get_pulse_event_latch_state()** functions. For more information about how to choose this value, please refer to **mma8451q_set_pulse_event_latch_state()** function description.

- MMA8451Q_PULSE_CFG_ELE_DISABLE
- MMA8451Q_PULSE_CFG_ELE_ENABLE

13.5.7.4 High Pass Filter Macro

These macros are used with the **mma8451q_set_pulse_hpf_state()** and **mma8451q_get_pulse_hpf_state()** functions.

- MMA8451Q_PULSE_HIGH_PASS_FILTER_DISABLE
- MMA8451Q_PULSE_HIGH_PASS_FILTER_ENABLE

13.5.7.5 Low Pass Filter Macro

These macros are used with the **mma8451q_set_pulse_lpf_state()** and **mma8451q_set_pulse_lpf_state()** functions.

- MMA8451Q_PULSE_LOW_PASS_FILTER_DISABLE
- MMA8451Q_PULSE_LOW_PASS_FILTER_ENABLE

13.5.7.6 Tap Detection Enable Macro

These macros are used with the **mma8451q_set_pulse_detect_state()** and **mma8451q_get_pulse_detect_state()** functions.

- MMA8451Q_SINGLE_PULSE_ENABLE_Z
- MMA8451Q_SINGLE_PULSE_DISABLE_Z
- MMA8451Q_SINGLE_PULSE_ENABLE_Y
- MMA8451Q_SINGLE_PULSE_DISABLE_Y
- MMA8451Q_SINGLE_PULSE_ENABLE_X
- MMA8451Q_SINGLE_PULSE_DISABLE_X
- MMA8451Q_DOUBLE_PULSE_ENABLE_Z
- MMA8451Q_DOUBLE_PULSE_DISABLE_Z
- MMA8451Q_DOUBLE_PULSE_ENABLE_Y
- MMA8451Q_DOUBLE_PULSE_DISABLE_Y
- MMA8451Q_DOUBLE_PULSE_ENABLE_X
- MMA8451Q_DOUBLE_PULSE_DISABLE_X

13.5.7.7 Pulse Event Status Macro

These macros are used with the **mma8451q_get_pulse_detect_status()** function.

- MMA8451Q_PULSE_EVENT_ACTIVE
- MMA8451Q_PULSE_EVENT_Z_DETECT
- MMA8451Q_PULSE_EVENT_Y_DETECT
- MMA8451Q_PULSE_EVENT_X_DETECT
- MMA8451Q_DOUBLE_PULSE_EVENT_DETECT
- MMA8451Q_PULSE_EVENT_Z_POSITIVE
- MMA8451Q_PULSE_EVENT_Z_NEGATIVE
- MMA8451Q_PULSE_EVENT_Y_POSITIVE
- MMA8451Q_PULSE_EVENT_Y_NEGATIVE
- MMA8451Q_PULSE_EVENT_X_POSITIVE
- MMA8451Q_PULSE_EVENT_X_NEGATIVE

13.5.8 Transient Detection Macro

13.5.8.1 Debounce Counter Mode Macro

These macros are used with the **mma8451q_set_transient_db_cnt_mode()** and **mma8451q_get_transient_db_cnt_mode()** functions. For more information about how to choose this value, please refer to **mma8451q_set_transient_db_cnt_mode()** function description.

- MMA8451Q_TRANSIENT_CFG_DBCNTM_DECREMENT
- MMA8451Q_TRANSIENT_CFG_DBCNTM_CLEAR

13.5.8.2 Event Latch Enable Macro

These macros are used with the **mma8451q_set_transient_event_latch_state()** and **mma8451q_get_transient_event_latch_state()** functions. For more information about how to choose this value, please refer to **mma8451q_set_transient_event_latch_state()** function description.

- MMA8451Q_TRANSIENT_CFG_ELE_DISABLE
- MMA8451Q_TRANSIENT_CFG_ELE_ENABLE

13.5.8.3 Bypass High-Pass Filter Macro

These macros are used with the **mma8451q_set_transient_bypass_hpf()** and **mma8451q_get_transient_bypass_hpf()** functions.

- MMA8451Q_TRANSIENT_CFG_HPF_ENABLE
- MMA8451Q_TRANSIENT_CFG_HPF_DISABLE

13.5.8.4 Transient Detection Enable Macro

These macros are used with the **mma8451q_set_transient_state()** and **mma8451q_get_transient_state()** functions.

- MMA8451Q_TRANSIENT_ENABLE_Z
- MMA8451Q_TRANSIENT_DISABLE_Z
- MMA8451Q_TRANSIENT_DISABLE_Y
- MMA8451Q_TRANSIENT_ENABLE_X
- MMA8451Q_TRANSIENT_DISABLE_X

13.5.8.5 Transient Status Macro

These macros are used with the **mma8451q_get_transient_status()** function.

- MMA8451Q_TRANSIENT_EVENT_ACTIVE
- MMA8451Q_TRANSIENT_EVENT_Z_DETECT
- MMA8451Q_TRANSIENT_EVENT_Y_DETECT
- MMA8451Q_TRANSIENT_EVENT_X_DETECT
- MMA8451Q_TRANSIENT_EVENT_Z_POSITIVE
- MMA8451Q_TRANSIENT_EVENT_Z_NEGATIVE
- MMA8451Q_TRANSIENT_EVENT_Y_POSITIVE
- MMA8451Q_TRANSIENT_EVENT_Y_NEGATIVE
- MMA8451Q_TRANSIENT_EVENT_X_POSITIVE
- MMA8451Q_TRANSIENT_EVENT_X_NEGATIVE

13.6 MMA8451Q Driver Data Typedef

13.6.1 MMA8451Q Initialize Typedef

MMA8451Q_INIT_STRUCT is defined in mma8451q_basic.h

Field description:

- SLAVE_ADDRESS:

The slave address that the sensor can be addressed on the i2c bus. The slave address should be set according to mma8451q **SA0** Pin connection. If **SA0** connect to logic 0, slave address will be 0x1C. If **SA0** connect to logic 1, slave address will be 0x1D. For more information, please refer to MMA8451Q datasheet.

- OUTPUT_DATA_RATE:

This field configures the output data rate of the accelerometer, and should be set according to application requirement. This field can be set to one of the output data rate macro defined in mma8451q_basic.h

- FULL_SCALE_RANGE:

This field configures the full scale range of the accelerometer, and should be set according to application requirement. This field can be set to one of the full scale range macro defined in mma8451q_basic.h

- ACTIVE_POWER_SCHEME:

This field configures the power scheme of ACTIVE operating mode. This value is closely related to power consumption and resolution. This field can be set to one of the active power scheme macro defined in mma8451q_basic.h

- BURST_READ_MODE:

This field configures the data length of output acceleration. For **Normal mode**, output data length is 14-bit. For **fast read mode**, output data length is 8-bit. This filed should be set according to application requirement and it can be set to one of the burst read mode macro defined in mma8451q_basic.h

13.7 Error Codes

No additional error codes are generated.

13.8 Example

The source code of the MMA8451Q driver example is located in `mqx\examples\sensor\mma8451q` directory. There are 6 examples in the directory:

Example Name	Description
freefall_motion	This example is for Freefall / Motion detection function.
generic	This example is for generic data acquisition function.
low_power	This example is for mma8451q power saving function.
portrait_landscape	This example is for Landscape Portrait detection function.

pulse	This example is for Pulse detection function.
transient	This example is for Transient detection function.

User can develop their application base on the examples listed above.

Chapter 14 MAG3110 Digital Magnetometer Driver

14.1 Overview

This chapter describes the MAG3110 device driver. The driver defines interface for MAG3110 Three-Axis Digital Magnetometer and accompanies the MQX release.

14.2 Source Code Location

The source code of the MAG3110 driver is located in `mqx\source\io\sensor\mag3110` directory.

14.3 Header Files

- To use MAG3110 device driver, include the header file *mag3110.h* in your application.
- The file *mag3110_basic.h* contains basic level I/O driver function declarations and useful macros.
- The file *mag3110_fun.h* contains functional level IO driver function declarations.
- The file *mag3110_reg.h* contains register definitions and bit field masks.
- The file *mag3110_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.

14.4 MAG3110 Driver API Description

The following section lists the various functions of the MAG3110 library.
The mag3110 driver is divided into 2 layers: basic level driver and functional level driver. Basic level driver just include init/deinit and register read/write functions. Functional level driver include a set of meaningful, easy to use functions. Just using basic level functions listed in *mag3110_basic.h* in your application can reduce total code size. you can also make your own higher level driver based on it.

14.4.1 How To Use This Driver

1. Open an I2C connection by using the **fopen()** function and store the I2C pointer in a MQX_FILE_PTR type variable for the need of **mag3110_init()** function.
2. Set the I2C bus to master mode and set the bus speed according to your application requirement.
3. Program the slave address, ADC sample rate, over sample ratio, burst read mode, auto magnetometer reset mode and data correction mode using the **mag3110_init()** function.
4. Optionally you can configure the following parameters without re-initialization (that is, there is no need to call the **mag3110_init()** function again).
   - Set the slave address by using the **mag3110_set_slave_address()** function.

- Set the ADC sample rate by using the **mag3110_set_adc_sample_rate()** function.
- Set the over sample ratio by using the **mag3110_set_over_sample_ratio()** function.
- Set the burst read mode by using the **mag3110_set_burst_read_mode()** function.
- Set the auto magnetometer reset mode by using the **mag3110_set_auto_mrst()** function.
- Set the data correction mode by using the **mag3110_set_output_correction()** function.

5. Set data correction offset by using the **mag3110_set_user_offset()** function, if the data correction feature is enabled by using the **mag3110_set_output_correction()** function or in the initialization structure.
6. Enable and configure the corresponding GPIO pin interrupt which connected to mag3110 INT1 pin (Interrupt - active high output), if interrupt driven operation is needed. Interrupt will be generated when new measurement data is available. INT1 is cleared when measurement data is read.
7. Switch the device to active mode by using **mag3110_set_operating_mode()** function. After that, the sensor will acquire data and store the data into internal register continuously.

NOTE

To use the sensor in manual trigger operation mode, user should keep the sensor working in standby mode. For such purpose, you can use **mag3110_set_operating_mode()** to switch the sensor to standby mode.

14.4.2 Initialization and Configuration Functions

This section provides a set of functions which can re-configure the available features after device initialization.
Some of the functions can be only used in STANDBY mode, make sure that MAG3110 is working in STANDBY mode before call these functions. For further information, please refer to function's description.

- **mag3110_init()**
- **mag3110_deinit()**
- **mag3110_set_slave_address()**
- **mag3110_set_user_offset()**
- **mag3110_set_adc_sample_rate()**
- **mag3110_set_over_sample_ratio()**
- **mag3110_set_burst_read_mode()**
- **mag3110_set_operating_mode()**
- **mag3110_set_auto_mrst()**
- **mag3110_set_output_correction()**
- **mag3110_reset_mag_sensor()**

14.4.3 Basic I/O Functions

This section provides a set of functions which can be used to access MAG3110 internal register through I2C bus.
Basic I/O level driver aims at provide user a fundamental interface with MAG3110. User can write their own high level driver based on it. To use basic I/O functions, just include *mag3110.h* in your application and call basic level driver after **mag3110_init()**. *mag3110_basic.h* include basic level driver function declarations and many useful macros which can reduce user programming difficulty.

NOTE

User can use basic I/O functions and functional driver functions interlaced in their application.

- **mag3110_write_reg()**
- **mag3110_read_reg()**
- **mag3110_write_single_reg()**
- **mag3110_read_single_reg()**

14.4.4 Data Acquisition Functions

This section provides a set of functions which can be used to get magnetometer output data and mag3110 build-in temperature sensor output data.

- **mag3110_get_temperature()**
- **mag3110_get_mag_data()**
- **mag3110_trigger_measurement()**

14.4.5 Status Inquiry Functions

This section provides a set of functions which can be used to get the status of current configuration or useful information (for example device ID).

- **mag3110_get_slave_address()**
- **mag3110_get_dr_status()**
- **mag3110_get_device_id()**
- **mag3110_get_system_mode()**
- **mag3110_get_user_offset()**
- **mag3110_get_adc_sample_rate()**
- **mag3110_get_over_sample_ratio()**
- **mag3110_get_burst_read_mode()**
- **mag3110_get_operating_mode()**
- **mag3110_get_output_correction()**
- **mag3110_get_reset_status()**

14.4.6 Function Descriptions

This section describes MAG3110 driver functions in detail.

14.4.6.1 Initialization and Configuration Functions

14.4.6.1.1 mag3110_init

Function Name:

```
void * mag3110_init
(
    MAG3110_INIT_STRUCT       *mag3110_init_ptr,
    MQX_FILE_PTR               fd
)
```

Function Description:

Initialize MAG3110 with parameter set in MAG3110 initialize structure. This function should be called after i2c driver initialization and before other mag3110 driver be called. It will initialize the mag3110 slave address, ADC sample rate, over sample ratio, burst read mode, auto magnetometer reset mode and data correction mode defined in **MAG3110_INIT_STRUCT**. After initialization, the mag3110 will stay in **STANDBY** operation mode.

Parameters:

- *mag3110_init_ptr[IN]*: MAG3110 Initialize structure pointer.
- *fd[IN]*: File pointer for the I2C channel connected to mag3110.

Return Value:

- **MAG3110_handle** if initialize successful.
- **NULL** if mag3110 initialize failed.

Note:

None.

14.4.6.1.2 mag3110_deinit

Function Name:

```
bool mag3110_deinit
(
    void          *mag3110_handle
)
```

Function Description:

mag3110_deinit() function will force the device operation mode back into STANDBY mode and recover all the mag3110 register content to the default value.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.1.3 mag3110_set_slave_address

Function Name:

```
bool mag3110_set_slave_address
(
    void         *mag3110_handle,
    uint8_t       slave_address
)
```

Function Description:

Configure the mag3110 driver's internal data structure slave address field. The slave address is used when MCU communicate with mag3110 during address cycle.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *slave_address[IN]*: Slave address to be set.

Return Value:

- **TRUE** if successful.

Note:

Slave address should be set to **MAG3110FCR1_ADDRESS** or **FXMS3110CDR1_ADDRESS** defined in mag3110_reg.h.

14.4.6.1.4 mag3110_set_user_offset

Function Name:

```
bool mag3110_set_user_offset
(
    void            *mag3110_handle,
    int16_t          offset_x,
    int16_t          offset_y,
    int16_t          offset_z
)
```

Function Description:

Set the user offset of magnetometer output data, if output data correction feature is enabled using **mag3110_set_output_correction()** or through mag3110 initialize structure.

The maximum range for the user offsets is in the range -10,000 to 10,000 bit counts comprising the sum of the correction for the sensor zero-flux offset and the PCB hard-iron offset (range -1000 T to 1000 T or -10,000 to 10,000 bit counts).

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *offset_x[IN]*: User Offset Correction on x axis to be set.
- *offset_y[IN]*: User Offset Correction on y axis to be set.
- *offset_z[IN]*: User Offset Correction on z axis to be set.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.1.5 mag3110_set_adc_sample_rate

Function Name:

```
bool mag3110_set_adc_sample_rate
(
    void        *mag3110_handle,
    uint8_t      adc_sample_rate
)
```

Function Description:

mag3110_set_adc_sample_rate() function is used to set the mag3110 internal ADC's sample rate. This field is closely related to output data rate and power consumption level. How to choose this value is listed in MAG3110 datasheet "Table. Over-Sampling Ratio and Data Rate Description".

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *adc_sample_rate [IN]*: ADC sample rate value to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be use with adc sample rate macro in mag3110_basic.h.

14.4.6.1.6 mag3110_set_over_sample_ratio

Function Name:

```
bool mag3110_set_over_sample_ratio
(
    void          *mag3110_handle,
    uint8_t        over_sample_ratio
)
```

Function Description:

mag3110_set_over_sample_ratio() function is used to set the over sample ratio of the mag3110 internal data acquisition logic. Over sample ratio is related to resolution. If the over sample ratio is increased, the final resolution will increase under same adc sample rate at same time, but the output data rate will slow down with same ratio.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *over_sample_ratio[IN]*: Over Sample Ratio value to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.

- Should be used with over sample ratio macro in mag3110_basic.h.

14.4.6.1.7 mag3110_set_burst_read_mode

Function Name:

```
bool mag3110_set_burst_read_mode
(
    void            *mag3110_handle,
    uint8_t          read_mode
)
```

Function Description:

mag3110_set_burst_read_mode() is used to enable/disable the burst read mode of the sensor. This field should be set according to application requirement. For application need precise output data, burst read mode should be set to **NORMAL_MODE**. In this mode, magnetometer output data will be 16-bit width. For application need higher i2c bus access speed, burst read mode should be set to **BURST_READ_MODE**. In this mode, magnetometer output data will be 8-bit width.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *read_mode[IN]*: Burst read mode to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be use with burst read mode macro in mag3110_basic.h.

14.4.6.1.8 mag3110_set_operating_mode

Function Name:

```
bool mag3110_set_operating_mode
(
    void          *mag3110_handle,
    uint8_t        operating_mode
)
```

Function Description:

mag3110_set_operating_mode() function is used to set the operating mode of the sensor. **ACTIVE** mode will make periodic measurements based on values programmed in the ADC sample rate and Over Sampling Ratio fields.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *operating_mode[IN]*: Operating mode to be set.

Return Value:

- **TRUE** if successful.

Note:

Should be use with operating mode macro in mag3110_basic.h.

14.4.6.1.9 mag3110_set_auto_mrst

Function Name:

```
bool mag3110_set_auto_mrst
(
    void         *mag3110_handle,
    uint8_t       auto_reset
)
```

Function Description:

This function is similar to **mag3110_reset_mag_sensor()**, however, the resets occur automatically before each data acquisition. This feature is recommended to be always explicitly enabled by the host application.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *auto_reset[IN]*: Automatic magnetic sensor reset configuration to be set.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be use with automatic magnetic sensor reset macro in mag3110_basic.h.

14.4.6.1.10 mag3110_set_output_correction

Function Name:

```
bool mag3110_set_output_correction
(
    void          *mag3110_handle,
    uint8_t        output_correction
)
```

Function Description:

mag3110_set_output_correction() function is used to configure the magnetometer output data correction. Enable this feature data values will be corrected by the user offset set using **mag3110_set_user_offset()** function.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *output_correction[IN]*: Data output correction configuration.

Return Value:

- **TRUE** if successful.

Note:

- This function can only be used when the device is in "**STANDBY**" mode.
- Should be use with data correction macro in mag3110_basic.h.

14.4.6.1.11 mag3110_reset_mag_sensor

Function Name:

```
bool mag3110_reset_mag_sensor
(
    void      *mag3110_handle
)
```

Function Description:

This function will initiate a magnetic sensor reset cycle that will restore correct operation after exposure to an excessive magnetic field which exceeds the Full Scale Range but is less than the Maximum Applied Magnetic Field.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.

Return Value:

- **TRUE** if successful.

Note:

This function can only be used when the device is in "**STANDBY**" mode.

14.4.6.2 Basic I/O Functions

14.4.6.2.1 mag3110_write_reg

Function Name:

```
bool mag3110_write_reg
(
    void            *mag3110_handle,
    uint8_t          addr,
    uint8_t         *buffer,
    uint16_t         n
)
```

Function Description:

MAG3110 basic multi-byte write function.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *addr[IN]*: MAG3110 register address.
- *buffer[IN]*: Buffer for write function.
- *n[IN]*: Number of bytes to be sent.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.2.2 mag3110_read_reg

Function Name:

```
bool mag3110_read_reg
(
```

```
    void           *mag3110_handle,
    uint8_t         addr,
    uint8_t        *buffer,
    uint16_t        n
)
```

Function Description:

MAG3110 basic multi-byte read function.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *addr[IN]*: MAG3110 register address.
- *buffer[IN]*: Buffer for read function.
- *n[IN]*: Number of bytes to be read.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.2.3 mag3110_write_single_reg

Function Name:

```
bool mag3110_write_single_reg
(
    void           *mag3110_handle,
    uint8_t         addr,
    uint8_t         data
)
```

Function Description:

MAG3110 basic single byte write function.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *addr[IN]*: MAG3110 register address.
- *data[IN]*: Data to be written into MAG3110.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.2.4 mag3110_read_single_reg

Function Name:

```
bool mag3110_read_single_reg
(
    void           *mag3110_handle,
    uint8_t         addr,
```

```
    uint8_t         *buffer
)
```

Function Description:

MAG3110 basic single byte read function.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *addr[IN]*: MAG3110 register address.
- *buffer[IN]*: Buffer to store single register value.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.3 Data Acquisition Functions

14.4.6.3.1 mag3110_get_temperature

Function Name:

```
bool mag3110_get_temperature
(
    void          *mag3110_handle,
    int8_t        *buffer
)
```

Function Description:

This function will read the mag3110 die temperature and store it into given buffer. The sensitivity of the temperature sensor is factory trimmed to 1°C/LSB.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store die temperature value.

Return Value:

- **TRUE** if successful.

Note:

The temperature sensor offset is not factory trimmed and must be calibrated by the user software if higher absolute accuracy is required.

14.4.6.3.2 mag3110_get_mag_data

Function Name:

```
bool mag3110_get_mag_data
(
    void          *mag3110_handle,
    int16_t       *data_x,
    int16_t       *data_y,
    int16_t       *data_z
```

```
)
```

Function Description:

This function will get magnetic field strength in 3 axes and store them into given buffer. The output data will be 16-bit width, if **BUREST READ MODE** is disabled. The output data will be 8-bit width(the MSB 8-bit data of **NORAMAL MODE**), if **BUREST READ MODE** is enabled. For burst read mode selection method, please refer to **mag3110_set_burst_read_mode()** function description. The sensitivity of the magnetic field strength sensor is factory trimmed to 0.10 µT/LSB.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *data_x[OUT]*: Buffer to store magnetic field strength on x axis.
- *data_y[OUT]*: Buffer to store magnetic field strength on y axis.
- *data_z[OUT]*: Buffer to store magnetic field strength on z axis.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.3.3 mag3110_trigger_measurement

Function Name:

```
bool mag3110_trigger_measurement
(
    void            *mag3110_handle
)
```

Function Description:

This function will trigger immediate measurement single time. If part is in ACTIVE mode, any measurement in progress will continue with the highest ODR possible for the selected OSR. In STANDBY mode triggered measurement will occur immediately and part will return to STANDBY mode as soon as the measurement is complete.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.4 Status Inquiry Functions

14.4.6.4.1 mag3110_get_slave_address

Function Name:

```
bool mag3110_get_slave_address
(
```

```
    void            *mag3110_handle,
    uint8_t         *buffer
)
```

Function Description:

This function will store current mag3110 slave address configuration into given buffer. For more information about slave address, please refer to **mag3110_set_slave_address()** and mag3110 initialization structure SLAVE_ADDRESS field.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store slave address.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.4.2 mag3110_get_dr_status

Function Name:

```
bool mag3110_get_dr_status
(
    void            *mag3110_handle,
    uint8_t         *buffer
)
```

Function Description:

This function stores current data ready status into given buffer. The data ready status can be one of the data ready status macro defined in mag3110_basic.h or their combination.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store Data Ready Status.

Return Value:

- **TRUE** if successful.

Note:

Should be used with data ready status macro in mag3110_basic.h.

14.4.6.4.3 mag3110_get_device_id

Function Name:

```
bool mag3110_get_device_id
(
    void          *mag3110_handle,
    uint8_t       *buffer
)
```

Function Description:

This function gets mag3110 Device ID Number and stores the number into given buffer. The Device ID Number read back should match MAG3110_DEVICE_ID.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store WHO_AM_I Register.

Return Value:

- **TRUE** if successful.

Note:

None.

14.4.6.4.4 mag3110_get_system_mode

Function Name:

```
bool mag3110_get_system_mode
(
    void            *mag3110_handle,
    uint8_t         *buffer
)
```

Function Description:

This function gets mag3110 system mode and stores it into given buffer. System mode should match one of the system mode macro defined in mag3110_biasic.h

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store SYSMOD Register value.

Return Value:

- **TRUE** if successful.

Note:

Should be used with system mode macro in mag3110_basic.h

14.4.6.4.5 mag3110_get_user_offset

Function Name:

```
bool mag3110_get_user_offset
(
    void            *mag3110_handle,
    int16_t         *offset_x,
    int16_t         *offset_y,
    int16_t         *offset_z
)
```

Function Description:

This function gets user offset correction on x, y and z axes.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *offset_x[OUT]*: User Offset Correction on x axis from OFF_X reg.

- *offset_y[OUT]*: User Offset Correction on y axis from OFF_Y reg.
- *offset_z[OUT]*: User Offset Correction on z axis from OFF_Z reg.

Return Value:
- **TRUE** if successful.

Note:

None.

14.4.6.4.6 mag3110_get_adc_sample_rate

Function Name:
```
bool mag3110_get_adc_sample_rate
(
    void          *mag3110_handle,
    uint8_t       *buffer
)
```
Function Description:
This function is used to get current ADC sample rate and stores it into given buffer. For more information about ADC sample rate, please refer to **mag3110_set_adc_sample_rate()** function.
Parameters:
- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store ADC sample rate.

Return Value:
- **TRUE** if successful.

Note:

Should be used with adc sample rate macro in mag3110_basic.h.

14.4.6.4.7 mag3110_get_over_sample_ratio

Function Name:
```
bool mag3110_get_over_sample_ratio
(
    void            *mag3110_handle,
    uint8_t         *buffer
)
```
Function Description:
This function gets current Over Sample Ratio and stores it into given buffer. For more information about Over Sample Ratio, please refer to **mag3110_set_over_sample_ratio()** function description.
Parameters:
- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store Over Sample Ratio.

Return Value:
- **TRUE** if successful.

Note:

Should be used with over sample ratio macro in mag3110_basic.h.

14.4.6.4.8 mag3110_get_burst_read_mode

Function Name:

```
bool mag3110_get_burst_read_mode
(
    void          *mag3110_handle,
    uint8_t       *buffer
)
```

Function Description:

This function gets current Burst Read Mode and stores it into given buffer. For more information about Burst Read Mode, please refer to **mag3110_set_burst_read_mode()** function description.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store Burst Read Mode.

Return Value:

- **TRUE** if successful.

Note:

Should be used with burst read mode macro in mag3110_basic.h.

14.4.6.4.9 mag3110_get_output_correction

Function Name:

```
bool mag3110_get_output_correction
(
    void          *mag3110_handle,
    uint8_t       *buffer
)
```

Function Description:

This function gets current data output correction configuration and stores it into given buffer. For more information about data output correction configuration, please refer to **mag3110_set_output_correction()** function description.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store data output correction configuration.

Return Value:

- **TRUE** if successful.

Note:

Should be used with data correction macro in mag3110_basic.h.

14.4.6.4.10 mag3110_get_reset_status

Function Name:

```
bool mag3110_get_reset_status
```

```
(
    void          *mag3110_handle,
    uint8_t       *buffer
)
```

Function Description:

This function is used to get current reset progress and stores it into given buffer. For more information about reset progress, please refer to **mag3110_reset_mag_sensor()** function description.

Parameters:

- *mag3110_handle[IN]*: MAG3110 device instance handler.
- *buffer[OUT]*: Buffer to store current sensor reset status.

Return Value:

- **TRUE** if successful.

Note:

Should be used with MAG3110_CTRL_REG2_MAG_RST_MASK in mag3110_reg.h

14.5 MAG3110 Driver Defines

14.5.1 I2C Slave Address Macro

These macros are used with **mag3110_set_slave_address()** & **mag3110_get_slave_address()** function to set or identify current i2c slave address of mag3110 device. MAG3110_DEFAULT_ADDRESS is equal to MAG3110FCR1_ADDRESS.

- MAG3110FCR1_ADDRESS
- FXMS3110CDR1_ADDRESS
- MAG3110_DEFAULT_ADDRESS

14.5.2 MAG3110 Device ID Number

This macro is used with **mag3110_get_device_id()** function to distinguish mag3110 device from other i2c slave device.

- MAG3110_DEVICE_ID
  - This value is factory programmed to 0xC4.

14.5.3 Data Ready Status Macro

These macros are used with **mag3110_get_dr_status()** function.

- MAG3110_DATA_READY_ZYXOW
  - Previous X or Y or Z data was overwritten by new X or Y or Z data before it was completely read.
- MAG3110_DATA_READY_ZOW
  - Previous Z-axis data was overwritten by new Z-axis data before it was read.

- MAG3110_DATA_READY_YOW
  - Previous Y-axis data was overwritten by new Y-axis data before it was read.
- MAG3110_DATA_READY_XOW
  - Previous X-axis data was overwritten by new X-axis data before it was read.
- MAG3110_DATA_READY_ZYXDR
  - New set of data is ready.
- MAG3110_DATA_READY_ZDR
  - New Z-axis data is ready.
- MAG3110_DATA_READY_YDR
  - New Y-axis data is ready.
- MAG3110_DATA_READY_XDR
  - New X-axis data is ready.

14.5.4 System Mode Macro

These macros are used with **mag3110_get_system_mode()** function. The system mode should equals to one of the following macros.

- MAG3110_SYSMOD_STANDBY
  - STANDBY mode.
- MAG3110_SYSMOD_ACTIVE_RAW
  - ACTIVE mode with RAW data.
- MAG3110_SYSMOD_ACTIVE_NORMAL
  - ACTIVE mode with non-RAW user-corrected data.

14.5.5 ADC Sample Rate Macro

These macros are used with **mag3110_set_adc_sample_rate()** and **mag3110_get_adc_sample_rate()** function. For more information about how to choose ADC sample rate, please refer to mag3110 data sheet.

- MAG3110_ADC_SAMPLE_RATE_80HZ
- MAG3110_ADC_SAMPLE_RATE_160HZ
- MAG3110_ADC_SAMPLE_RATE_320HZ
- MAG3110_ADC_SAMPLE_RATE_640HZ
- MAG3110_ADC_SAMPLE_RATE_1280HZ

14.5.6 Over Sample Ratio Macro

These macros are used with **mag3110_set_over_sample_ratio()** and **mag3110_get_over_sample_ratio()** function. For more information about how to choose over sample ratio, please refer to mag3110 data sheet.

- MAG3110_OVER_SAMPLE_RATIO_16
- MAG3110_OVER_SAMPLE_RATIO_32
- MAG3110_OVER_SAMPLE_RATIO_64
- MAG3110_OVER_SAMPLE_RATIO_128

14.5.7 Burst Read Mode Macro

These macros are used with **mag3110_set_burst_read_mode()** and **mag3110_get_burst_read_mode()** function. For more information about how to choose burst read mode, please refer to **mag3110_set_burst_read_mode()** function description.

- MAG3110_BURST_READ_MODE_NORMAL
- MAG3110_BURST_READ_MODE_FAST

14.5.8 Operating Mode Macro

These macros are used with **mag3110_set_operating_mode()** and **mag3110_get_operating_mode()** function. For more information about how to choose operating mode, please refer to **mag3110_set_operating_mode()** function description.

- MAG3110_OPERATING_MODE_STANDBY
- MAG3110_OPERATING_MODE_ACTIVE

14.5.9 Automatic Magnetic Sensor Reset Macro

These macros are used with **mag3110_set_auto_mrst()** and **mag3110_get_auto_mrst()**function. This feature is recommended to be always explicitly enabled by the host application.

- MAG3110_AUTO_MRST_DISABLE
- MAG3110_AUTO_MRST_ENABLE

14.5.10 Data Correction Macro

These macros are used with **mag3110_set_output_correction ()** and **mag3110_get_output_correction ()**function. Data correction is set using **mag3110_set_user_offset()** function.

- MAG3110_OUTPUT_CORRECT_DISABLE
- MAG3110_OUTPUT_CORRECT_ENABLE

14.6 MAG3110 Driver Data Type Description

14.6.1 MAG3110 Initialize Typedef

MAG3110_INIT_STRUCT is defined in mag3110_basic.h
Field description:

- SLAVE_ADDRESS:

The slave address that the sensor can be addressed on the i2c bus. The slave address should be set according to the Part number of the sensor. For more information, please refer to MAG3110 datasheet page 1.

- ADC_SAMPLE_RATE:

This field configures the ADC sample rate of the sensor, and should be set according to application requirement.

- OVER_SAMPLE_RATIO:

This field configures the over sample ratio of the sensor, and should be set according to application requirement.

- BURST_READ_MODE:

This field configures the data length of output Magnetic field strength. For **Normal mode**, output data length is 16-bit. For **fast read mode**, output data length is 8-bit. This filed should be set according to application requirement.

- AUTO_MRST_MODE:

This field configures Automatic Magnetic Sensor Reset. This feature is recommended to be always explicitly enabled by the host application.

- DATA_CORRECTION_MODE:

This field configures output data correction feature. The data correction offset can be set using **mag3110_set_user_offset()** after initialization.

14.7 Error Codes

No additional error codes are generated.

14.8 Example

The source code of the MAG3110 driver example is located in the `mqx\examples\sensor\mag3110` directory.

Chapter 15 Core_mutex Driver

15.1 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.

15.2 Source Code Location

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

15.3 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*).

15.4 API Function Reference

This sections provides functions for the core_mutex MQX RTOS driver.

15.4.1 _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.)

15.4.2 _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.)

15.4.3 _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.)

15.4.4 _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.)

15.4.5 _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.)

15.4.6 _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.)

15.4.7 _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.)

15.4.8 _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.)

15.4.9 _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.)

15.5 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 );
    }
}
```

Chapter 16 Revision History

Table 16-1. Revision history

Revision number	Date	Substantive changes
0	01/2015	Initial release.
1	05/2015	Added the software version number to the document title and document number, and changed the revision number.