MMA9559L Intelligent, Motion-Sensing Platform Software Reference Manual

Devices Supported:
MMA9559L

Document Number: MMA9559LSWRM
Rev. 0.1, 03/2012

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

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

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

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

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

Contents

Chapter 1 About this Document

Chapter 2 Firmware Overview

Chapter 3 User Code Example

Chapter 4 Functional Details

Appendix A
Revision History

Chapter 1 About this Document

1.1 Overview

1.1.1 Purpose

This reference manual describes the features, architecture and programming model of the MMA9559L Intelligent, Motion-Sensing Platform. This device incorporates dedicated accelerometer MEMS transducers, signal conditioning, data conversion, and a 32-bit programmable microcontroller. For information about the device's hardware, see the *MMA955xL Intelligent, Motion-Sensing Hardware Reference Manual* (MMA955xLRM). (See "References" on page 10.)

1.1.2 Audience

This document is primarily for system architects and software application developers who are using or considering use of the MMA9559L device in a system.

1.1.3 Document structure

This document combines a firmware overview with detailed functional documentation and sample user code.

- "Firmware Overview": Provides an overview of the device's memory configuration and firmware requirements and explanations of the firmware's basic functional blocks.
- "User Code Example": Provides sample code that can be used as a template for creating user code.
- "Functional Details": Gives the technical details of the firmware's macros, enumerations, data structures, and functions.

1.2 Terms and acronyms

AFE	Analog Front End
API	Application Programming Interface
CC	Command Complete
CI	Command Interpreter
CMD	Command
COCO	Conversion Complete
CRC	Cyclic Redundancy Check
DFC	Data Format Code
EVM	Evaluation Module
FIFO	First In First Out (Data structure)
FOPT	Flash Options Register
GPIO	General-Purpose Input / Output
IIR	Infinite Impulse Response
MBOX	Mailbox
MCU	Microcontroller
MTIM	Modulo Timer Module
PDB	Program Delay Block
RCSR	Reset Control and Status Register
SFD	Start Frame Digital
TPM	Timer Program Module

1.3 Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value 0, it is said to be cleared; when it takes a value of 1, it is said to be set.
MNEMONICS	In text, instruction mnemonics are shown in uppercase.
`mnemonics`	In code and tables, instruction mnemonics are shown in lowercase.
italics	Italics indicate variable command parameters. Book titles also are italicized.
0x*0*	Prefix of *0x* to denote a hexadecimal number
*0*b	Suffix of *b* to denote a binary number
REG[FIELD]	Abbreviations for registers are shown in uppercase. Specific bits, fields or ranges appear in brackets. For example, RAMBAR[BA] identifies the base address field in the RAM base-address register.
nibble	A 4-bit data unit
byte	An 8-bit data unit
word	A 16-bit data unit
longword	A 32-bit data unit
x	In some contexts, such as signal encodings, x indicates a “do not care.”
n	Used to express an undefined numerical value.
~	NOT logical operator
&	AND logical operator
|	OR logical operator
||	Field concatenation operator
$\overline{\text{OVERLINE}}$	Indicates that a signal is active-low.

1.4 References

1. *MMA955xL Intelligent, Motion-Sensing Hardware Reference Manual* (MMA955xLRM); also see the MMA955xL documentation webpage
2. The *I^2C-Bus Specification*, Version 2.1, January 2000, Philips Semiconductors
3. *I^2C-Bus Specification and User Manual*, NXP Semiconductors Document UM10204, Rev. 03 - 19 June 2007
4. *ColdFire Family Programmer's Reference Manual (CFPRM)* Rev. 3, 03/2005, Freescale Semiconductor, Inc.
5. IIR Filter application note AN4464, *Digital Filtering with MMA955xL*

Chapter 2 Firmware Overview

The MMA9559L device is a member of the Freescale MMA955xL intelligent, motion-sensing platform family. Unlike the other members of the MMA955xL family—where the factory-programmed firmware provides out-of-the-box operation—the MMA9559L only provides a lightweight infrastructure and requires user-programmed firmware for the device to run anything other than the internal, ROM-command-line interpreter.

This approach reduces the factory programmed firmware on the MMA9559L to only 2 KB of flash memory, leaving the other 14 KB for customer firmware.

After an introduction to the firmware file, this chapter provides an overview of the firmware and related hardware topics. The information is divided into the following functional categories:

- "Memory and CPU usage"
- "Hardware support"
- "Events and scheduling"
- "FIFOs"
- "Power"

These same categories are used in the following chapter, that provides functional details.

2.1 Firmware elements and functionality

The **`mma9559.h`** file contains the type and function declarations required to build custom user firmware that can be loaded onto the Freescale MMA9559L device.

In order to use this device, CodeWarrior 10.1 or later with the MMA9550 service pack must be installed for development.

The MMA9559L device's built-in firmware provides the following basic functionality:

- Device initialization: This code clears the RAM within the MMA9559L and sets up the Exception Vector Base Register, Vector Table, and the user and supervisor stack pointers. The firmware enables interrupts and switches to User mode and jumps to the function whose address is stored in the flash location 0x0000_0800.

  If the user flash memory is blank, the Freescale firmware resets back to the ROM command interpreter.
- Exception vector handler: This code provides a thin handler that is called whenever an exception occurs (other than a Trap instruction). This handler saves the state of the volatile registers, disables/masks the interrupts, switches to User mode, and then calls a user-supplied, exception-handler function.

The user-written **`user_exception_handler()`** code returns a bit field that sets bits in the **`mma9559_vars_t`** events field. This can be used to control a simple execution loop in the user firmware.

- Trap vector handler: Traps are software-triggered exceptions that can be issued by user firmware. This code provides a set of functions that can be called by the user firmware. A set of four trap exceptions are reserved for the user firmware that can register user-supplied trap handlers for each of them.

The contents of the **mma9959.h** header file may be used by customer firmware, but *this file should not be modified.*

2.2 Memory and CPU usage

The Freescale firmware uses 2 KB of the MMA9559L device's 16 KB of flash memory and 384 bytes of its 2 KB of RAM.

2.2.1 Flash memory

The MMA9559L has 16 KB of flash memory, located at the address range 0x0000_0000 to 0x0000_3FFF. The flash space is assigned as shown in the following table.

Table 2-1. Firmware flash usage

Flash Region	Size (bytes)	Usage
0x0000_0000 to 0x0000_07FF	2048	Freescale primary firmware, including the exception table.
0x0000_0800 to 0x0000_0803	4	User firmware startup vector: Contains the address of the user firmware startup code to execute.
0x0000_0804 to 0x0000_3FFB	14328	User firmware image.
0x0000_3FFC to 0x0000_3FFF	4	Optional Cyclic Redundancy Check (CRC) and default Flash Options Register (FOPT) value.

The startup vector stored at 0x800 is used to locate the **`_startup()`** function in the user firmware. If this vector is not 0xFFFF_FFFF, the Freescale firmware jumps to it after the firmware initialization is complete. This vector is normally set in the vector table entry in the user project **`exceptions.c`**.

2.2.2 RAM

The MMA9559L has 2 KB of RAM, located at the address range 0x0080_0000 to 0x0080_07FF. This space is assigned as shown in the following table.

Table 2-2. Firmware RAM usage

RAM Region	Size (bytes)	Usage
0x0080_0000 to 0x0080_00FF	256	Variables for ROM code, trim and Freescale firmware
0x0080_0100 to 0x0080_077F	1664	Space for user firmware variables, heap and user stack
0x0080_0780 to 0x0080_07FF	128	Supervisor stack used by the Freescale firmware

The first section of RAM is reserved for use by the Freescale firmware to store variables used by the firmware and the ROM functions.

The second section of 1664 bytes is available for use by the user firmware. This includes space for global variables, the heap, and the user stack. The Freescale firmware initializes the entire user variable and heap region with the value 0, so that all global variables are initialized to 0.

The last section of RAM is reserved for the Freescale firmware to store the supervisor stack, which is used whenever the firmware is running. The **`user_exception_handler()`** and the **`user_trap_handler()`** functions are called from the Freescale firmware, but are run in User mode, using the user stack space rather than using the supervisor stack space.

2.2.3 Supervisor stack usage

The MMA955xL devices use a ColdFire v1 core as the CPU. This CPU contains a set of user registers and a set of supervisor registers. When executing in User mode, only the user registers are accessible, but when running in Supervisor mode, all registers are accessible. The ColdFire CPU enables separate stacks to be used for User and Supervisor modes. The MMA9559L firmware operates in Supervisor mode, but switches back to User mode when executing any user firmware.

The ColdFire CPU uses the A7 and alternate A7 registers to provide the stack pointer, so that the MMA9559L firmware only permits a single level of interrupts to be processed, eliminating the risk of reentrant code, and a limit is placed on the amount of RAM required for the supervisor stack. Interrupts are disabled when the device is in a critical section or executing an MMA9559L firmware function, a user trap call, or the exception handler.

However, the INT pin is a non-maskable interrupt that can always execute, even when the interrupts are masked. The INT pin uses a special, lightweight interrupt handler that converts the non-maskable INT interrupt to a maskable software interrupt. But in doing so, it consumes 12 bytes of stack space.

Usage of the supervisor stack space should be analyzed to ensure that it does not exceed the reserved space, otherwise it starts to overwrite the user stack space and may corrupt the user program.

For details on the different operational scenarios and the amount of stack space used by each function, see “Memory and CPU usage” on page 39.

2.3 Hardware support

User code can access the underlying functionality of the MMA955xL hardware, but the MMA9559L also provides functions to fully enable some key hardware capabilities. This section explains how these functions can be used. For more details, see “Hardware support” on page 42.

2.3.1 Frame interval counter

The MMA955xL hardware is designed on a frame structure. Each frame starts when the FIC (Frame Interval Counter) reaches the configurable frame-rate timer interval and creates a Start-of-Frame exception. This exception can be used in the **`user_exception_handler()`** to start a sequence of activities to be executed in each frame.

The frame-rate interval is set by calling the function **`mma9559_framerate_set()`** with one of the following parameter values:

- FRAMERATE_NONE: Disables the FIC, preventing the device from waking up periodically.
- FRAMERATE_3906HZ: Configures the FIC to run at 3906 Hz
- FRAMERATE_1953HZ: Configures the FIC to run at 1953 Hz
- FRAMERATE_977HZ: Configures the FIC to run at 977 Hz
- FRAMERATE_488HZ: Configures the FIC to run at 488 Hz
- FRAMERATE_244HZ: Configures the FIC to run at 244 Hz
- FRAMERATE_122HZ: Configures the FIC to run at 122 Hz
- FRAMERATE_61HZ: Configures the FIC to run at 61 Hz
- FRAMERATE_30HZ: Configures the FIC to run at 30.5 Hz
- FRAMERATE_15HZ: Configures the FIC to run at 15.3 Hz
- FRAMERATE_8HZ: Configures the FIC to run at 7.6 Hz
- FRAMERATE_4HZ: Configures the FIC to run at 3.8 Hz
- FRAMERATE_2HZ: Configures the FIC to run at 1.9 Hz
- FRAMERATE_1HZ: Configures the FIC to run at 0.95 Hz
- FRAMERATE_POINT5HZ: Configures the FIC to run at 0.48 Hz
- FRAMERATE_POINT2HZ: Configures the FIC to run at 0.24 Hz

Setting the frame rate to FRAMERATE_NONE reduces the power consumption of the device to the lowest level, because the clock to the FIC is disabled. The power consumption will generally increase as the frame rate is increased.

When the frame interval is set to a frame rate of FRAMERATE_NONE, the Start-of-Frame exception does not occur to wake the device. The device can be awakened by the mailbox or INT pin exceptions.

2.3.2 Analog Front End (AFE)

The hardware registers for the Analog Front End (AFE) cannot be directly accessed by user code. They can be controlled and monitored through the functions provided in the Freescale firmware.

The AFE can be configured using the **`mma9559_afe_csr_set()`** function that takes a single parameter which combines one of each of the three configurable aspects of the AFE by OR-ing them together. The three aspects and their configurable elements include:

- G-mode range: Selects the G range operating mode by selecting one of the following:
  - AFE_CSR_GRANGE_8G: Sets the AFE to the +/-8g range
  - AFE_CSR_GRANGE_4G: Sets the AFE to the +/-4g range
  - AFE_CSR_GRANGE_2G: Sets the AFE to the +/-2g range
- Fourth-channel data source: Selects the source of the data for the fourth channel of the AFE:
  - AFE_CSR_C4MODE_NONE: Does not measure anything with the ADC's fourth channel
  - AFE_CSR_C4MODE_TEMP: Uses the ADC's fourth channel to measure the temperature sensor
  - AFE_CSR_C4MODE_EXT: Uses the ADC's fourth channel to measure the external inputs
- AFE conversion mode: Sets the number of bits for the AFE conversion:
  - AFE_CSR_CMODE_10BIT: Performs a 10-bit ADC conversion, so the 6 LSBs is be 0
  - AFE_CSR_CMODE_12BIT: Performs a 12-bit ADC conversion, so the 4 LSBs is be 0
  - AFE_CSR_CMODE_14BIT: Performs a 14-bit ADC conversion, so the 2 LSBs is be 0
  - AFE_CSR_CMODE_16BIT: Performs a 16-bit ADC conversion using all 16 data bits

For example, to configure the AFE to capture 10 bits of data in the 2g range, ignoring the fourth ADC channel, the code in the following example would be used.

Example 2-1.

```
// Configure the AFE Control and Status Register
  mma9559_afe_csr_set((afe_csr_options_t)(
       AFE_CSR_GRANGE_2G   |        // Use +/- 2 g range
       AFE_CSR_C4MODE_NONE |        // Measure X, Y, Z channels only
       AFE_CSR_CMODE_10BIT          // Use 10 bit conversion
  );
```

The current AFE configuration can be read using the **`mma9559_afe_csr_get()`** function.

The AFE conversion is started by calling the **`mma9559_afe_conversion_start()`** function. This is normally called within the **`user_exception_handler()`** when a Start-of-Frame exception occurs, but the function can be called from anywhere in user code.

When the AFE completes the AFE sample, it creates a Conversion-Complete exception. The **`user_exception_handler()`** can respond to this exception by enabling a sequence of operations to use the AFE data. Normally, this sequence is executed in the main execution level, rather than in the **`user_exception_handler()`**.

The AFE registers contain semi-trimmed data values that need to complete the rest of the trimming process before they can be used. In most cases, the **`user_exception_handler()`** responds to the Conversion-Complete exception by signaling an event and the code in the main execution loop reads the trimmed AFE data using code similar to the following example:

Example 2-2.

```
   mma9559_afe_data_t afe_data;      // Variable to hold the trimmed AFE data
   ...
// Read and trim the AFE data
   mma9559_afe_trimmed_sensor_data_get(afe_data.data);
```

NOTE

The signaling of an event is described in more detail in "Events and scheduling" on page 18.

In a few circumstances, it may be desirable to read the raw, semi-trimmed data values and then complete the trimming later. Two situations where this might be useful are:

- A FIFO is being used to pass data from the **`user_exception_handler()`** to the main execution loop
- The data samples are being filtered and decimated before use. (Since the trim process in this device is linear, it is permissible to filter and decimate the semi-trimmed data and then trim the result.)

The FIFO case could be implemented with code like the following example:

Example 2-3.

```
// In the user exception handler
   mma9559_afe_data_t raw_data;      // Variable to hold the semi-trimmed AFE data
   ...
   case VectorNumber_Vconversion_complete:
                                     // AFE conversion complete exception
     mma9559_afe_raw_sensor_data_get(raw_data.data);
                                     // Read semi-trimmed AFE data
     mma9559_fifo_push((mma9559_fifo_t*)&afe_fifo, (uint8*)raw_data.data, 1);
                                     // store in a FIFO break;

// In the main execution loop
   mma9559_afe_data_t raw_data;      // Variable to hold the semi-trimmed AFE data
   mma9559_afe_data_t afe_data;      // Variable to hold the trimmed AFE data
   ...
   case EVENT_AFE_FIFO:              // Get the semi trimmed data from the FIFO
     mma9559_fifo_pop((mma9559_fifo_t*)&afe_fifo, (uint8*)raw_data.data, 1);
     mma9559_afe_raw_sensor_data_trim(afe_data.data, raw_data.data);
                                     // and trim it
   ...                               // continue processing with the trimmed AFE data
```

A more complete description of the FIFO operation is given in "FIFOs" on page 24.

The trim process modifies the gain and offset of the X, Y, Z, and temperature sensor values using part-specific trim values that were measured and calculated during the manufacture of the device. The trim process also applies a set of three, user-specified offsets to the X, Y, and Z axes that can be set to user-configured values, by adding them to the results of the normal trim calculation.

The user-specified offsets can be set using the **`mma9559_afe_offsets_set()`** function and read back using the **`mma9559_afe_offsets_get()`** function. The user-specified offset values are treated as values in the 8g range and adjusted automatically to the appropriate g range when the g range is changed with the **`mma9559_afe_csr_set()`** function. The user-specified offset values can be determined by putting the device into the 8g mode, reading the accelerometer values without the effects of gravity, and then negating these values.

The user offset values are reset whenever the device is reset.

2.3.3 Stop mode control

When there is no further software processing required, the CPU can be put into STOP mode to reduce the power consumption. The MMA955xL hardware enables additional power savings by also enabling the system clock rate to be controlled.

Three different stop clock modes are supported:

- Stop Fast Clock: The CPU is stopped and the clock tree is still running at the normal rate of 8 MHz
- Stop Slow clock: The CPU is stopped and the clock tree is running at a reduced rate of 62.5 kHz.
- Stop No Clock: The CPU is stopped and the clock tree is disabled

There are some limitations on which clock rate can be used and they vary with the hardware functionality being used. For example:

- When the AFE is converting a data sample then the clock must remain at 8 MHz (fast), therefore the Stop Fast Clock mode must be used when stopping the CPU, but not Stop Slow Clock or Stop No Clock.
- If the frame interval counter is being used to wake the device periodically, then the clock must remain running, so either Stop Fast Clock or Stop Slow Clock modes must be used when stopping the CPU, but not Stop No Clock (in fast or slow mode).

To stop the CPU and determine which clock rate to use, the **`mma9559_idle()`** function can be called whenever there is no other software to run. For more information on this usage, see “Events and scheduling” on page 18. For a more detailed explanation of the clock’s operation, see “Power” on page 28.

2.4 Events and scheduling

While the MMA9559L firmware does not provide a task scheduler, it does provide the main components that can be used to create a simple, execution-loop scheduler, in the form of Events. This section explains how these functions can be used. For more details, see “Events and scheduling” on page 56.

2.4.1 Events

The MMA9559L firmware supports up to 32 events, each one represented by a single bit in the **`mma9559_vars_t`** events field. Whenever an event is signaled (activated), the corresponding bit is set in the events field. When the event is handled, the bit is cleared.

The MMA9559L firmware does not use any of the event bits, allowing the assignment of the 32 bits by the user firmware. In this implementation, the least-significant bit of the events bit field corresponds to event 0—which may be treated as the highest-priority event. The most-significant bit of the event field corresponds to event 31, which may be treated as the lowest-priority event.

The currently signaled (active) events can be read directly from the MMA9559L firmware **`mma9559_vars_t`** data structure events field, but it should not be modified except through the return value from the **`user_exception_handler()`** or using the **`mma9559_events_set_clear()`** function.

Events are normally signaled in the **`user_exception_handler()`** which is created by the user and exists in the user firmware. The easiest way to signal events is for the **`user_exception_handler()`** to return an event bit field as the function return value where the corresponding bits have been set for the events that should be signaled. With this, the Freescale exception handler takes care of signaling the events.

This methodology enables the **user_exception_handler()** to complete its work quickly, keeping the interrupt duration short. A simple, **user_exception_handler()** is shown in the following example.

Example 2-4.

```
__declspec(register_abi) events_t user_exception_handler(
   int      vector)                 // exception vector number
{
   events_t events = 0;             // Events to be signaled (defaults to none)

   switch (vector)
   {
   case VectorNumber_Vstart_of_frame:// Start of Frame
      FCSR_SF = 1;                  // Clear interrupt source
      // Add Start of Frame event to the events to be signaled (asserted)
      events |= EVENT_BITFIELD(EVENT_START_OF_FRAME);
      break;

   default:                         // code to handle the other exception sources
      break;
   }
   return events;                   // signal events to be signaled (activated)
}
```

Events can also be managed by FIFOs. (See “FIFOs” on page 24.) In this case, events that are associated with FIFOs are signaled when the FIFO contains data and the FIFO events are cleared when the FIFO is empty.

Events are normally handled in a simple execution loop within the user **main()** function. The user firmware selects an active event bit and performs the required processing. When the processing has completed, the event bit can be cleared using the **mma9559_events_set_clear()** function. Alternatively, if the event is associated with a FIFO, it is cleared when the last data is read from the FIFO.

Two functions are provided to assist with the selection of the event to process:

- **mma9559_events_find_first()** selects the highest-priority event that is active and returns the event number (from 0 to 31) for the chosen event.
- **mma9559_events_find_next()** selects the next active task in a round-robin manner, working from the least-significant bit to the most-significant bit and then wrapping around again.

The **mma9559_idle()** function is used to put the MMA9559L into the lowest permissible power state when no computational work is required. The function considers the events field and if any event bits are still signaled (set) when the **mma9559_idle()** function is called, it returns immediately to allow the events to be processed.

The function only enters a CPU Stop mode to save power when the **mma9559_vars_t** events field is 0. When an exception occurs, the MMA9559L wakes up from STOP mode and runs the exception handler. The **mma9559_idle()** function then returns back to the scheduling loop in the main function.

The **mma9559_idle_use_stop_config()** function can be used by user code to adjust the lowest permissible power state when required by hardware. For more details, see “Power” on page 28.

The discussed commands can be combined in a simple execution loop within the main function, as shown in the following example.

Example 2-5.

```
events_t events;
int event = 0;

// Initialization to set up the Freescale firmware data structure
...

// Loop forever processing events
while (1) {
   // retrieve the bitfield of active events
   events = mma9559_vars_ptr->events;
   if (events) {         // if there are any active events
      // select the next event to process with priority scheduling
      // or use mma9559_events_find_next() for round robin scheduling
      event = mma9559_events_find_first(events);
      // priority scheduling
      switch (event) {
         // insert code to respond to each of the events here
         ....
      }
      // clear event
      mma9559_events_set_clear(0, EVENT_BITFIELD(event));
   }
   else               // if there are no outstanding events then go to sleep
      mma9559_idle();
};
```

Event signaling can also be used creatively by tasks in the main execution loop:

- When one task completes, it can wake up the next task by signaling the appropriate event.
- Internal state variables can be used to process more time-consuming tasks.

  Some tasks may require longer execution times to complete their processing. The event scheduling is cooperative rather than preemptive, requiring long tasks to break up their processing into multiple shorter executions.

  Internal state variables may be used to keep track of the processing state of the task. Such tasks may deliberately not clear the event that causes them to run, but instead leave it set for the task to execute repeatedly to perform the rest of the processing steps. Only when all of the required executions have been performed is the event bit cleared.
- Using FIFOs for inter-process communication may be easier, when some longer tasks are difficult to break up.

The **`mma9559_vars_t`** structure also contains an **`events_missed`** field. The bits in this field are set whenever an event is signaled and the event bit is already active in the event field. This situation indicates that the scheduler may be “losing” events because a new occurrence of the event has been signaled before the processing for the previous event has been completed and the event bit cleared.

The **events_missed** field is never cleared by the Freescale firmware and may be used as a diagnostic to identify problems with the scheduling of event processing. The **events_missed** field has a slightly different operation when associated with FIFOs. See “FIFOs” on page 24.

2.4.2 Initialization

The user firmware must perform some initialization steps to configure the MMA9559L firmware before the user exception handler can run or events can be used. The initialization steps are:

1. Retrieve the address of the MMA9559L Freescale firmware data structure.

   This structure is needed in order to read the currently signaled events and to set the pointers to the user handler functions. The address does not change at run time, and it only needs to be read once.
2. Set the pointer to the **user_exception_handler()** function.

   This must be done before any exceptions are enabled or the firmware may end up trapped in the exception handler because the exception cause is not being cleared.
3. Optionally, set the pointer to the **user_trap_handler()** functions.

   This is only required if user trap functions are being used and the pointer should be set before the user trap is called. If a user trap is called before the pointer is initialized, the trap-call function just returns, but the system does not lock up.

This initialization normally occurs at the beginning of the user main function. The pointer to Freescale firmware data structure can be a global variable, to avoid retrieving it multiple times. The following example gives initialization code.

Example 2-6.

```
void main(void)
{
   ...                    // Instantiate local variables

   // Retrieve the address of the mma9559 variables structure
   mma9559_vars_ptr = mma9559_vars_addr_get();

   // Register the user exception handler (must occur before interrupt sources are enabled)
   mma9559_vars_ptr->user_exception_handler = user_exception_handler;

   // Register any user trap handlers (only needed if there are any user trap handlers)
   mma9559_vars_ptr->user_trap_handler[0] = user_trap_handler0;
   mma9559_vars_ptr->user_trap_handler[1] = user_trap_handler1;
   mma9559_vars_ptr->user_trap_handler[2] = user_trap_handler2;
   mma9559_vars_ptr->user_trap_handler[3] = user_trap_handler3;

   // Interrupt sources can now be enabled
   ...

   // Loop forever processing events
   while (1) {
      ...
   };
}
```

2.4.3 Interrupts and critical sections

Interrupts are treated in a simple manner. Only one level of interrupt is permitted, therefore interrupts can never be nested or reentrant. The only exception is the INT pin IRQ which is non-maskable. This pin is handled with its own special handler, as explained in “Supervisor stack usage” on page 13.

The **`user_exception_handler()`** and **`user_trap_handler()`** functions are always called with interrupts already disabled/masked so that they are not interrupted. The user firmware should not change the interrupt settings inside these functions.

Normal user code executes with interrupts enabled/unmasked. Occasionally, there are small sections of code that should not be interrupted by exceptions, in order to avoid unintended side effects. For example, the **`mma9559_vars_t`** events and **`mma9559_idle_t`** fields may be modified within exception handlers, as a result any changes to these fields within normal user code would be protected.

The **`mma9559_interrupts_disable()`** and **`mma9559_interrupts_restore()`** functions can be used by normal code to create critical sections that cannot be interrupted by exceptions. They may be used in either of the following manners.

In simple implementations—where calls to disable interrupts can never be nested—a simpler form can be used, a shown in the following example.

Example 2-7.

```
mma9559_interrupts_disable();     // start critical section (disable interrupts)
....
mma9559_interrupts_restore(0);    // enable interrupts
```

If there is any possibility that calls to disable interrupts (such as critical sections) may be nested, a more-advanced form can be used that saves the state of the interrupts at the start of the critical section. This form also restores the interrupts to the same state (masked or unmasked) that they were at the start of the critical section. See the following example.

Example 2-8.

```
int status = mma9559_interrupts_disable();
// start critical section (disable interrupts)
....
mma9559_interrupts_restore(status);      // restores previous interrupt mask state
```

The **`mma9559_interrupts_disable()`** function changes the Interrupt Priority Mask in the ColdFire Status Register to 7, that masks all interrupt sources. Any exceptions that occur while the interrupts are disabled are not lost, but remain pending in the hardware. Those pending interrupts are serviced as soon as the interrupt level is restored at the end of the critical section.

Since interrupts are delayed, the critical section should be kept as short as possible to avoid unnecessarily increasing the interrupt latency.

The **`mma9559_interrupts_restore()`** function does not just enable interrupts by changing the Interrupt Priority Mask back to 0. It also restores the mask to the state supplied in the status parameter—which the user should have saved from the result of the **`mma9559_interrupts_disable()`** function call. This distinction enables the interrupt functions to work correctly, even when interrupts are nested accidentally or deliberately.

The **`mma9559_interrupts_restore()`** function uses only the interrupt-priority mask bits of the supplied status parameter. If any of the interrupt priority mask bits are set, the interrupt priority mask is set to 7 to mask all interrupts. Otherwise, the interrupt priority mask is set to 0 to enable/unmask the interrupts.

2.5 FIFOs

Most applications' needs can be met using the Events and Scheduling capabilities, where an event can be signaled in the user exception handler and handled in the main execution loop. This works when the tasks in the execution loop are short and can be completed before the next interrupt or AFE sample.

Occasionally, longer processing is required—processing that takes more time than the interval between interrupts/AFE samples. In such cases, the AFE samples would be lost—if the regular event method was used.

This can be handled in a couple of ways:

- Break the longer tasks into a number of smaller sections which can each be executed separately. (See "Events and scheduling" on page 18.)
- Use FIFOs for inter-process communications, as described in more detail in this section.

FIFOs provide a configurable amount of buffering between tasks in order to store data before being processed. This can be used in situations where the tasks with longer execution times execute infrequently *and* the total average processing time per AFE sample is less than the AFE sample period. The FIFO provides a means for storing data temporarily during the times when the long-execution task runs, so that that data can be retrieved from the FIFO and processed when shorter tasks run.

The MMA9559L firmware includes simple FIFO functionality that enables multiple, user-configurable FIFOs to be created by the user code. For each of the FIFOs, the FIFO buffer size is configured in terms of the maximum number of entries that can be stored and the size of each entry.

All entries within a FIFO are the same size, although different FIFOs may have a different, fixed entry size. The size of each FIFO entry should be determined by the data being stored in it. For example, if a FIFO is storing AFE samples, the entry size might be sizeof(**`mma9559_afe_data_t`**). The maximum number of entries should be determined according to the maximum delay that may be caused by the long-execution tasks.

The FIFO functions also can use the event functionality. When a FIFO is initialized, the event bit field is specified that defines the events associated with the FIFO. Typically, there is either zero or one event associated with a FIFO. If there are any events associated with a FIFO, when data is pushed into the FIFO the associated events are signaled. When the FIFO becomes empty—either because the last data was popped out of the FIFO or the FIFO is reset—the associated events are cleared.

This can be used in conjunction with the simple event scheduling capability to trigger processing to occur whenever the FIFO contains data. In order to easily identify which FIFO contains data, it is best to use an event bit in only one FIFO. That enables the appropriate FIFO to be uniquely identified by the event bit, when the event is signaled.

If an event is associated with a FIFO by being set in the events field when the FIFO is initialized, the event is added to the **`mma9559_vars events_fifos`** field. That prevents the event from being signaled by the **`user_exception_handler()`** code or modified by the **`ma9559_event_set_clear()`** function.

The **`events_missed`** field in the **`mma9559_vars_t structure`** is used slightly differently for events that are associated with FIFOs. In this case, the **`events_missed`** event bit is set whenever the FIFO overflows because it cannot store all of the data entries that are being pushed into it. This situation indicates

that the scheduler may be “losing” events because one or more FIFO samples were not stored on the FIFO, although user code may make some accommodation for this. The **events_missed** field is never cleared by the Freescale firmware and may be used as a diagnostic to identify problems with the chosen FIFO size.

FIFOs are used with the following steps:

1. Instantiate the FIFO variable to reserve the space for the FIFO structure.
2. Initialize the FIFO at run time to configure it before it is used.
3. Push data onto the FIFO, as it becomes available.
4. Pop data off the FIFO in the tasks that process the data.
5. Reset the FIFO to discard the current contents of the FIFO and restore it to its empty state.

FIFOs are normally created as global variables so that they are always permanently allocated and are never de-allocated. They may also be created as local variables within functions, in which case they are allocated on the user stack. If they are on the user stack, the user must ensure that they are not referenced after they have gone out of scope and been removed from the stack.

2.5.1 Instantiate FIFO

The data buffer and control variables for each FIFO are held in a single data structure. The data structure can either be created on a per-instance basis or by creating a data type that enables multiple instances of the same FIFO to be created.

A FIFO instance can be created with the FIFO_STRUCT macro, as in the following code that creates the FIFO test_fifo. That FIFO can hold up to FIFO_MAX_ENTRIES entries, where each entry is the size of the **mma9559_afe_data_t** data type.

```
FIFO_STRUCT(FIFO_MAX_ENTRIES, sizeof(mma9559_afe_data_t)) test_fifo;
```

An alternate method of creating a FIFO is to create a data type for the required FIFO configuration and instantiate one or more FIFO variables from the data type with the same configuration. In the following example, the fifo_20_t data type is created and used to create fifo_1 and fifo_2 variables:

Example 2-9.

```
typedef FIFO_STRUCT(10, 2) fifo_20_t;
fifo_20_t fifo1, fifo2;
```

2.5.2 Initialize FIFO

Once the FIFO data structure has been created through the definition of a FIFO variable, the space for the FIFO structure has been reserved, but the contents of the FIFO structure must be initialized before it is used. Initialization of a FIFO occurs at runtime and performs the following steps:

- Sets the events that are associated with the FIFO, and reserves them in the **mma9559_vars events_fifos** field. It also clears the associated events, in case any events were already signaled because the FIFO is now empty and contains no data.
- Sets the maximum number of data entries that the FIFO can hold and the size of each data entry.

Both values must be greater than 0 and less than 256. The product of the maximum number of entries and the size of each entry must be less than or equal to the size that was reserved when the FIFO variables was created. (Normally, the values are the same as when the FIFO variable was created/instantiated.)

- Resets the internal variables, so that the FIFO is empty.

From Example 2-9 on page 25, fifo1 may be initialized with the following code to set the FIFO size to the same values that were used in the FIFO data type (maximum entries of 10, each entry is 2 bytes). The following example shows how to associate fifo1 with the EVENT_FIFO event value:

Example 2-10.

```
ret = mma9559_fifo_init(
       (mma9559_fifo_t*)&fifo1,     // pointer to the FIFO structure
       EVENT_BITFIELD(EVENT_FIFO),  // bitfield of associated events
       2,                           // maximum number of data entries
       sizeof(mma9559_afe_data_t)   // number of bytes in each data entry
     );
```

The return value from **`mma9559_fifo_init()`** is 0, if the initialization is successful. If the initialization fails, the value is -1. If the values for the maximum number of entries or the bytes per entry are invalid (0 or greater than 255), the initialization fails. The buffer size (the product of the maximum number of entries and the bytes per entry) is not checked to see if it exceeds the space reserved during the FIFO instantiation.

2.5.3 Push data onto the FIFO

Data can be stored in a FIFO after it has been successfully initialized. Data is always stored as an integer number of data entries and is copied (byte by byte) into the FIFO from the supplied data pointer. Since the data entry size was already set when the FIFO was configured, all that must be provided is the number of data entries to be stored.

Usually data is pushed onto the FIFO one at a time, but the writing of multiple entries also is supported. If the FIFO does not have enough empty data entries to store the data, the push operation is limited to the amount of free space in the FIFO.

The **`mma9559_fifo_push()`** function returns the number of entries that were actually pushed onto the FIFO, allowing user code to check that the requested data was stored correctly. If the number of entries stored does not match the requested number of entries to push, the FIFO overflows and the associated event bits are set in the **`mma9559_vars events_missed`** field.

If the FIFO has not been successfully initialized, the return value is always 0. No data is stored in the FIFO.

Example 2-11.

```
ret = mma9559_fifo_push(
          (mma9559_fifo_t*)&fifo1,   // pointer to the FIFO structure
          (uint8*)raw_data.data,     // pointer to the first byte of the first entry
          1                          // the number of data entries to store
     );
```

Since the FIFO push operation is performed within the firmware with interrupts disabled, it is an atomic transaction. This ensures that multiple asynchronous processes can safely push data into the same FIFO without any risk of interruption or corruption.

2.5.4 Pop data off the FIFO

Data can be read from a FIFO after it has been successfully initialized. Data is always read as an integer number of data entries and is copied (byte by byte) from the FIFO to the supplied data pointer. Since the data entry size is already set when the FIFO was configured, all that is needed is the number of data entries to be fetched.

Data can be read as single or multiple entries. If the FIFO does not contain enough valid data entries to store the data, the pop operation is limited to the number of non-empty entries in the FIFO.

The **`mma9559_fifo_pop()`** function returns the number of entries that were actually popped off the FIFO, allowing user code to check that the requested data was retrieved correctly. If the number of entries read does not match the requested number of entries to pop, the FIFO underflows.

If the FIFO has not been successfully initialized, the return value is always 0 and no data is read from the FIFO.

Example 2-12.

```
ret = mma9559_fifo_pop(
          (mma9559_fifo_t*)&fifo1,// pointer to the FIFO structure
          (uint8*)raw_data.data,  // pointer to the start of the destination buffer
          1                       // the number of data entries to retrieve
     );
```

Since the FIFO pop operation is performed within the firmware with interrupts disabled, it is an atomic transaction. This ensures that multiple asynchronous processes can safely pop data off the same FIFO without any risk of interruption or corruption.

2.5.5 Reset the FIFO

A FIFO can be returned to an empty state, discarding its contents and resetting the associated events, using the **`mma9559_fifo_reset()`** function. This is useful for quickly discarding the contents of a FIFO without having to read all of the data entries:

Example 2-13.

```
ret = mma9559_fifo_reset(
          (mma9559_fifo_t*)&fifo1    // pointer to the FIFO structure to reset
     );
```

2.5.6 Other FIFO functions

The following additional FIFO functions are provided by the firmware:

- **`mma9559_fifo_entries_used()`**: Returns the number of data entries currently used in the FIFO.
- **`mma9559_fifo_entries_free()`**: Returns the number of data entries currently unused in the FIFO.

2.6 Power

The MMA9559L device supports four major power modes, listed below from highest to lowest:

- System clock running at full speed (8 MHz) with the CPU running (RunFastClock)
- System clock running at full speed (8 MHz) with the CPU stopped (StopFastClock)
- System clock running at slow speed (62.5 kHz) with the CPU stopped (StopSlowClock)
- System clock and CPU stopped (StopNoClock)

Ideally, the device would always run in the lowest power mode (StopNoClock), but that is not feasible because of interactions with the hardware. For instance:

- In order for the frame interval counter to run and wake the device up periodically, the system clock must be running in either Fast or Slow Clock rates, but cannot be stopped in the No Clock mode.
- The AFE ADC requires the system clock to be running at full speed in order for the conversion to complete. Once the ADC conversion is complete, the clock rate may be reduced.

In order to accommodate dynamic changes of the stop mode clock rate, the **`mma9559_idle_t`** structure is used. It contains two 32-bit fields:

- **`use_stop_fc`**
- **`use_stop_sc`**

Whenever the device has no further CPU processing to perform, the Freescale firmware can put the CPU into Stop mode until the next interrupt occurs. However, the appropriate clock speed mode must be used. The **`mma9559_idle()`** function selects the clock speed based on the two fields in the **`mma9559_idle_t`** structure:

- If **`use_stop_fc`** is not zero, then use StopFastClock mode,
- Else if **`use_stop_sc`** is not zero, then use StopSlowClock mode,
- Else, use StopNoClock mode.

The 32 bits in each field are split into 16 bits that are reserved for use by Freescale firmware and 16 bits that user code can allocate to different activities that are impacted by the clock rate so that the activities can each independently control the selection of the Stop-mode clock speed.

The Freescale firmware already uses two of the reserved bits:

- IDLE_BITS_CKOSC is used by the frame interval counter to prevent the use of StopNoClock when the frame interval counter is being used.

- IDLE_BITS_AFE is used by the AFE to force the use of StopFastClock when the ADC is performing a conversion.

The **`use_stop_fc`** and **`use_stop_sc`** fields may be modified inside the **`user_exception_handler()`** function. While these functions can be read anywhere, any code that modifies the functions should only be executed when interrupts are disabled.

This can be achieved in three ways:

- Within the **`user_exception_handler()`** or **`user_trap_handler()`** functions,
- When interrupts are disabled using the **`interrupts_disable()`** function,
- Using the **`mma9559_idle_use_stop_config()`** function.

Chapter 3 User Code Example

The functionality of the device is determined by the user firmware. This section shows sample code that can be used as a template for creating that firmware.

The following code usually would be contained in a single file. The code is split into four sections:

- Header
- User exception handler
- User trap handler
- User main

Each of the sections is required, except the user trap handler, which is optional and usually not needed.

3.1 Header

The user code should include the **`mma9559.h`** header file, instantiate the key global variables, assign event bits, and set the FOPT value that is loaded from address 0x3FFE.

Example 3-1.

```
  #include "derivative.h"      // include peripheral declarations
  #include "mma9559.h"         // include mma9559 firmware functions

// User assignment of event bits (0 to 31)
  #define EVENT_AFE_DATA   (0) // AFE conversion is complete
  #define EVENT_SLAVE_PORT (1) // write to the slave port mailboxes completes

// Set Flash Options to enable boot from Flash
  word flash_opt @0x3FFE = 0x0337;
  //  //FOPT_BF_MASK   |       // This bit is inverted so leave
  //                           // blank for boot from flash.
  //  FOPT_CHECKB_MASK |       // Do not perform Flash CRC
  //  FOPT_PW_MASK     |       // PROTB writable
  //  FOPT_PROTB_MASK  |       // Flash Array NOT protected
  //  FOPT_SSW_MASK    |       // Security State Writable
  //  FOPT_SSC_MASK;           // Security State = Unsecured

// Instantiate global variables

// Use a pointer to access the MMA9559 firmware variables
  mma9559_vars_t *mma9559_vars_ptr;
```

```
// The following variables are useful if working with the slave mailboxes, to
// record the mailbox read and write status bits before they are cleared in the
// user exception handler
   vuint32 sp_reads;
   vuint32 sp_writes;
```

3.2 User exception handler

The user firmware normally contains a single user exception handler function that is similar to Equation 3-2 on page 33. This example lists all of the exception vectors that are specific to the MMA955xL device.

For additional information on the exception sources, see the *MMA955xL Intelligent, Motion-Sensing Hardware Reference Manual* (MMA955xLRM). (See "References" on page 10.).

The MMA9559L firmware handles the exception, masks the interrupts, and saves the volatile registers. Then, it switches to User mode (with interrupts still masked) and calls the **`user_exception_handler()`**.

The **`user_exception_handler()`** code is modified/written by the user to perform the following functions:

- Clear the source, or mask—the interrupt that caused the exception.
- Optionally, perform processing based on the interrupt.

  This should be short, to limit the interrupt latency.
- Do either of the following:
  - If the user wants to use the scheduling capabilities (described in "Events and scheduling" on page 18): Signal events by setting a return value that causes the execution of code in the main function loop.
  - If the user does not want to signal events: Return 0.

When the **`user_exception_handler()`** function completes, control returns to the MMA9559L firmware. The firmware signals the events based on the return value from the **`user_exception_handler()`** function, restores the volatile registers, and restores the Program Counter and Status Register to their previous states before the exception occurred.

NOTE

Two of the interrupts sources operate differently from the other interrupt sources:

- The **`VectorNumber_Vconversion_complete`** vector occurs on the completion of an AFE conversion, and uniquely clears the interrupt source without requiring the **`user_exception_handler()`** function to clear it.

  The **`user_exception_handler()`** function may still perform processing or signal events when this exception occurs, but it does not need to clear the AFE conversion complete interrupt source.

- The INT / IRQ pin is a non-maskable interrupt. In order to prevent it from preempting already-running exception processing that exceeds the supervisor stack space, the pin is handled by a thin handler.

  The handler disables the IRQ interrupt and signals a Level-5 software interrupt that is maskable. The user trap handler has to acknowledge/clear the software interrupt and re-enable the INT / IRQ pin interrupt.

Example 3-2.

```
// User Exception Handler function declaration
  __declspec(register_abi) events_t user_exception_handler(
       int      vector);

// User Exception Handler function definition example
// This function is called in User mode with interrupts disabled/masked
  __declspec(register_abi) events_t user_exception_handler(
    int      vector)               // exception vector number
  {
    events_t events = 0;         // Events to be signaled (defaults to none)
    switch (vector) {

    case VectorNumber_VL5swi:    // Proxy for the INT pin (IRQ) external interrupt
         INTC_CFRC    = 0x3A;      // Clear level 5 Software Interrupt (first)
         IRQSC_IRQACK = 1;         // Clear interrupt source
         IRQSC_IRQIE  = 1;         // Re-enable IRQ interrupt
         break;

    case VectorNumber_Vtpm1ovf:    // TPM Overflow
         TPMSC_TOF = 0;            // Clear interrupt source
         break;

    case VectorNumber_Vtpm1ch0:    // TPM Channel 0
         TPMC0SC_CH0F = 0;         // Clear interrupt source
         break;

    case VectorNumber_Vtpm1ch1:    // TPM Channel 1
         TPMC1SC_CH1F = 0;         // Clear interrupt source
         break;

    case VectorNumber_Vmtim:       // MTIM (Modulo Timer) Overflow
         MTIM16SC_TOF = 0;         // Clear interrupt source
         break;

    case VectorNumber_Vpdb_a:      // PDB_A
         PDB_SCR_SA = 1;           // Clear interrupt source
         break;

    case VectorNumber_Vpdb_b:      // PDB_B
         PDB_SCR_SB = 1;           // Clear interrupt source
         break;
```

```
    case VectorNumber_Vsp_wake:    // Slave mailbox
         // Save the mailbox read and write status bits if needed for later
         sp_writes = *((vuint32*)&SP_WSTS0);
         sp_reads  = *((vuint32*)&SP_RSTS0);
         SP_SCR_ACTIVE_CSR = 1;    // Clear slave port read and write status bits
         // Signal an event to trigger processing in the main execution loop
         events = EVENT_BITFIELD(EVENT_SLAVE_PORT);
         break;

    case VectorNumber_Vstart_of_frame:  // Start of Frame
         FCSR_SF = 1;                   // Clear interrupt source
         mma9559_afe_conversion_start();// Start a new AFE conversion
         break;

    case VectorNumber_Vconversion_complete:// AFE conversion complete
         // Interrupt source already cleared in the Freescale exception handler
         // Signal an event to enable processing in the main execution loop
         events = EVENT_BITFIELD(EVENT_AFE_DATA);
         break;

    case VectorNumber_Vmaster_i2c:         // I2C Master
         IICS_IICIF = 1;                   // Clear interrupt source
         break;

    default:
         break;
    }

    return events;              // signal events to run in the main execution loop
}
```

3.3 User trap handler

Although most firmware developers do not need to use these capabilities, the user firmware may optionally create up to four user trap handlers. The trap handlers can be called by any of the supplied functions:

- **`mma9559_user_trap0()`**
- **`mma9559_user_trap1()`**
- **`mma9559_user_trap2()`**
- **`mma9559_user_trap3()`**

Alternatively, the trap call can be issued directly, using the following assembly trap instructions:

- **`asm { trap #TRAP_USER_0 }`**
- **`asm { trap #TRAP_USER_1 }`**
- **`asm { trap #TRAP_USER_2 }`**
- **`asm { trap #TRAP_USER_3 }`**

When issuing a trap call directly with an assembly instruction, the user code is responsible for setting the required values in the registers.

The MMA9559L firmware handles the trap exception, masks the interrupts, switches to User mode (with interrupts still masked), and calls the appropriate **`user_trap_handler()`**.

When the **`user_trap_handler()`** function completes, control returns to the MMA9559L firmware. The firmware leaves the return value in the D0 register and restores the Program Counter and Status Register to their previous state before the trap call. The firmware does not save and restore the volatile registers because the trap call is treated as a function call. The compiler takes care of the values in the volatile registers in the calling function.

A trap handler normally appears similar to the following example. The parameters are the values in the d0, d1, d2, a0 and a1 registers when the trap call is issued. The use of the parameters is up to the user code.

Example 3-3.

```
// User Trap Handler function declaration
   __declspec(register_abi) int user_trap_handler(
     int    d0,
     int    d1,
     int    d2,
     void   *a0,
     void   *a1);

// User trap Handler function definition example
   __declspec(register_abi) int user_trap_handler(
     int    d0,                    // first integer parameter
     int    d1,                    // second integer parameter
     int   d2,                     // third integer parameter
     void *a0,                     // first pointer parameter
     void *a1)                     // second pointer parameter
   {
     switch (d0) {
          case 0:
          ....
     }
     return 0;
   }
```

3.4 User main

The **`user main()`** function is called from the startup code. It should initialize all key variables and enter an infinite loop, waiting for events and processing them when they occur.

Events are signaled in the User Exception Handler. The exception wakes up from the **`mma9559_idle()`** function and keeps processing until all of the signaled events have been cleared. At that point, the function again calls the **`mma9559_idle()`** function to enter a low-power sleep.

Example 3-4.

```
void main(void)
{
  mma9559_afe_data_t afe_data;
  events_t events;
  int event;
  int frame_ctr = 0;

  // Get the address of the mma9559 variables structure
  mma9559_vars_ptr = mma9559_vars_addr_get();

  // Register the exception handler and any trap handlers
  mma9559_vars_ptr->user_exception_handler = user_exception_handler;
  mma9559_vars_ptr->user_trap_handler[0] = user_trap_handler;

  // Setup the slave port mailboxes to generate an interrupt when written to
  SP_SCR = (SP_SCR & (SP_SCR_EN_MASK | SP_SCR_PS_MASK)) |
           SP_SCR_ACTIVE_CSR_MASK |
           SP_SCR_STOP_EN_MASK |
           SP_SCR_WIE_MASK;

  // Configure the AFE and Frame Interval Counter
  mma9559_afe_csr_set((afe_csr_options_t)(
           AFE_CSR_GRANGE_8G   |          // Use AFE 8 g mode
           AFE_CSR_C4MODE_NONE |          // Do not use the 4th ADC channel
           AFE_CSR_CMODE_16BIT ) );       // Use 16 bit ADC conversion
  mma9559_framerate_set(FRAMERATE_POINT2HZ);// Set the frame rate to 1/5 Hz

  // Loop forever processing events
  while (1) {
    events = mma9559_vars_ptr->events;
    if (events) {
      // select the next event to process
      event = mma9559_events_find_first(events); // priority scheduling
      switch(event) {

      case EVENT_AFE_DATA:                       // AFE Conversion Complete
        frame_ctr++;
        mma9559_afe_trimmed_sensor_data_get(afe_data.data);
        ....                                     // add user code here
        break;
```

```
        case EVENT_SLAVE_PORT:                      // Slave Port Maibox write
          if (sp_writes & 0x01)
            ....                                    // add user code here
          break;
        }
        mma9559_events_set_clear(0, EVENT_BITFIELD(event));// clear event
      }
      else
        mma9559_idle();       // if there are no outstanding events, then go to sleep
    };
  }
```

Chapter 4 Functional Details

This chapter provides the details for the following functional categories and special topics:

- "Memory and CPU usage"
- "Hardware support"
- "Events and scheduling"
- "FIFOs"
- "Other functions"
- "IIR filter"

4.1 Memory and CPU usage

This section documents the resources used by each of the MMA9559L Freescale firmware functions in terms of Supervisor Stack Space (RAM) and CPU (cycles and time).

In examining the firmware's use of memory and the CPU, there are three scenarios to consider:

- When in a Freescale firmware function, all interrupts are disabled except for the INT / IRQ interrupt. The supervisor stack usage for each of the Freescale functions is shown in the "Supervisor stack usage" on page 13.

  The total stack usage could also include an additional 12 bytes for the IRQ / INT interrupt, as the worst-case usage in this scenario is **`mma9559_device_get_info()`** (100 bytes) plus the IRQ handler (12 bytes) giving a total of 112 bytes supervisor stack usage.
- Waking up from idle sleep mode to handle an exception and then getting an IRQ / INT exception.

  The **`mma9559_idle()`** function uses 8 bytes of stack space, the Freescale portion of the exception handler uses 36 bytes, and the INT / IRQ handler uses 12 bytes, for a total of 56 bytes. Additional stack space will be used by functions called in the **`user_exception_handler()`**.

  There are 128 bytes available and 56 bytes already have been used, with 72 bytes of supervisor stack space remaining. Since the user exception handler runs on the user stack, it is only necessary to count Freescale firmware functions.

  This enables any Freescale function except **`mma9559_device_info_get()`** to be safely called from within the exception handler.
- Being in a user trap function and then getting an IRQ / INT exception.

  A user trap function consumes 20 bytes of supervisor stack space and the INT / IRQ exception handler takes 12 bytes, for a total of 32 bytes that leaves 96 bytes available for calling Freescale firmware functions or other user trap calls.

  This enables any Freescale function except **`mma9559_device_info_get()`** to be safely called from within the user trap handler. If the **`user_trap_handler()`** calls other **`user_trap_handler()`** functions or is recursive, an additional 20 bytes of stack space is used for each instance.

4.1.1 Supervisor stack

As described in "Memory and CPU usage" on page 12, the last section of RAM is reserved for the supervisor stack that is used whenever the firmware is running.

The **`user_exception_handler()`** and the **`user_trap_handler()`** functions are called from the Freescale firmware, but are run in User mode, using the user stack space rather than using the supervisor stack space. Four other firmware functions do not use any supervisor stack space, because they are in-line functions that are expanded in the customer code and do not issue any trap call. Those functions are:

- **`mma9559_events_find_first()`**
- **`mma9559_events_find_next()`**
- **`mma9559_fifo_entries_free()`**
- **`mma9559_fifo_entries_used()`**

Since the MMA9559L clocks at 8 MHz when the CPU is running, the CPU time needed for the remaining functions can be estimated by dividing the CPU usage in cycles by eight. The following table gives the memory space, CPU cycles, and time required for firmware functions.

Table 4-1. Firmware's use of RAM and CPU

Function	Supervisor stack usage (bytes)	CPU cycles (approximately)[1]	Time (µs)[2]
Exception Handler	36	115	14
mma9559_afe_conversion_start()	24	77	10
mma9559_afe_csr_get()	24	83	10
mma9559_afe_csr_set()	40	217	27
mma9559_afe_offsets_get()	24	78	10
mma9559_afe_offsets_set()	24	78	10
mma9559_afe_interrupt_clear()	24	80	10
mma9559_afe_raw_sensor_data_get()	24	88	11
mma9559_afe_raw_sensor_data_trim()	52	210	26
mma9559_afe_trimmed_sensor_data_get()	52	218	27
mma9559_boot_options_set()	24	79	10
mma9559_device_info_get()	100	718	90
mma9559_events_find_first()	0	8	1
mma9559_events_find_next()	0	65	8
mma9559_events_set_clear()	24	103	13
mma9559_fifo_entries_free()	0	39	5
mma9559_fifo_entries_used()	0	16	2

Table 4-1. Firmware's use of RAM and CPU (Continued)

Function	Supervisor stack usage (bytes)	CPU cycles (approximately)[1]	Time (µs)[2]
mma9559_fifo_init()	24	106	13
mma9559_fifo_pop()[3]	40	145 + E * (12 + 6 * B)	—
mma9559_fifo_push()[3]	40	134 + E * (22 + 6 * B)	—
mma9559_fifo_reset()	24	103	13
mma9559_framerate_set()	28	105	13
mma9559_idle()	8	34	4
mma9559_idle_use_stop_config()	24	87	11
mma9559_iir_filter() (order O)[4]	40	143 + 9 * O	—
mma9559_interrupts_disable()	12	44	6
mma9559_interrupts_restore()	12	46	6
mma9559_rom_command (RMF_DEV_INFO, 0)[5]	68	515	64
mma9559_user_trap0()	20	101	13
mma9559_vars_addr_get()	24	76	10

1 E = The number of entries that are being transferred.
B = The size of each entry, in bytes.
O = The order of the filter.

2 The CPU time may be calculated for the blank cells, by calculating the number of CPU cycles and dividing by 8.

3 The CPU usage of the **`mma9559_fifo_pop()`** and **`mma9559_fifo_push()`** functions depends on the number of entries that are being transferred (E) and the size of each entry in bytes (B). Once the number of cycles has been calculated, the CPU time can be determined by dividing by 8.

4 The CPU usage of the **`mma9559_iir_filter()`** function depends on the order (O) of the filter. Once the number of cycles has been calculated, the CPU time can be determined by dividing by 8.

5 The CPU time for the **`mma9559_rom_command()`** function depends on which of its ROM functions are being executed. The stack usage is fixed, however, because the ROM functions do not use stack space. The ROM Device Info function was used for the measurement of the CPU cycles and time in Table 4-1.

4.2 Hardware support

Enables user code to access key hardware functionality of the MMA955xL device.

The MMA9559L Freescale firmware provides a set of functions that enable access to key areas of functionality in the MMA955xL hardware. These areas include:

- Frame Interval Counter (FIC): Creates a periodic interval timer
- Analog Front End (AFE): Configures and reads the sensor hardware values

4.2.1 Macros

4.2.1.1 #define NUM_SENSOR_AXIS 5

Specifies the total number of axes/sensors in the (AFE) Analog Front End. These include:

- Accelerometer X, Y, and Z
- Temperature
- External input

4.2.2 Enumerations

4.2.2.1 enum afe_csr_options_t

This enumeration configures the AFE (Analog Front End) through the AFE CSR (Control and Status Register).

Since the AFE_CSR is not directly accessible in User mode, it can be accessed using the Freescale **`mma9559_afe_csr_set()`** and **`mma9559_afe_csr_get()`** functions.

The AFE_CSR controls three parameters of the AFE operation:

- G range selection: The AFE can operate in +/-2g, +/- 4g or +/-8g range.
- Fourth Conversion: The X, Y and Z accelerometer channels are measured using three out of four of the ADC channels. The fourth channel can be disabled or used to measure the temperature or external ADC input.
- Conversion mode: The ADC can perform a 10, 12, 14, or 16-bit conversion. A higher bit conversion gives more resolution in the result, but takes longer to complete, such that more power is consumed.

  The significance of the most-significant bit remains the same in all conversion modes. In other words, if the conversion mode is less than 16 bits, the least-significant bits of the raw sensor value is 0.

The AFE_CSR setting value is normally constructed by OR-ing one entry from each of the three sections and casting it to the correct type, as shown in the following example.

Example 4-1.

```
\\ Configure the AFE Control and Status Register
  mma9559_afe_csr_set((afe_csr_options_t)(
            AFE_CSR_GRANGE_2G   |          // Use +/- 2 g range
            AFE_CSR_C4MODE_TEMP |          // Measure X, Y, Z and Temperature
            AFE_CSR_CMODE_10BIT            // Use 10 bit conversion
       );
```

See also:

- **void mma9559_afe_csr_set(afe_csr_options_t options)**
- **void mma9559_afe_csr_set(afe_csr_options_t options)**

Table 4-2. enum afe_csr_options_t enumerators

Enumerator	Description
AFE_CSR_GRANGE_8G	Set the AFE to the +/-8g range, so each LSB is 0.244 mg
AFE_CSR_GRANGE_4G	Set the AFE to the +/-4g range, so each LSB is 0.122 mg
AFE_CSR_GRANGE_2G	Set the AFE to the +/-2g range, so each LSB is 0.061 mg
AFE_CSR_C4MODE_NONE	Do not measure anything with the ADC's fourth channel
AFE_CSR_C4MODE_TEMP	Use the ADC's fourth channel to measure the temperature sensor
AFE_CSR_C4MODE_EXT	Use the ADC's fourth channel to measure the external inputs
AFE_CSR_CMODE_10BIT	Perform a 10-bit ADC conversion so the six LSBs are 0
AFE_CSR_CMODE_12BIT	Perform a 12-bit ADC conversion so the four LSBs are 0
AFE_CSR_CMODE_14BIT	Perform a 14-bit ADC conversion so the two LSBs are 0
AFE_CSR_CMODE_16BIT	Perform a 16-bit ADC conversion using all 16 data bits

4.2.2.2 enum framerate_t

This enumeration configures the FIC (Frame Interval Counter) frequency through the CK_OSCTRL register.

The frame interval counter is the primary mechanism for waking the device up periodically to take some action such as reading the AFE values. The frame interval counter register is accessible in User mode, but a function is provided to set the frame rate for two reasons:

- The upper three bits of the CK_OSCTRL register control other attributes of the oscillator, and should be preserved when the frame rate is changed.
- In order for the frame interval counter to run, the lowest power idle mode that can be used is Stop SC (slow clock). This is ensured by setting the IDLE_BITS_CKOSC bit in the idle **`mma9559_idle_t use_stop_sc`** field. While this field can be accessed by user code, it should only be modified while interrupts are disabled, either in the exception or trap handlers, or when interrupts have been disabled.

If the frame rate is set to FRAMERATE_NONE, the IDLE_BITS_CKOSC bit in the idle **`mma9559_idle_t use_stop_sc`** field is cleared to enable idle mode to drop to the lowest power Stop NC (no clock) mode. If the frame rate is set to any other valid value, the IDLE_BITS_CKOSC bit in the idle **`mma9559_idle_t use_stop_sc`** field is set to prevent the idle mode from dropping below the Stop SC (slow clock) mode.

If the device enters the Stop NC state, the frame interval counter does not run and cannot wake the device at the start of each frame.

See also:

- **framerate_t mma9559_framerate_set(framerate_t rate)**

Table 4-3. enum framerate_t enumerators

Enumerator	Description
FRAMERATE_NONE	The FIC is disabled and does NOT wake up the MMA9559L periodically.
FRAMERATE_3906HZ	Configures the FIC to run at 3906 Hz
FRAMERATE_1953HZ	Configures the FIC to run at 1953 Hz
FRAMERATE_977HZ	Configures the FIC to run at 977 Hz
FRAMERATE_488HZ	Configures the FIC to run at 488 Hz
FRAMERATE_244HZ	Configures the FIC to run at 244 Hz
FRAMERATE_122HZ	Configures the FIC to run at 122 Hz
FRAMERATE_61HZ	Configures the FIC to run at 61 Hz
FRAMERATE_30HZ	Configures the FIC to run at 30.5 Hz
FRAMERATE_15HZ	Configures the FIC to run at 15.3 Hz
FRAMERATE_8HZ	Configures the FIC to run at 7.6 Hz
FRAMERATE_4HZ	Configures the FIC to run at 3.8 Hz

Table 4-3. enum framerate_t enumerators

Enumerator	Description
FRAMERATE_2HZ	Configures the FIC to run at 1.9 Hz
FRAMERATE_1HZ	Configures the FIC to run at 0.95 Hz
FRAMERATE_POINT5HZ	Configures the FIC to run at 0.48 Hz
FRAMERATE_POINT2HZ	Configures the FIC to run at 0.24 Hz

4.2.3 Data structure

4.2.3.1 mma9559_afe_data_t

The **`mma9559_afe_data_t`** structure is used to contain a set of raw or trimmed AFE sensor data taken during a single AFE sample. A 16-bit, signed value is stored for each axis/sensor, where the number of axes/sensors in each sample is defined in the **`NUM_SENSOR_AXIS`** macro.

See also:

- **framerate_t mma9559_framerate_set(framerate_t rate)**

Field

int16 data [5]: Raw or trimmed value for each axis/sensor

4.2.4 Functions

4.2.4.1 framerate_t mma9559_framerate_set(framerate_t rate)

This function configures the FIC (Frame Interval Counter) with the supplied frequency. If the frame rate is invalid, it is ignored and the frame rate is not adjusted.

Example 4-2.

```
framerate_t framerate;
framerate = mma9559_framerate_set(FRAMERATE_488HZ);
```

If the rate value is FRAMERATE_NONE, the IDLE_BITS_CKOSC bit is cleared in the **`mma9559_idle_t use_stop_sc`** field, enabling the **`mma9559_idle()`** function to use the StopNC stop mode (unless some other setting of the **`idle_config_t`** structure prevents it).

If the rate value is any other valid rate, the IDLE_BITS_CKOSC bit is set in the **`mma9559_idle_t use_stop_sc`** field, preventing the **`mma9559_idle()`** function from using the StopNC mode, and stopping the FIC from running.

NOTE

At the highest supported frame rate of 3.906 kHz, the 16-bit and 14-bit AFE conversion cannot be completed within the frame interval, as a result only an AFE conversion length of 12 bits or less should be used.

See also:

- **enum framerate_t**
- **“Frame interval counter” on page 14**
- **“User main” on page 37**

Return

The resulting frame rate value is the frame interval counter. This should be the same as the requested value, unless the requested value was invalid, in which case the current frame rate setting in the frame interval counter is returned.

Parameter

rate: The frame rate setting for the frame interval counter frequency.

4.2.4.2 void mma9559_afe_conversion_start(void)

This function triggers the AFE to start an ADC conversion cycle. The duration of the conversion depends on the AFE_CSR configuration. When the conversion completes, the **`VectorNumber_Vconversion_complete`** exception occurs and can be used by the **`user_exception_handler`** to start subsequent processing.

This function can be called from anywhere in user code, but is usually called from within the **`user_exception_handler()`** function, when the Start of Frame interrupt is serviced, to minimize the jitter on the AFE sampling:

Example 4-3.

```
__declspec(register_abi) events_t user_exception_handler(
   int  vector)                              // exception vector number
{
   ...
   case VectorNumber_Vstart_of_frame:        // Start of Frame
      FCSR_SF = 1;                           // Clear interrupt source
      mma9559_afe_conversion_start();        // Start a new AFE conversion
      break;
   ...
}
```

The function also sets the IDLE_BITS_AFE in the **`mma9559_idle_t use_stop_fc`** field so that the **`mma9559_idle()`** function uses StopFC (fast clock) during any idle time. The ADC requires the system clock to run at full speed during the AFE conversion.

See also:

- **“User exception handler” on page 32**

4.2.4.3 void mma9559_afe_interrupt_clear(void)

This function resets the AFE conversion complete bit that is set on the completion of the ADC conversion. It also clears the IDLE_BITS_AFE in the **`mma9559_idle_t use_stop_fc`**. As a result, the **`mma9559_idle()`** function is no longer forced to use StopFC (fast clock) during any idle time.

NOTE

Since the Freescale exception handler already executes this functionality whenever an AFE conversion complete exception occurs, users do not normally need to call this function unless they are operating with interrupts disabled.

4.2.4.4 void mma9559_afe_raw_sensor_data_get(int16 *data_ptr)

This function gets the raw (semi-trimmed) data from the AFE sensors.

Normally, the trimmed AFE sensor values are read using the **`mma9559_afe_trimmed_sensor_data_get()`** function which reads the hardware registers and applies the trim calculations in a single function call.

This function copies the semi-trimmed sensor values directly from the AFE hardware registers into the supplied data structure. The raw sensor values can then be trimmed using the **`mma9559_afe_raw_sensor_data_trim()`** function. This can be useful in saving power when basic operations can be performed on the sensor data (such as decimation), prior to applying the trim calculations.

Example 4-4.

```
mma9559_afe_data_t raw_afe_data;
...
mma9559_afe_raw_sensor_data_get(raw_afe_data.data);
```

The raw sensor values are stored in the array of 16-bit signed integers in the order:

- Accelerometer X axis
- Accelerometer Y axis
- Accelerometer Z axis
- Temperature
- External input

The Temperature and External input values do not change if the AFE_CSR is not configured to measure them using the fourth ADC channel.

NOTE

The user must ensure that the supplied data structure is large enough to hold the values from all of the axis/sensors. The number of axes/sensors is set in the NUM_SENSOR_AXIS definition.

See also:

- **void mma9559_afe_trimmed_sensor_data_get(int16 *trim_ptr)**
- **void mma9559_afe_raw_sensor_data_trim(int16 *trim_ptr, int16 *data_ptr)**
- **mma9559_afe_data_t**
- “Analog Front End (AFE)” on page 15

Parameter

data ptr: Address to store the raw AFE sensor data values.

4.2.4.5 void mma9559_afe_raw_sensor_data_trim(int16 *trim_ptr, int16 *data_ptr)

This function trims the supplied raw AFE sensor data to produce trimmed AFE sensor values.

Normally the trimmed AFE sensor values are read using the **`mma9559_afe_trimmed_sensor_data_get()`** function which reads the hardware registers and applies the trim calculations in a single function call.

This function trims previously read semi-trimmed sensor values, that were read using the **`mma9559_afe_raw_sensor_data_get()`** function, and optionally pre-processed. The trim processing is identical to the **`mma9559_afe_trimmed_sensor_data_get()`** function.

Example 4-5.

```
mma9559_afe_data_t raw_afe_data;
mma9559_afe_data_t trimmed_afe_data;
...
mma9559_afe_raw_sensor_data_trim(trimmed_afe_data.data, raw_afe_data.data);
```

NOTE

The user must ensure that the supplied data structure is large enough to hold the values from all of the axis/sensors. The number of axis/sensors is set in the NUM_SENSOR_AXIS definition.

See also:

- **void mma9559_afe_raw_sensor_data_get(int16 *data_ptr)**
- **mma9559_afe_data_t**
- “Analog Front End (AFE)” on page 15

Parameters

- trim_ptr: Address to store the trimmed AFE sensor data values.
- data_ptr: Address of the raw AFE sensor data to be trimmed.

4.2.4.6 void mma9559_afe_trimmed_sensor_data_get(int16 *trim_ptr)

This function reads the data from the AFE sensor hardware registers and applies the trim calculations to provide the accurate values.

The accelerometer sensor values are corrected for gain and offset using device-specific trim values. User specified offsets are also applied to account for offsets caused by board mounting during the assembly process. The user offsets can be set using the **`mma9559_afe_offsets_set()`** function.

Example 4-6.

```
mma9559_afe_data_t trimmed_afe_data;
...
mma9559_afe_trimmed_sensor_data_get(trimmed_afe_data.data);
```

The trimmed sensor values are stored in the array of 16-bit, signed integers in the order:

- Accelerometer X axis
- Accelerometer Y axis
- Accelerometer Z axis
- Temperature
- External input

The temperature and external input values remain unchanged, if the AFE_CSR is not configured to measure them using the fourth ADC channel.

NOTE

The user must ensure that the supplied data structure is large enough to hold the values from all of the axis/sensors. The number of axis/sensors is set in the NUM_SENSOR_AXIS definition.

See also:

- **void mma9559_afe_offsets_set(int16 *data_ptr)**
- **mma9559_afe_data_t**
- “Analog Front End (AFE)” on page 15

Parameter

trim_ptr**:** Address to store the trimmed AFE sensor data values.

4.2.4.7 void mma9559_afe_csr_set(afe_csr_options_t options)

This function configures the AFE (Analog Front End) by setting the AFE CSR (Control and Status Register) contents. The AFE_CSR is not directly accessible in User mode, so this function enables it to be set by the user. The function also preserves some fixed configuration settings.

The AFE_CSR controls three parameters of the AFE operation:

- G range selection: The AFE can operate in +/-2g, +/- 4g or +/-8g range.
- Fourth Conversion: The X, Y, and Z accelerometer channels are measured using three out of the four ADC channels. The fourth ADC channel can be disabled or used to measure the temperature or external input.
- Conversion mode: The ADC can perform a 10, 12, 14, or 16-bit conversion. A higher bit conversion gives more resolution in the result, but takes longer to complete, and consumes more power.

  The significance of the most-significant bit remains the same in all conversion modes. If the conversion mode is less than 16 bits, the least-significant bits of the raw sensor value is 0.

The AFE_CSR setting value is normally constructed by OR-ing one entry from each of the three sections and casting it to the correct type, as shown in the following example.

Example 4-7.

```
// Configure the AFE Control and Status Register
  mma9559_afe_csr_set((afe_csr_options_t)(
            AFE_CSR_GRANGE_2G   | // Use +/- 2 g range
            AFE_CSR_C4MODE_TEMP | // Measure X, Y, Z and Temperature
            AFE_CSR_CMODE_10BIT) // Use 10 bit conversion
          );
```

Parameter

options: Configuration option to be loaded into the AFE CSR.

4.2.4.8 afe_csr_options_t mma9559_afe_csr_get(void)

This function reads the configuration bits from the AFE CSR register.

Example 4-8.

```
afe_csr_options_t csr;
csr = mma9559_afe_csr_get();      // Read the current AFE CSR value
```

See also:

- **void mma9559_afe_offsets_set(int16 *data_ptr)**
- **enum afe_csr_options_t**

Return

The current setting of the AFE CSR options.

4.2.4.9 void mma9559_afe_offsets_set(int16 *data_ptr)

This function gets the current AFE accelerometer user offset values.

The **`mma9559_afe_trimmed_sensor_data_get()`** and **`mma9559_afe_raw_sensor_data_trim()`** functions add user-programmable offset values to the trimmed accelerometer readings. These values may be set or read by the user.

The offset values are treated as values in the 8g range (where the hex value 0x4000 equates to 1g), and are adjusted automatically to the appropriate g range when the g range is changed with the **`mma9559_afe_csr_set()`** function. The values can be determined by putting the device into the 8g mode, reading the accelerometer values without the effects of gravity, and negating these values.

Example 4-9.

```
int16 offsets[3] = { 0, 0, 0x4000 };     // X, Y and, Z offsets in 8g mode
mma9559_afe_offsets_set(offsets);
```

The user offset values are reset whenever the device is reset.

See also:

- **void mma9559_afe_trimmed_sensor_data_get(int16 *trim_ptr)**

Parameter

data ptr: Address containing the three accelerometer user offset values.

4.2.4.10 void mma9559_afe_offsets_get(int16 *data_ptr)

This function enables the user to read the current values of the three accelerometer user offset settings. The offset values are in 8g mode, with the hex value of 0x4000 corresponding to an offset of 1g.

Example 4-10.

```
int16 offsets[3];                    // X, Y and Z offsets in 8 g mode
mma9559_afe_offsets_get(offsets);
```

NOTE

The user must ensure that the supplied data structure is large enough to hold the values from all three of the accelerometer axes.

See also:

- **void mma9559_afe_offsets_set(int16 *data_ptr)**

Parameter

data ptr: Address containing the three accelerometer user offset values.

4.3 Events and scheduling

This functional category manages events and task-scheduling.

The MMA9559L firmware creates events and provides a set of functions to manage them and use them to schedule tasks or activities. These tasks or activities include:

- Signaling and clearing up to 32 events
- Selecting events, using priority or round-robin scheduling algorithms
- Providing configurable, power-saving idle control
- Disabling or restoring interrupts for critical sections
- Providing prototypes for user-created exception and trap-handler functions

4.3.1 Macros

4.3.1.1 #define EVENT_BITFIELD(b) ((events_t)(1<<b))

This macro creates an events_t bit field from the event number.

The MMA9559 firmware supports up to 32 events, numbered from 0 to 31. The events_t data type can represent multiple events by using a single bit for each event. This macro provides an easy way to create the events_t bit field from an event number.

4.3.2 Enumerations

4.3.2.1 enum idle_config_t

This value is used by the **mma9559_idle_use_stop_config()** to set how the **idle_bits** bit field is used to modify the idle mode operation.

See also:

- **void mma9559_idle_use_stop_config(idle_config_t config, idle_bits_t bits)**

Enumerators

Table 4-4. enum idle_config_t enumerators

Enumerators	Description
IDLE_USE_STOP_FC_CLEAR	Clear (remove) a bit in the use_stop_fc field of the mma9559_idle_t structure
IDLE_USE_STOP_FC_SET	Set a bit in the **use_stop_fc** field of the **mma9559_idle_t** structure. This forces the idle function to stop using the Stop_FC mode so that the system clock still runs at full speed
IDLE_USE_STOP_SC_CLEAR	Clear (remove) a bit in the **use_stop_sc** field of the **mma9559_idle_t** structure
IDLE_USE_STOP_SC_SET	Set a bit in the **use_stop_sc** field of the **mma9559_idle_t** structure. This prevents the idle function from using the Stop_NC mode, allowing the system clock to still run

4.3.2.2 enum idle_bits_t

This value may be used individually or combined by the **`mma9559_idle_use_stop_config()`** function to set how the bit field is used to modify the idle mode operation. Bits can be individually set to separate tasks, allowing tasks to modify the idle mode configuration without overwriting the configuration set by other tasks or activities.

These bit fields also can be used to directly modify the **`idle_config_t`** fields within the **`user_exception_handler`** function. The fields also can be used whenever interrupts are disabled.

- The first 16 idle bits are reserved for Freescale use and should not be used by user code.
- The second set of 16 bits are available for user code.

See also:

- **void mma9559_idle_use_stop_config(idle_config_t config, idle_bits_t bits)**

Enumerators

Table 4-5. enum idle_bits_t enumerators

Enumerator	Description
IDLE_BITS_CKOSC	Sets the idle configuration for the FIC (Frame Interval Counter)
IDLE_BITS_AFE	Sets the idle configuration for the AFE (Analog Front End)
IDLE_BITS_RSVD_2	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_3	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_4	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_5	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_6	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_7	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_8	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_9	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_10	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_11	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_12	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_13	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_14	Idle configuration bit reserved for Freescale use
IDLE_BITS_RSVD_15	Idle configuration bit reserved for Freescale use
IDLE_BITS_USER_0	User-assignable idle configuration bit
IDLE_BITS_USER_1	User-assignable idle configuration bit

Table 4-5. enum idle_bits_t enumerators

Enumerator	Description
IDLE_BITS_USER_2	User-assignable idle configuration bit
IDLE_BITS_USER_3	User-assignable idle configuration bit
IDLE_BITS_USER_4	User-assignable idle configuration bit
IDLE_BITS_USER_5	User-assignable idle configuration bit
IDLE_BITS_USER_6	User-assignable idle configuration bit
IDLE_BITS_USER_7	User-assignable idle configuration bit
IDLE_BITS_USER_8	User-assignable idle configuration bit
IDLE_BITS_USER_9	User-assignable idle configuration bit
IDLE_BITS_USER_10	User-assignable idle configuration bit
IDLE_BITS_USER_11	User-assignable idle configuration bit
IDLE_BITS_USER_12	User-assignable idle configuration bit
IDLE_BITS_USER_13	User-assignable idle configuration bit
IDLE_BITS_USER_14	User- assignable idle configuration bit
IDLE_BITS_USER_15	User-assignable idle configuration bit

4.3.3 Data structures

This section describes the data structures used in the events and scheduling functional category.

4.3.3.1 mma9559_idle_t

This structure holds the configuration variables that control the operation of the device when the **`mma9559_idle()`** function is called. The use of this structure is described in the **`mma9559_idle()`** function.

The **`use_stop_fc`** and **`use_stop_sc`** fields can be modified inside the **`user_exception_handler()`** function. While these fields can be read anywhere, any code that modifies them should only be executed when interrupts are disabled.

This can be achieved in three ways:

- Within the **`user_exception_handler()`** or **`user_trap_handler()`** functions
- When interrupts are disabled, using the **`interrupts_disable()`** function
- Using the **`mma9559_idle_use_stop_config()`** function

See also:

- **void mma9559_idle(void)**
- **void mma9559_idle_use_stop_config(idle_config_t config, idle_bits_t bits)**

Fields

Table 4-6. mma9559_idle_t fields

Field	Description
int8 stop_cfg	Override the normal idle mode stop configuration, based upon the value of this field: • stop_cfg < 0: do not use any Stop mode • stop_cfg > 0: use the supplied value as the Stop mode • stop_cfg = 0: use the use_stop_fc and use_stop_sc fields
vuint32 use_stop_fc	If this is non-zero then STOP mode must use StopFC (Fast Clock) mode.
vuint32 use_stop_sc	If this is non-zero and use_stop_fc is zero then STOP mode must use StopSC (Slow Clock) mode.

4.3.3.2 mma9559_vars_t

This structure holds all of the variables that are used by the Freescale firmware. Most of these variables can be used by the user firmware to configure and control the operation of the Freescale firmware. User code obtains the address of the data structure using the **`mma9559_vars_addr_get()`** function.

The **`events`** and **`events_missed`** fields can be read anywhere, but—since they can be updated by the Freescale **`exception_handler()`** function—any code that modifies them should only be executed when interrupts are disabled. (These fields are described in "Events" on page 18 and "FIFOs" on page 74.)

Modifying the two fields can be achieved in three ways:

- Within the **`user_exception_handler()`** or **`user_trap_handler()`** functions
- When interrupts are disabled, using the **`interrupts_disable()`** function
- Using the **`mma9559_idle_use_stop_config()`** function

The **`user_exception_handler`** and **`user_trap_handler`** fields can be loaded with the addresses of the respective user functions, if they are present. If no handlers are present, they are set to 0.

See also:

- **void mma9559_idle(void)**
- **mma9559_vars_t* mma9559_vars_addr_get(void)**
- **events_t mma9559_events_set_clear(events_t set_events, events_t clear_events)**

Fields

Table 4-7. mma9559_idle_t fields

Field	Description
volatile events_t events	Currently signaled events. (See "Events and scheduling" on page 56.)
volatile events_t events_missed	Events that were signaled before they had been cleared from the previous time that they had been signaled. (See "Events and scheduling" on page 56.)
events_t events_fifos	Events that are associated with FIFOs should not be modified by user code. (See "FIFOs" on page 74.)
uint16 afe_cm_gain	AFE trim configuration. *Do not modify.*
uint8 afe_fs_shift	AFE trim configuration. *Do not modify.*
mma9559_idle_t idle	Idle stop mode configuration. (See "mma9559_idle_t" on page 60.)
user_exception_handler_t user_exception_handler	Address of the user exception handler.
user_trap_handler_t user_trap_handler[4]	Addresses of the user trap handlers. These are optional, but should be set prior to issuing the associated trap call.

4.3.4 Functions

4.3.4.1 mma9559_vars_t* mma9559_vars_addr_get(void)

The Freescale firmware uses an instance of the **`mma9559_vars_t`** structure to store the variables used for its configuration and control. The operation of the firmware can be modified and/or observed through this control structure.

Since the address of the structure instance may change in future releases of the firmware, this function is provided to enable the user code to find the address of the structure and use it.

NOTE

This function calls the ROM trap function to get device information from the ROM code. This causes both this function call and the ROM trap being on the supervisor stack at the same time—consuming a large part of the supervisor stack. This function should *not* be called from within a user exception handler or a user trap call because that would exceed the supervisor stack space.

Return

Address of the Freescale firmware data structure.

4.3.4.2 events_t mma9559_events_set_clear(events_t set_events, events_t clear_events)

This function updates the **mma9559_vars_t events** and **events_missed** fields with the supplied **set_events** and **clear_events** parameters.

The **mma9559_vars_t events_missed** field is updated to mark any events that were signaled in the **set_events** parameter, were *not* set in the **clear_events** parameter, and are still active in the events field.

The **mma9559_vars_t events** field is updated, setting the events signaled in the **set_events** parameter and marking the events in the **clear_events** parameter. If an event is set in both the **set_events** and the **clear_events** fields, the event is signaled.

Any events that have been associated with FIFOs, using **mma9559_fifo_init()**, are masked, indicating that they can only be signaled and cleared by the FIFO functions and cannot be modified by user code, including this function.

The function performs the operations shown in the following example:

Example 4-11.

```
set_events &= ~(mma9559_vars.events_fifos);
clear_events &= ~(mma9559_vars.events_fifos);
mma9559_vars.events_missed |= set_events & ~clear_events & mma9559_vars.events;
mma9559_vars.events = set_events | (~clear_events & mma9559_vars.events);
```

The **mma9559_vars_t events** field may be modified in an exception handler, which means it must not be modified in User mode. This is because User mode cannot ensure an atomic transaction is not interrupted by an exception, unless it is modified within a critical section (that cannot be interrupted). The **mma9559_events_set_clear()** function runs with interrupts masked, providing a safe way for user code to modify the events field without the risk of being interrupted by an exception and without needing to create a critical section.

Return

The resulting value of the mma9559_vars_t events field

Parameters

Table 4-8. events_t mma9559_events_set_clear parameters

Parameter	Description
set_events	Bits that are set in this parameter are signaled in the mma9559_vars_t events field
clear_events	Bits that are set in this parameter are cleared from the mma9559_vars_t events field

4.3.4.3 int mma9559_events_find_first(events_t events)

This function may be used in conjunction with the Events and Scheduling functionality of the Freescale firmware to provide a basic scheduling capability for a user's execution loop.

The function returns the number of the highest-priority event bit that is set in the **`events`** field. The priority is arranged so that the least-significant bit (LSB) of the events parameter is the highest-priority event—which generates the return value 0. The most-significant bit (MSB) of the events parameter is the lowest-priority event—which generates the return value 31.

Return

The priority number of the highest-priority active task in the **`events`** bit field, from 0 for the LSB to 31 for the MSB. If there are no active priority bits, the value 32 is returned.

Parameter

events: The events bit field to be searched for the first event.

4.3.4.4 int mma9559_events_find_next(events_t events, int current_event)

The purpose of this function is to find the next active event for round-robin scheduling. This function may be used in conjunction with the Events and Scheduling functionality of the Freescale firmware to provide a basic scheduling capability for a user's execution loop.

This function returns the number of the next-highest priority event bit that is set in the events field and a lower priority than the current event. The priority is arranged, such that the least-significant bit (LSB) of the events parameter is the highest-priority event—which generates the return value 0. The most-significant bit (MSB) of the events parameter is the lowest-priority event—which generates the return value 31.

If there are no active priority bits lower than the **`current_event`**, the selection wraps around and the highest-priority active event is returned.

Calling this function with a **`current_event`** parameter of 32 or higher causes the function to return the highest-priority active event.

Return

The priority number of the next-highest priority active task in the events bit field that is lower than the **`current_event`**—from 0 for the LSB to 31 for the MSB. If there are no active priority bits, the value 32 is returned.

Parameters

Table 4-9. int mma9559_events_find_next parameters

Parameter	Description
events	The events bit field to be searched for the next event.
current_event	The currently running priority level (0 to 31).

4.3.4.5 void mma9559_idle_use_stop_config(idle_config_t config, idle_bits_t bits)

This function enables the **mma9559_idle_t use_stop_fc** and **use_stop_sc** fields to be configured from user space.

The **mma9559_idle_t** fields are modified directly by the Freescale exception handler and may also be modified by the **user_exception_handler** function. To avoid modification of these fields from user space or corruption of the fields by an exception during processing, the user code must use a critical section (disabling then enabling interrupts) or call this function.

Example 4-12.

```
// Set a use StopFastClock bit so that idle will use Stop Fast Clock mode
   mma9559_idle_use_stop_config(
      IDLE_USE_STOP_FC_SET,        // Set bit(s) in the Use Stop Fast field
      IDLE_BITS_USER_0);           // Bits to be set
   ...
// When the hardware has finished running clear the bit to allow normal Stop mode operation
   mma9559_idle_use_stop_config(
      IDLE_USE_STOP_FC_CLEAR,      // Clear bit(s) in the Use Stop Fast field
      IDLE_BITS_USER_0);           // Bits to be cleared
```

These fields are used by the **mma9559_idle()** function to determine which Stop mode to use when the device is idle to achieve the lowest possible power.

The Freescale firmware reserves the lower 16 idle_bits and already uses the two least-significant idle_bits:

- IDLE_BITS_CKOSC: Used by the frame interval counter to prevent the use of StopNoClock when the frame interval counter is being used.
- IDLE_BITS_AFE: Used by the AFE to force the use of StopFastClock when the ADC is performing a conversion.

  User code may use any of the upper 16 bits that have the prefix IDLE_BITS_USER_.

See also:

- **void mma9559_idle(void)**
- “Power” on page 28

Parameters

Table 4-10. void mma9559_idle_use_stop_config parameters

Parameter	Description
config	Selects which field and type of modification to make.
bits	Designates the bits to be modified in the selected field.

4.3.4.6 void mma9559_idle(void)

This function executes the idle processing function, to sleep until the next exception.

The **`mma9559_idle()`** function manages the CPU Stop modes to use the lowest possible power. This is done using the Stop mode, based on the configuration set in the **`mma9559_idle_t`** fields and the **`mma9559_vars_t events`** field:

- If the **`mma9559_vars_t events`** field is non-zero, then return.

  This occurs if there are unhandled events that should be processed before entering Stop mode.
- If the **`mma9559_idle_t stop_cfg`** field is negative, then return.

  This enables the use of Stop modes to be disabled for debug purposes.
- If the **`mma9559_idle_t stop_cfg`** field is positive, that value is loaded directly into the STOP_CR register and a stop issued.

  This enables testing of particular Stop configurations but should not generally be used.
- If the **`mma9559_idle_t stop_cfg`** field is 0 and the **`mma9559_idle_t use_stop_fc`** is non-zero, the device uses StopFC (fast clock) mode.
- If the **`mma9559_idle_t stop_cfg`** and **`use_stop_fc`** fields are 0 and the **`mma9559_idle_t use_stop_sc`** field is non-zero, the device uses StopSC (slow clock) mode.
- If the **`mma9559_idle_t stop_cfg`**, **`use_stop_fc`**, and **`use_stop_sc`** fields are all zero, the device uses StopNC (no clock) mode—the lowest power mode.

See also:

- **mma9559_idle_t**
- **mma9559_vars_t**

4.3.4.7 int mma9559_interrupts_disable(void)

This function can be used to create critical sections in user firmware that is not interrupted by exceptions. This implementation supports the use of nested critical sections (accidentally or deliberately) by enabling the user to record the interrupt level/mask interrupts when they are disabled. The interrupts are restored to their original value, when the interrupts are restored.

The **`user_exception_handler()`** and **`user_trap_handler()`** run with the interrupts disabled; therefore, these functions are not required for either of these handlers.

Any exception sources that become active are handled once the interrupt priority mask is returned to 0 by the call to the **`mma9559_interrupts_restore()`** function.

A critical section can be created as shown in the following example.

Example 4-13.

```
int status = mma9559_interrupts_disable(); // start critical section
....
mma9559_interrupts_restore(status);        // restores previous interrupt level/mask
```

If the critical section is never nested, this solution can be simplified by removing the status variable and making the parameter to the **`mma9559_interrupts_restore()`** function zero. See the following example.

Example 4-14.

```
mma9559_interrupts_disable();              // start critical section
....
mma9559_interrupts_restore(0);             // enables interrupts
```

Return

The value of the status register, including the interrupt priority mask before the interrupts were disabled.

See also:

- **void mma9559_interrupts_restore(int status)**
- “Interrupts and critical sections” on page 22

4.3.4.8 void mma9559_interrupts_restore(int status)

This function is used in conjunction with the **`mma9559_interrupts_disable()`** function to enable user code to create critical sections that cannot be interrupted by exceptions. See int mma9559_interrupts_disable(void) for more information.

- If user code supports nested critical sections, then the status parameter should be the value returned by the matching **`mma9559_interrupts_disable()`** function call.
- If user code does not support nested critical sections, then the status parameter may be set to 0 to re-enable interrupts. No other values should be used

Only the interrupt priority mask bits of the status parameter are used. If any of the interrupt priority mask bits are set, then the interrupt priority mask is set to 7 to disable/mask all interrupts; otherwise, the interrupt priority mask is set to 0 to enable/unmask the interrupts.

For example code using the **`mma9559_interrupts_restore()`** function, see the description of the int mma9559_interrupts_disable(void) function.

See also:

- **int mma9559_interrupts_disable(void)**
- “Interrupts and critical sections” on page 22

Parameter

`status`: Interrupt status state to be restored.

4.3.4.9 int mma9559_user_trap0(int d0, int d1, int d2, void *a0, void *a1)

This function issues a trap instruction that calls the function registered in the firmware variables structure entry **`user_trap_handler[0]`**. The parameter variables are passed to the trap function and their usage is determined by the user trap handler function.

Return

The integer value returned by the **`user_trap_handler`** function.

NOTE

The user trap handler can be called directly using the **`asm { trap #TRAP_USER_0 }`** instruction. This function makes it easier to set the parameter values.

Parameters

Table 4-11. int mma9559_user_trap0 parameters

Parameter	Description
d0	First integer parameter (user-defined meaning).
d1	Second integer parameter (user-defined meaning).
d2	Third integer parameter (user-defined meaning).
a0	First pointer parameter (user-defined meaning).
a1	Second pointer parameter (user-defined meaning).

4.3.4.10 int mma9559_user_trap1(int d0, int d1, int d2, void *a0, void *a1)

This function issues a trap instruction that calls the function registered in the firmware variables structure entry **`user_trap_handler[1]`**. The parameter variables are passed to the trap function and their usage is determined by the user trap handler function.

Return

The integer value returned by the **`user_trap_handler`** function.

NOTE

The user trap handler can be called directly using the **`asm { trap #TRAP_USER_1 }`** instruction. This function makes it easier to set the parameter values.

Parameter

Table 4-12. int mma9559_user_trap1 parameters

Parameter	Description
d0	First integer parameter (user-defined meaning).
d1	Second integer parameter (user-defined meaning).
d2	Third integer parameter (user-defined meaning).
a0	First pointer parameter (user-defined meaning).
a1	Second pointer parameter (user-defined meaning).

4.3.4.11 int mma9559_user_trap2(int d0, int d1, int d2, void *a0, void *a1)

This function issues a trap instruction that calls the function registered in the firmware variables structure entry **`user_trap_handler[2]`**. The parameter variables are passed to the trap function and their usage is determined by the user trap handler function.

Return

The integer value returned by the **`user_trap_handler`** function.

NOTE

The user trap handler can be called directly using the **`asm { trap #TRAP_USER_2 }`** instruction. This function makes it easier to set the parameter values.

Parameters

Table 4-13. int mma9559_user_trap2 parameters

Parameter	Description
d0	First integer parameter (user-defined meaning).
d1	Second integer parameter (user-defined meaning).
d2	Third integer parameter (user-defined meaning).
a0	First pointer parameter (user-defined meaning).
a1	Second pointer parameter (user-defined meaning).

4.3.4.12 int mma9559_user_trap3(int d0, int d1, int d2, void *a0, void *a1)

This function issues a trap instruction that calls the function registered in the firmware variables structure entry **`user_trap_handler[3]`**. The parameter variables are passed to the trap function, and their usage is determined by the user trap handler function.

Return

This is the integer value returned by the **`user_trap_handler`** function.

NOTE

The user trap handler can be called directly using the **`asm { trap #TRAP_USER_3 }`** instruction. This function makes it easier to set the parameter values.

Parameters

Table 4-14. int mma9559_user_trap3 parameters

Parameter	Description
d0	First integer parameter (user-defined meaning)
d1	Second integer parameter (user-defined meaning)
d2	Third integer parameter (user-defined meaning)
a0	First pointer parameter (user-defined meaning)
a1	Second pointer parameter (user-defined meaning)

4.4 FIFOs

The MMA9559L Freescale firmware supports the use of FIFOs for inter-process communication, with a configurable FIFO structure definition and a set of functions to manage them. For overview information, see “FIFOs” on page 24.

4.4.1 Macros

4.4.1.1 #define FIFO_STRUCT(max_entries, bytes_per_entry) struct { uint32 rsvd[2]; uint8 data[max_entries * bytes_per_entry]; }

The MMA9559L firmware provides functions to store and retrieve data in FIFOs. The FIFO data structure contains a common set of internal variables that are the same in every FIFO and a data buffer whose length is determined by the number of data entries in the FIFO and the size of each entry. The entry size is in bytes.

This macro can be used to create FIFOs in two ways:

- A single variable instance of a FIFO can be created using the macro, as shown in the following example.

Example 4-15.

```
  FIFO_STRUCT(5, sizeof(uint32)) fifo1;
// FIFO variable to hold five 32 bit unsigned integers
```

- A typedef can be created for a particular FIFO configuration and individual variable instances of the same configuration created from that typedef, as shown in the following example.

Example 4-16.

```
typedef FIFO_STRUCT(10, sizeof(uint8)) fifo10_t;// FIFO data type to hold 10 bytes
fifo10_t fifo2;                  // Instance variable of a FIFO to hold 10 bytes
```

NOTE

When creating the FIFO structure, this macro calculates the total data buffer size (in bytes) by multiplying the **`maximum_entries`** field by the **`bytes_per_entry`** field. The two, separate values are not needed here, but are kept in this format to maintain consistency with the **`mma9559_fifo_init()`** function.

See also:

- **int mma9559_fifo_init(volatile mma9559_fifo_t *fifo_ptr, events_t events, unsigned int max_entries, unsigned int bytes_per_entry)**
- “FIFOs” on page 24

4.4.2 Data structures

4.4.2.1 mma9559_fifo_t

This structure provides a generic FIFO structure that contains the internal variables used by the FIFO functions and a placeholder for the data buffer. This structure is not used directly because it does not have any data buffer space. Instead, users should create the FIFOs they need using the FIFO_STRUCT macro and include a correctly sized data buffer.

The contents of the structure should not be accessed directly by user code, but accessed by MMA9559L-firmware functions.

See also:

- **int mma9559_fifo_init(volatile mma9559_fifo_t *fifo_ptr, events_t events, unsigned int max_entries, unsigned int bytes_per_entry)**
- **int mma9559_fifo_pop(volatile mma9559_fifo_t *fifo_ptr, uint8 *data_ptr, unsigned int entries)**
- **#define FIFO_STRUCT(max_entries, bytes_per_entry) struct { uint32 rsvd[2]; uint8 data[max_entries * bytes_per_entry]; }**

Fields

Table 4-15. mma9559_fifo_t fields

Field	Description
events_t events	Events to signal when the FIFO has data.
uint8 read_index	Index for the next entry to be read.
uint8 maximum_entries	Maximum number of elements in the FIFO.
uint8 number_of_entries	Number of entries currently in the FIFO.
uint8 bytes_per_entry	Number of bytes in each data element.
uint8 data[]	Configurable size buffer for FIFO data.

4.4.3 Functions

4.4.3.1 int mma9559_fifo_init(volatile mma9559_fifo_t *fifo_ptr, events_t events, unsigned int max_entries, unsigned int bytes_per_entry)

This function initializes the FIFO data structure. The structure first must be defined, using the FIFO_STRUCT macro that creates a customized FIFO variable or data type. At run time, the FIFO structure must be initialized using this function.

Whenever data is pushed into the FIFO, the events in the events field is signaled and the event is cleared when either the last data is popped off the FIFO (leaving it empty) or the FIFO is reset. The events are also added to the **`mma9559_vars events_fifos`** field, which means they can be signaled or cleared by the user exception handler or the **`mma9559_events_set_clear()`** function.

The **`max_entries`** and **`bytes_per_entry`** values both must be in the range of 1 to 255. If either value is out of this range, the function returns a value of -1 and leaves the FIFO structure marked as unconfigured.

The following example:

- Initializes fifo1.
- Sets the FIFO size to the same values that were used in the FIFO data type (maximum entries of 10, entry size of 2 bytes).
- Associates fifo1 with the EVENT_FIFO event.

Example 4-17.

```
// Instantiate the FIFO to reserve the memory space
  FIFO_STRUCT(2, sizeof(mma9559_afe_data_t)) fifo1;

  ...
// At run time initialize the FIFO before using it
  ret = mma9559_fifo_init(
         (mma9559_fifo_t*)&fifo1,     // pointer to the FIFO structure
         EVENT_BITFIELD(EVENT_FIFO),  // bitfield of associated events
         2,                           // maximum number of data entries
         sizeof(mma9559_afe_data_t)   // number of bytes in each data entry
    );
```

The **`max_entries`** and **`bytes_per_entry`** fields normally match the values used in the FIFO_STRUCT macro to create the FIFO variable. Other values may be used, however, as long as the product of the **`max_entries`** and **`bytes_per_entry`** values—used in the **`mma9559_fifo_init()`** function—is not larger than the product of the values used by the FIFO_STRUCT macro to determine the FIFO data buffer size.

NOTE

There is no automatic validation of the **`max_entries`** and **`bytes_per_entry`** values. The user must ensure that these values are consistent with the FIFO_STRUCT usage.

See also:

- **#define FIFO_STRUCT(max_entries, bytes_per_entry) struct { uint32 rsvd[2]; uint8 data[max_entries * bytes_per_entry]; }**
- “Initialize FIFO” on page 25

Return

Error status:

- If the **max_entries** or **bytes_per_entry** values are greater than 255 or less than 1, then -1 is returned.
- Otherwise, 0 is returned on success

Parameters

Table 4-16. int mma9559_fifo_init parameters

Field	Description
fifo_ptr	Address of the FIFO data structure to be initialized.
events	Bit field of events signaled, when the FIFO contains data.
max_entries	Maximum number of entries that can be stored.
bytes_per_entry	Bytes per FIFO entry.

4.4.3.2 void mma9559_fifo_reset(volatile mma9559_fifo_t *fifo_ptr)

This function resets the FIFO variables to empty the FIFO and clears any currently signaled event bits for the FIFO. When the FIFO is reset, all data in the FIFO is discarded and the FIFO data structure is re-initialized.

Example 4-18.

```
mma9559_fifo_reset(
  (mma9559_fifo_t*)&fifo1      // pointer to the FIFO structure to be reset
);
```

See also:

- “Reset the FIFO” on page 27

Parameter

fifo_ptr: Address of the FIFO data structure to be reset.

4.4.3.3 int mma9559_fifo_pop(volatile mma9559_fifo_t *fifo_ptr, uint8 *data_ptr, unsigned int entries)

This function copies the requested number of entries from the FIFO to memory, starting at the supplied data pointer. If the FIFO contains less entries than the number of entries requested, only the number of entries in the FIFO are copied.

After the data has been copied from the FIFO, the data is discarded from the FIFO buffer—freeing space for new data. If the FIFO is empty after the data has been read, the events associated with the FIFO are cleared.

It does not corrupt the FIFO to attempt to pop data off an empty FIFO or to read more data entries from a FIFO than it contains. Either action, however, results in all of the available entries being read and the FIFO being emptied.

Example 4-19.

```
ret = mma9559_fifo_pop(
       (mma9559_fifo_t*)&fifo1,   // pointer to the FIFO structure
       (uint8*)afe_data.data,     // pointer to the start of the destination buffer
       1                          // the number of data entries to retrieve
  );
```

If the FIFO has not been successfully initialized with the **`mma9559_fifo_init()`** function, no data is read from the FIFO and the return value is 0.

NOTE

The buffer addressed by the **`data_ptr`** parameter must be large enough to hold the number of bytes being popped. This value must be greater than or equal to the entries parameter multiplied by the number of bytes per entry, set when the FIFO was initialized. This function does not check the size of the data buffer.

Return

The actual number of entries that were popped off the FIFO and copied to the data pointer.

Parameters

Table 4-17. int mma9559_fifo_pop parameters

Field	Description
fifo_ptr	Address of the FIFO data structure to obtain data.
data_ptr	Address of the structure to store the popped data.
entries	Number of entries to pop off the FIFO.

4.4.3.4 int mma9559_fifo_push(volatile mma9559_fifo_t *fifo_ptr, uint8 *data_ptr, unsigned int entries)

This function copies the requested number of entries—from the memory starting at the supplied data pointer—into the FIFO. If the FIFO does not have enough space (empty FIFO entries) to store the requested number of entries, then only the number of entries fitting within that space are copied.

The events associated with the FIFO are signaled every time this function is called, regardless of the number of entries that are pushed into the FIFO.

If the number of entries to be pushed into the FIFO exceeds the space in the FIFO, the FIFO overflows. The entries that do not fit are not stored in the FIFO and the events bit field, associated with the FIFO, are set in the **`mma9559_vars_t events_missed`** field.

The FIFO is not corrupted by attempting to push data into a full FIFO or to write data entries that exceed the available space. However, the excess data (in both scenarios) are not stored in the FIFO and are lost.

Example 4-20.

```
ret = mma9559_fifo_push(
       (mma9559_fifo_t*)&fifo1,   // pointer to the FIFO structure
       (uint8*)afe_data.data,     // pointer to the first byte of the first entry
       1                          // the number of data entries to store
  );
```

If the FIFO has not been successfully initialized with the **`mma9559_fifo_init()`** function, no data is read from the FIFO and the return value is 0.

NOTE

The buffer addressed by the **`data_ptr`** parameter must contain the correct number of bytes to match the number of entries being pushed. The parameter's value must be equal to the entries parameter multiplied by the number of bytes per entry, set when the FIFO was initialized. This function does not check the size of the data buffer.

Return

The actual number of entries that were copied from the data pointer and pushed into the FIFO.

Parameters

Table 4-18. int mma9559_fifo_push parameters

Field	Description
fifo_ptr	Address of the FIFO data structure.
data_ptr	Address of the structure from which the data is pushed.
entries	Number of entries to push into the FIFO.

4.4.3.5 int mma9559_fifo_entries_used(volatile mma9559_fifo_t *fifo_ptr)

This function returns the number of entries in the FIFO that currently contain valid data. User code may pop this many entries off the FIFO without running out of data. The FIFO may still contain data after this many reads because more data may be pushed onto the FIFO by interrupt handlers while the FIFO is being read. The FIFO should not run out of data.

Example 4-21.

```
used_entries = mma9559_fifo_entries_used(
                 (mma9559_fifo_t*)&fifo1,   // pointer to the FIFO structure
               );
```

Return

The number of entries in the FIFO that currently contain data that can be popped off the FIFO.

Parameter

fifo_ptr: Address of the FIFO data structure.

4.4.3.6 int mma9559_fifo_entries_free(volatile mma9559_fifo_t *fifo_ptr)

This function returns the number of entries in the FIFO that currently are empty. User code may push this many entries into the FIFO without running out of FIFO space and overflowing the FIFO. The entries value is only correct at the time it is read. If any other tasks (such as interrupt handlers) are using the FIFO, there may be more or less space in the FIFO when the user code actually accesses the FIFO.

Example 4-22.

```
free_entries = mma9559_fifo_entries_free(
                    (mma9559_fifo_t*)&fifo1,   // pointer to the FIFO structure
               );
```

Return

The number of entries in the FIFO that are currently available to store more data.

Parameter

fifo_ptr: Address of the FIFO data structure.

4.5 Other functions

This section describes additional firmware functionality that does not fit into the previous functional categories.

4.5.1 Enumerations

4.5.1.1 enum boot_options_t

This sets the execution path for a startup other than power-on reset.

When the device first powers up, the boot mode is loaded with the FOPT settings from the flash location at 0x3FFE. This is set in the flash code image.

On subsequent resets, the FOPT settings are not reloaded from the flash location. Instead, the register settings are used directly. This function enables users to modify the FOPT setting for the execution path in order to set the operation next time the device resets.

See also:

- **void mma9559_boot_options_set(boot_options_t option)**

Enumerators

Table 4-19. enum boot_options_t enumerators

Enumerator	Description
BOOT_TO_ROM	On the next non power-on reset, execute the ROM command line interpreter.
BOOT_TO_FLASH	On the next non power-on reset, execute the firmware loaded in the flash memory.

4.5.1.2 enum rmf_func_t

This value specifies the ROM command to execute. For more information on the operation and data formats for the commands, see the “User Callable ROM Functions” section of the “ROM” chapter in the *MMA955xL Intelligent, Motion-Sensing Hardware Reference Manual* (MMA955xLHWRM).

See also:

- **void* mma9559_rom_command(rmf_func_t func_id, void *addr)**

Enumerators

Table 4-20. enum rmf_func_t enumerators

Enumerator	Description
RMF_DEV_INFO	Retrieves device information data structure.
RMF_FLASH_PROGRAM	Programs flash memory, through the flash controller.
RMF_FLASH_ERASE	Erases flash memory, through the flash controller.
RMF_EXTENSION	Executes extended flash functions.
RMF_CRC	Calculates the CRC, over a range in memory.
RMF_CI	Transfers command to the ROM-based command interpreter.
RMF_CHANGE_CONFIG	Updates the device capabilities configuration.
RMF_FLASH_PROTECT	Protects the flash against accidental erasure and programming.
RMF_FLASH_UNPROTECT	Unprotects access to the flash program and erase functions.
RMF_FLASH_UNSECURE	Modifies the current security status of the device.

4.5.2 Data structures

4.5.2.1 mma9559_device_info_t

This structure holds device information that is derived from a combination of the ROM device, version information, and firmware-version information.

- The **device_id**, **rom_version**, and **hw_version** values are retrieved from the ROM device information.
- The **fw_version**, **build_code**, **part_number**, **reset cause**, and **security state** are reported by the MMA9559L firmware.

Fields

Table 4-21. mma9559_device_info_t fields

Field	Description
uint32 device_id	ROM: Pseudo-random part identification value.
uint16 rom_version	ROM: ROM version code, major.minor.
uint16 fw_version	FW: Firmware version code, major.minor.
uint16 hw_version	ROM: Hardware version code, major.minor.
uint16 build_code	FW: Firmware build number and date code. The value is encoded in the following bit fields: • [15:12] Daily build number, 0 to 15 • [11: 8] Build month, 1 to 12 • [7: 3] Build day, 1 to 31 • [2: 0] Build year, 2010 to 2017
uint16 part_number	FW: BCD-encoded part number. For example, 0x9559.
uint8 reset_cause	FW: Lower five bits from the RCSR reports reset source.
uint8 secure_mode	FW: Lower two bits of FOPT report the security mode of the device. Values: • 2 = secure • Otherwise = not secure

4.5.2.2 union rmf_return_t

This returns a value from the ROM command, accessible as a value or a pointer.

ROM commands return a single value that can be either a pointer or a value, depending on which function was called. The **`mma9559_rom_command()`** function returns a pointer that can be assigned to the **`ptr`** field in the union and handled by user code as either a pointer or a value, depending on which ROM command was executed.

See also:

- **void* mma9559_rom_command(rmf_func_t func_id, void *addr)**

Fields

Table 4-22. union rmf_return_t fields

Field	Description
void* ptr	Return value handled as a pointer
unsigned long val	Return value handled as an integer

4.5.3 Functions

4.5.3.1 void mma9559_boot_options_set(boot_options_t option)

Sets the execution path for the next non-POR reset.

This function enables the execution path on the next reset, other than a power-on reset, to be configured to run either the ROM command interpreter or to execute the user firmware in flash.

When the device first powers up, the boot mode is loaded with the FOPT settings from the flash location at 0x3FFE. This is set in the flash code image. On subsequent resets, the FOPT settings are not reloaded from the flash location. Instead, the register settings are used directly.

This function enables users to modify the FOPT setting for the execution path to set the operation next time the device resets.

Once the execution path has been set using the **`mma9559_boot_options_set()`** function, a reset can be generated by writing to the Assert Software Reset (ASR) bit of the user-accessible, Reset Control and Status Register (RCSR). Resets caused by anything other than the POR also follow the execution path set with this function.

The following example resets the device back to the ROM Command Interpreter.

Example 4-23.

```
mma9559_boot_options_set(BOOT_TO_ROM);        // Set next reset boot path to ROM
RCSR_ASR = 1;                                 // Issue software reset
```

NOTE

The MMA955xL EVM board issues a reset to the MMA955xL device whenever it connects to the device to ensure that the correct I^2C / SPI connection selection is used. If the user code already has set the boot path to ROM using **`mma9559_boot_options_set(BOOT_TO_ROM)`**, the part always runs the ROM Command interpreter when the PC connects to the part through the MMA955xL EVM board. This problem is avoided by not setting the boot path to ROM until a reset to ROM is desired.

Parameter

option: Selects the execution path on the next non-POR reset.

4.5.3.2 int mma9559_device_info_get(int length, mma9559_device_info_t *addr)

This function retrieves a combination of identification and version information from the ROM code and the MMA9559L device and copies it into the data structure at **addr**.

To avoid overflowing the supplied data buffer, the caller must provide a length parameter that limits the number of bytes that are copied to the supplied data buffer, as shown in the following example.

Example 4-24.

```
  mma9559_device_info_t device_info;
// storage to hold the returned device information
  device_info_length = mma9559_device_info_get(
               sizeof(mma9559_device_info_t), // maximum number of bytes to read
               &device_info                   // address of buffer to receive data
               );
```

NOTE

This function uses a large amount of the supervisor stack space, so it should *never* be called from within the **user_exception_handler()** function or any **user_trap_handler()** function.

Return

Reports the number of bytes copied into the buffer, which is the minimum of the length parameter and the length of the available device identification and version information.

Parameters

Table 4-23. int mma9559_device_info_get parameters

Field	Description
length	Size of the receiving buffer, in bytes.
addr	Address of the buffer into which the device information is being copied.

4.5.3.3 void* mma9559_rom_command(rmf_func_t func_id, void *addr)

This function provides access to the ROM functions documented in the *MMA955xL Intelligent, Motion-Sensing Hardware Reference Manual* (MMA955xLHWRM). For details on the operation of these functions, see that document's "ROM" chapter.

Example 4-25.

```
rmf_return_t rmf_ret;
rmf_ret.ptr = mma9559_rom_command(
            RMF_DEV_INFO,      // ROM function to be called = Get Device Info
            0                  // this command does not take any arguments
       );
```

Return

Either provides the address of a parameter block containing the result of the command or returns a value, depending on the ROM function called.

Parameters

Table 4-24. void* mma9559_rom_command parameters

Field	Description
func_id	Specifies the ROM functions to be executed.
addr	Gives the address of the parameter block used for the ROM function.

4.6 IIR filter

The MMA9559L firmware includes an optimized Direct Form I, *N*th-order Infinite Impulse Response (IIR) filter function that can be accessed by user code.

A Direct Form I IIR filter is calculated using a general formula than can be extended to an *N*th-order filter. See the following equation.

$$Y[n] = b0*X[n] + b1*X[n-1] - a1*Y[n-1] + b2*X[n-2] - a2*Y[n-2] \ldots \quad \textbf{Eqn. 4-1}$$

Where:

- X[*n*] is the current input
- X[*n*-*N*] is the input from *N* cycles earlier
- Y[*n*] is the current output
- Y[*n*-*N*] is the output from *N* cycles earlier
- a*N* and b*N* are the filter coefficients that are fixed for a particular filter response

In this implementation, the input, output, and coefficient values are all 16-bit, signed integers, but the intermediate accumulation of the result uses a 32-bit accumulator that is right-shifted, at the end of the operation, to normalize the result.

For more detailed information, see the related IIR Filter applications note AN4464, *Digital Filtering with MMA955xL*.

4.6.1 Data structures

4.6.1.1 mma9559_coef_t

This structure is used to hold the coefficients for an IIR filter. The methods of calculating the coefficients for the filter is beyond this reference manual, but an sample low-pass filter is shown, in the following example, to illustrate how the data structure is used.

Example 4-26.

```
// 6th order Chebyshev type 2 lowpass filter, cutoff at fs/4
  static const mma9559_coef_t lp_cheby6 = {
                         6,       // filter order = 6
          14,                     // coefficient shift = 1/16384
      { 16384,         441,       // a0, b0  =  1.000000,  0.000244 (a0 value not used)
       -16546,        1637,       // a1, b1  = -1.009888,  0.099915
        20052,        3191,       // a2, b2  =  1.223877,  0.194763
        -8837,        3925,       // a3, b3  = -0.539368,  0.239563
         3957,        3191,       // a4, b4  =  0.241516,  0.194763
         -624,        1637,       // a5, b5  = -0.038086,  0.099915
           78,         441 }      // a6, b6  =  0.004761,  0.000244
};
```

See also:

- **int16 mma9559_iir_filter(int16 input, const mma9559_coef_t *coef, void *buffer)**

Fields

Table 4-25. mma9559_coef_t fields

Field	Description
uint16 order	Defines the order of the filter and determines the size of the `coef_ary` array.
uint16 shift	Specifies the number of fractional bits in the coefficients to determine how many bits to right-shift the output of the filter.
int16 coef_ary[]	Holds the coefficient values for the filter. The number of coefficients is determined by the order of the filter and is calculated as 2 * (order + 1); for example, for a sixth order filter, there are 14 coefficients.

4.6.2 Functions

4.6.2.1 int16 mma9559_iir_filter(int16 input, const mma9559_coef_t *coef, void *buffer)

This function applies the supplied input data to the specified IIR filter. The coefficient structure is predefined for each particular transform. The same coefficient structure may be reused, such as for each channel of the X, Y, and Z accelerometer data.

The buffer is used to hold the previous X[*n*] and Y[*n*] values. In order to keep the previous values, the buffer must be persistent; therefore it is either global or static. It is updated by the filter function on each execution, as new data enters into the filter.

The size of the buffer depends on the order of the filter and is 2 * order-words long. This is declared as shown in the following code:

Example 4-27.

```
  int32 buffer[ORDER];      // each int32 holds a pair of 16 bit {yn,xn} values
```

A three-channel, low-pass filter could be implemented as shown in the following example.

Example 4-28.

```
// 6th order Chebyshev type 2 lowpass filter, cutoff at fs/4
  static const mma9559_coef_t lp_cheby6 = {
                    6,          // filter order = 6
                   14,          // coefficient shift = 1/16384
     { 16384,     441,          // a0, b0  =  1.000000,  0.000244 (a0 value not used)
      -16546,     1637,           // a1, b1  = -1.009888,  0.099915
       20052,     3191,           // a2, b2  =  1.223877,  0.194763
       -8837,     3925,           // a3, b3  = -0.539368,  0.239563
        3957,     3191,           // a4, b4  =  0.241516,  0.194763
        -624,     1637,           // a5, b5  = -0.038086,  0.099915
          78,     441 }           // a6, b6  =  0.004761,  0.000244
  };

// buffers are global to maintain their contents across function calls
  int32 buffer_x[6];    // buffer to hold X intermediate values for 6th order filter
  int32 buffer_y[6];    // buffer to hold Y intermediate values for 6th order filter
  int32 buffer_z[6];    // buffer to hold Z intermediate values for 6th order filter

// Function to low pass filter X, Y and Z channels
  void filter(int16 *output, int16 *input) {
    *(output++) = mma9559_iir_filter(*(input++), &lp_cheby6, &buffer_x);
// Filter X data
    *(output++) = mma9559_iir_filter(*(input++), &lp_cheby6, &buffer_y);
// Filter Y data
    *(output++) = mma9559_iir_filter(*(input++), &lp_cheby6, &buffer_z);
// Filter Z data
  }
```

For more information about using this function, see the related IIR Filter applications note AN4464, *Digital Filtering with MMA955xL*.

Parameters

Table 4-26. int16 mma9559_iir_filter parameters

Parameter	Description
input	Filter input data value.
coef	Pointer to the IIR filter coefficient data structure.
buffer	Pointer to the working data buffer used by the filter.

4.6.3 Typedefs

4.6.3.1 typedef struct mma9559_coef_t mma9559_coef_t

This structure is used to hold the coefficients for an IIR filter. The methods for calculating the coefficients for the filter is beyond this reference manual, but a sample, low-pass filter is given in the following example.

Example 4-29.

```
// 6th order Chebyshev type 2 lowpass filter, cutoff at fs/4
  static const mma9559_coef_t lp_cheby6 = {
                    6,           // filter order = 6
                   14,           // coefficient shift = 1/16384
     16384,       441,          // a0, b0  =  1.000000,  0.000244 (a0 value not used)
    -16546,       1637,         // a1, b1  = -1.009888,  0.099915
    20052,        3191,         // a2, b2  =  1.223877,  0.194763
    -8837,        3925,         // a3, b3  = -0.539368,  0.239563
    3957,         3191,         // a4, b4  =  0.241516,  0.194763
    -624,         1637,         // a5, b5  = -0.038086,  0.099915
    78,           441 }         // a6, b6  =  0.004761,  0.000244
  };
```

See also:

- **int16 mma9559_iir_filter(int16 input, const mma9559_coef_t *coef, void *buffer)**

Appendix A
Revision History

This appendix describes corrections to the MMA9559L Intelligent, Motion-Sensing Platform Software Reference Manual. For convenience, the corrections are grouped by revision.

A.1 Changes Between Revisions 0 and 0.1

Rev. 0 of this document was published on Feb 1 2012.

Table A-1. Changes Between Revisions 0 and 0.1

Chapter	Description
Chapter 2, “Firmware Overview”	• In Table 2-1., “Firmware flash usage,” for flash region 0x0000_0804 to 0x0000_3FFB , corrected the size to 14328 (was 14332, which is wrong). (Mar 8 2012) • Fixed a typo in Section 2.5.4, “Pop data off the FIFO”— changed “Since the FIFO push operation is performed” to “Since the FIFO pop operation is performed”.
	•
	•
	•