Actel SmartFusion™ MSS I²C Driver
User’s Guide

Version 2.0
# Table of Contents

**Introduction** ................................................................................................................................. 5
  Features ............................................................................................................................................... 5
  Supported Hardware IP ...................................................................................................................... 5

**Files Provided** ................................................................................................................................. 7
  Documentation ..................................................................................................................................... 7
  Driver Source Code ............................................................................................................................ 7
  Example Code ...................................................................................................................................... 8

**Driver Deployment** .......................................................................................................................... 9

**Driver Configuration** ....................................................................................................................... 11

**Application Programming Interface** .............................................................................................. 13
  Theory of Operation ............................................................................................................................ 13
  Types .................................................................................................................................................. 17
  Constant Values ................................................................................................................................. 18
  Data structures ................................................................................................................................. 19
  Global Variables ............................................................................................................................... 19
  Functions ............................................................................................................................................ 20

**Product Support** .............................................................................................................................. 31
  Customer Service ............................................................................................................................... 31
  Actel Customer Technical Support Center ......................................................................................... 31
  Actel Technical Support .................................................................................................................... 31
  Website ............................................................................................................................................... 31
  Contacting the Customer Technical Support Center ......................................................................... 31
Introduction

The SmartFusion™ microcontroller subsystem (MSS) includes two I2C peripherals for serial communication. This driver provides a set of functions for controlling the MSS I2Cs as part of a bare metal system where no operating system is available. These drivers can be adapted for use as part of an operating system, but the implementation of the adaptation layer between this driver and the operating system’s driver model is outside the scope of this driver.

Features

The MSS I2C driver provides the following features:
- Support for configuring each MSS I2C peripheral
- I2C master operations
- I2C slave operations

The MSS I2C driver is provided as C source code.

Supported Hardware IP

The MSS I2C bare metal driver can be used with Actel’s MSS_I2C IP version 0.2 or higher included in the SmartFusion MSS.
Files Provided

The files provided as part of the MSS I2C driver fall into three main categories: documentation, driver source code, and example projects. The driver is distributed via the Actel Firmware Catalog, which provides access to the documentation for the driver, generates the driver’s source files into an application project, and generates example projects that illustrate how to use the driver.

Documentation

The Actel Firmware Catalog provides access to these documents for the driver:

- User’s guide (this document)
- A copy of the license agreement for the driver source code
- Release notes

Driver Source Code

The Actel Firmware Catalog generates the driver’s source code into a `drivers\mss_i2c` subdirectory of the selected software project directory. The files making up the driver are detailed below.

`mss_i2c.h`

This header file contains the public application programming interface (API) of the MSS I2C software driver. This file should be included in any C source file that uses the MSS I2C software driver.

`mss_i2c.c`

This C source file contains the implementation of the MSS I2C software driver.
Example Code

The Actel Firmware Catalog provides access to example projects illustrating the use of the driver. Each example project is self-contained and is targeted at a specific processor and software toolchain combination. The example projects are targeted at the FPGA designs in the hardware development tutorials supplied with Actel’s development boards. The tutorial designs can be found on the Actel Development Kit web page (www.actel.com/products/hardware).
Driver Deployment

This driver is intended to be deployed from the Actel Firmware Catalog into a software project by generating the driver's source files into the project directory. The driver uses the SmartFusion ARM® Cortex™ Microcontroller Software Interface Standard – Peripheral Access Layer (CMSIS-PAL) to access MSS hardware registers. You must ensure that the SmartFusion CMSIS-PAL is either included in the software tool chain used to build your project or is included in your project. The most up-to-date SmartFusion CMSIS-PAL files can be obtained using the Actel Firmware Catalog.

The following example shows the intended directory structure for a SoftConsole ARM® Cortex™-M3 project targeted at the SmartFusion MSS. This project uses the MSS I2C and MSS UART drivers. Both of these drivers rely on SmartFusion CMSIS-PAL for accessing the hardware. The contents of the drivers directory result from generating the source files for each driver into the project. The contents of the CMSIS directory result from generating the source files for the SmartFusion CMSIS-PAL into the project.

```
- SmartFusion_MSS_project
  - CMSIS
    - startup_gcc
  - drivers
    - mss_i2c
    - mss_uart
```

Figure 1 - SmartFusion MSS Project Example
The configuration of all features of the MSS FCs is covered by this driver with the exception of the SmartFusion IOMUX configuration. SmartFusion allows multiple non-concurrent uses of some external pins through IOMUX configuration. This feature allows optimization of external pin usage by assigning external pins for use by either the microcontroller subsystem or the FPGA fabric. The MSS FCs serial signals are routed through IOMUXes to the SmartFusion device external pins. These IOMUXes are automatically configured correctly by the MSS configurator tool in the hardware flow when the MSS FCs are enabled in that tool. You must ensure that the MSS FCs are enabled by the MSS configurator tool in the hardware flow, otherwise the serial inputs and outputs will not be connected to the chip's external pins. For more information on IOMUX, refer to the IOMUX section of the SmartFusion Datasheet.

The base address, register addresses and interrupt number assignment for the MSS FC blocks are defined as constants in the SmartFusion CMSIS-PAL. You must ensure that the SmartFusion CMSIS-PAL is either included in the software toolchain used to build your project or is included in your project.
Application Programming Interface

This section describes the driver's API. The functions and related data structures described in this section are used by the application programmer to control the MSS I2C peripheral from the user's application.

Theory of Operation

The MSS I2C driver functions are grouped into the following categories:

- Initialization and configuration functions
- Interrupt control
- I2C master operations – functions to handle write, read and write-read transactions
- I2C slave operations – functions to handle write, read and write-read transactions

Initialization and Configuration

The MSS I2C driver is initialized through a call to the `MSS_I2C_init()` function. This function takes the MSS I2C's configuration as parameters. The `MSS_I2C_init()` function must be called before any other MSS I2C driver functions can be called. The first parameter of the `MSS_I2C_init()` function is a pointer to one of two global data structures used by the driver to store state information for each MSS I2C. A pointer to these data structures is also used as first parameter to any of the driver functions to identify which MSS I2C will be used by the called function. The names of these two data structures are `g_mss_i2c0` and `g_mss_i2c1`. Therefore any call to an MSS I2C driver function should be of the form `MSS_I2C_function_name(&g_mss_i2c0, ...)` or `MSS_I2C_function_name(&g_mss_i2c1, ...)`.

The `MSS_I2C_init()` function call for each MSS I2C also takes the I2C serial address assigned to the MSS I2C and the serial clock divider to be used to generate its I2C clock as configuration parameters.

Interrupt Control

The MSS I2C driver is interrupt driven and it enables and disables the generation of interrupts by MSS I2C at various times when it is operating. The driver automatically handles MSS I2C interrupts internally, including enabling disabling and clearing MSS I2C interrupts in the Cortex-M3 interrupt controller when required.

The function `MSS_I2C_register_write_handler()` is used to register a write handler function with the MSS I2C driver that it will call on completion of an I2C write transaction by the MSS I2C slave. It is your responsibility to create and register the implementation of this handler function that will process or trigger the processing of the received data.

Transaction Types

The MSS I2C driver is designed to handle three types of transactions:

- Write transactions
- Read transactions
- Write-read transactions
**Write transaction**

The master I²C device initiates a write transaction by sending a START bit as soon as the bus becomes free. The START bit is followed by the 7-bit serial address of the target slave device followed by the read/write bit indicating the direction of the transaction. The slave acknowledges receipt of its address with an acknowledge bit. The master sends data one byte at a time to the slave, which must acknowledge receipt of each byte for the next byte to be sent. The master sends a STOP bit to complete the transaction.

![I²C write transaction diagram](image)

The slave can abort the transaction by replying with a non-acknowledge bit instead of an acknowledge.

The application programmer can choose not to send a STOP bit at the end of the transaction causing the next transaction to begin with a repeated START bit.

**Read transaction**

The master I²C device initiates a read transaction by sending a START bit as soon as the bus becomes free. The START bit is followed by the 7-bit serial address of the target slave device followed by the read/write bit indicating the direction of the transaction. The slave acknowledges receipt of its slave address with an acknowledge bit. The slave sends data one byte at a time to the master, which must acknowledge receipt of each byte for the next byte to be sent. The master sends a non-acknowledge bit following the last byte it wishes to read followed by a STOP bit.

![I²C read transaction diagram](image)

The application programmer can choose not to send a STOP bit at the end of the transaction causing the next transaction to begin with a repeated START bit.
Write-read transaction

The write-read transaction is a combination of a write transaction immediately followed by a read transaction. There is no STOP bit between the write and read phases of a write-read transaction. A repeated START bit is sent between the write and read phases.

The write-read transaction is typically used to send a command or offset in the write transaction specifying the logical data to be transferred during the read phase.

![Figure 4: I2C write-read transaction](image)

Master Operations

The application can use the MSS_I2C_write(), MSS_I2C_read() and MSS_I2C_write_read() functions to initiate an I2C bus transaction. The application can then wait for the transaction to complete using the MSS_I2C_wait_complete() function or poll the status of the I2C transaction using the MSS_I2C_get_status() function until it returns a value different from MSS_I2C_IN_PROGRESS.

Slave Operations

The configuration of the MSS I2C driver to operate as an I2C slave requires the use of the following functions:

- MSS_I2C_set_slave_tx_buffer()
- MSS_I2C_set_slave_rx_buffer()
- MSS_I2C_set_slave_mem_offset_length()
- MSS_I2C_register_write_handler()
- MSS_I2C_enable_slave_rx()

Use of all functions is not required if the slave I2C does not need to support all types of I2C read transactions. The subsequent sections list the functions that must be used to support each transaction type.

Responding to read transactions

The following functions are used to configure the MSS I2C driver to respond to I2C read transactions:

- MSS_I2C_set_slave_tx_buffer()
- MSS_I2C_enable_slave_rx()

The function MSS_I2C_set_slave_tx_buffer() specifies the data buffer that will be transmitted when the I2C slave is the target of an I2C read transaction. It is then up to the application to manage the content of that buffer to control the data that will be transmitted to the I2C master as a result of the read transaction.

The function MSS_I2C_enable_slave_rx() enables the MSS I2C hardware instance to respond to I2C transactions. It must be called after the MSS I2C driver has been configured to respond to the required transaction types.
Responding to write transactions

The following functions are used to configure the MSS I²C driver to respond to I²C write transactions:

- MSS_I2C_set_slave_rx_buffer()
- MSS_I2C_register_write_handler()
- MSS_I2C_enable_slave_rx()

The function MSS_I2C_set_slave_rx_buffer() specifies the data buffer that will be used to store the data received by the I²C slave when it is the target of an I²C write transaction.

The function MSS_I2C_register_write_handler() specifies the handler function that must be called on completion of the I²C write transaction. It is this handler function that will process or trigger the processing of the received data.

The function MSS_I2C_enable_slave_rx() enables the MSS I²C hardware instance to respond to I²C transactions. It must be called after the MSS I²C driver has been configured to respond to the required transaction types.

Responding to write-read transactions

The following functions are used to configure the MSS I²C driver to respond to write-read transactions:

- MSS_I2C_set_slave_tx_buffer()
- MSS_I2C_set_slave_rx_buffer()
- MSS_I2C_set_slave_mem_offset_length()
- MSS_I2C_enable_slave_rx()

The function MSS_I2C_set_slave_mem_offset_length() specifies the number of bytes expected by the I²C slave during the write phase of the write-read transaction.

The function MSS_I2C_set_slave_tx_buffer() specifies the data that will be transmitted to the I²C master during the read phase of the write-read transaction. The value received by the I²C slave during the write phase of the transaction will be used as an index into the transmit buffer specified by this function to decide which part of the transmit buffer will be transmitted to the I²C master as part of the read phase of the write-read transaction.

The function MSS_I2C_set_slave_rx_buffer() specifies the data buffer that will be used to store the data received by the I²C slave during the write phase of the write-read transaction. This buffer must be at least large enough to accommodate the number of bytes specified through the MSS_I2C_set_slave_mem_offset_length() function.

The function MSS_I2C_enable_slave_rx() enables the MSS I²C hardware instance to respond to I²C transactions. It must be called after the MSS I²C driver has been configured to respond to the required transaction types.
Types

**mss_i2c_clock_divider_t**

**Prototype**

typedef enum mss_i2c_clock_divider {
    MSS_I2C_PCLK_DIV_256 = 0,
    MSS_I2C_PCLK_DIV_224,
    MSS_I2C_PCLK_DIV_192,
    MSS_I2C_PCLK_DIV_160,
    MSS_I2C_PCLK_DIV_960,
    MSS_I2C_PCLK_DIV_120,
    MSS_I2C_PCLK_DIV_60,
    MSS_I2C_BCLK_DIV_8
} mss_i2c_clock_divider_t;

**Description**
The `mss_i2c_clock_divider_t` type is used to specify the divider to be applied to the MSS I²C BCLK signal in order to generate the I²C clock.

**mss_i2c_status_t**

**Prototype**

typedef enum mss_i2c_status {
    MSS_I2C_SUCCESS = 0,
    MSS_I2C_IN_PROGRESS,
    MSS_I2C_FAILED
} mss_i2c_status_t;

**Description**
The `mss_i2c_status_t` type is used to report the status of I²C transactions.

**mss_i2c_slave_handler_ret_t**

**Prototype**

typedef enum mss_i2c_slave_handler_ret {
    MSS_I2C_REENABLE_SLAVE_RX = 0,
    MSS_I2C_PAUSE_SLAVE_RX = 1
} mss_i2c_slave_handler_ret_t;

**Description**
The `mss_i2c_slave_handler_ret_t` type is used by slave write handler functions to indicate whether the received data buffer should be released or not.


**mss_i2c_slave_wr_handler_t**

**Prototype**

typedef mss_i2c_slave_handler_ret_t (*mss_i2c_slave_wr_handler_t)( uint8_t *, uint16_t );

**Description**

This defines the function prototype that must be followed by MSS I2C slave write handler functions. These functions are registered with the MSS I2C driver through the `MSS_I2C_register_write_handler()` function.

**Declaring and Implementing Slave Write Handler Functions**

Slave write handler functions should follow the following prototype:

```c
mss_i2c_slave_handler_ret_t write_handler( uint8_t * data, uint16_t size );
```

The *data* parameter is a pointer to a buffer (received data buffer) holding the data written to the MSS I2C slave. The *size* parameter is the number of bytes held in the received data buffer.

Handler functions must return one of the following values:

- MSS_I2C_REENABLE_SLAVE_RX
- MSS_I2C_PAUSE_SLAVE_RX.

If the handler function returns MSS_I2C_REENABLE_SLAVE_RX, the driver will release the received data buffer and allow further I2C write transactions to the MSS I2C slave to take place.

If the handler function returns MSS_I2C_PAUSE_SLAVE_RX, the MSS I2C slave will respond to subsequent write requests with a non-acknowledge bit (NACK), until the received data buffer content has been processed by some other part of the software application.

A call to `MSS_I2C_enable_slave_rx()` is required at some point after returning MSS_I2C_PAUSE_SLAVE_RX in order to release the received data buffer so it can be used to store data received by subsequent I2C write transactions.

**Constant Values**

**MSS_I2C_RELEASE_BUS**

The MSS_I2CRELEASE_BUS constant is used to specify the *options* parameter to functions `MSS_I2C_read()`, `MSS_I2C_write()` and `MSS_I2C_write_read()` to indicate that a STOP bit must be generated at the end of the I2C transaction to release the bus.

**MSS_I2C_HOLD_BUS**

The MSS_I2C_HOLD_BUS constant is used to specify the *options* parameter to functions `MSS_I2C_read()`, `MSS_I2C_write()` and `MSS_I2C_write_read()` to indicate that a STOP bit must not be generated at the end of the I2C transaction in order to retain the bus ownership. This will cause the next transaction to begin with a repeated START bit and no STOP bit between the transactions.
Data structures

_mss_i2c_instance_t_

There is one instance of this structure for each of the microcontroller subsystem’s I2Cs. Instances of this structure are used to identify a specific I2C. A pointer to an instance of the _mss_i2c_instance_t_ structure is passed as the first parameter to MSS I2C driver functions to identify which I2C will perform the requested operation.

Global Variables

_g_mss_i2c0_

Prototype

```c
mss_i2c_instance_t g_mss_i2c0;
```

Description

This instance of _mss_i2c_instance_t_ holds all data related to the operations performed by MSS I2C 0. A pointer to _g_mss_i2c0_ is passed as the first parameter to MSS I2C driver functions to indicate that MSS I2C 0 will perform the requested operation.

_g_mss_i2c1_

Prototype

```c
mss_i2c_instance_t g_mss_i2c1;
```

Description

This instance of _mss_i2c_instance_t_ holds all data related to the operations performed by MSS I2C 1. A pointer to _g_mss_i2c1_ is passed as the first parameter to MSS I2C driver functions to indicate that MSS I2C 1 will perform the requested operation.
Functions

**MSS_I2C_init**

**Prototype**

```c
void MSS_I2C_init
{
    mss_i2c_instance_t * this_i2c,
    uint8_t ser_address,
    mss_i2c_clock_divider_t ser_clock_speed
};
```

**Description**

The `MSS_I2C_init()` function initializes and configures hardware and data structures of one of the SmartFusion MSS I2Cs.

**Parameters**

**this_i2c**

The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block to be initialized. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

**ser_address**

This parameter sets the I2C serial address being initialized. It is the I2C bus address to which the MSS I2C instance will respond. Any 8 bit address is allowed.

**ser_clock_speed**

This parameter sets the I2C serial clock frequency. It selects the divider that will be used to generate the serial clock from the APB clock. It can be one of the following:

- MSS_I2C_PCLK_DIV_256
- MSS_I2C_PCLK_DIV_224
- MSS_I2C_PCLK_DIV_192
- MSS_I2C_PCLK_DIV_160
- MSS_I2C_PCLK_DIV_960
- MSS_I2C_PCLK_DIV_120
- MSS_I2C_PCLK_DIV_60
- MSS_I2C_BCLK_DIV_8

**Return Value**

This function does not return a value.
**MSS_I2C_write**

**Prototype**

```c
void MSS_I2C_write
{
    mss_i2c_instance_t * this_i2c,
    uint8_t serial_addr,
    const uint8_t * write_buffer,
    uint16_t write_size,
    uint8_t options
}
```

**Description**

This function initiates an I2C master write transaction. This function returns immediately after initiating the transaction. The content of the write buffer passed as parameter will not be modified until the write transaction completes. It also means that the memory allocated for the write buffer will not be freed or go out of scope before the write completes. You can check for the write transaction completion using the `MSS_I2C_status()` function.

**Parameters**

**this_i2c**

The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

**serial_addr**

This parameter specifies the serial address of the target I2C device.

**write_buffer**

This parameter is a pointer to a buffer holding the data to be written to the target I2C device. Care must be taken not to release the memory used by this buffer before the write transaction completes. For example, it is not appropriate to return from a function allocating this buffer as an array variable before the write transaction completes as this would result in the buffer’s memory being de-allocated from the stack when the function returns. This memory could then be subsequently reused and modified causing unexpected data to be written to the target I2C device.

**write_size**

Number of bytes held in the `write_buffer` to be written to the target I2C device.

**Options**

The options parameter is used to indicate if the I2C bus should be released on completion of the write transaction. Using the `MSS_I2C_RELEASE_BUS` constant for the options parameter causes a STOP bit to be generated at the end of the write transaction causing the bus to be released for other I2C devices to use. Using the `MSS_I2C_HOLD_BUS` constant as options parameter prevents a STOP bit from being generated at the end of the write transaction, preventing other I2C devices from initiating a bus transaction.

**Return Value**

This function does not return a value.
MSS_I2C_read

Prototype

```c
void MSS_I2C_read
{
    mss_i2c_instance_t * this_i2c,
    uint8_t serial_addr,
    uint8_t * read_buffer,
    uint16_t read_size,
    uint8_t options
};
```

Description

This function initiates an I²C master read transaction. This function returns immediately after initiating the transaction.

The content of the read buffer passed as parameter will not be modified until the read transaction completes. It also means that the memory allocated for the read buffer will not be freed or go out of scope before the read completes. You can check for the read transaction completion using the MSS_I2C_status() function.

Parameters

**this_i2c**

The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I²C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I²C 0 and MSS I²C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I²C driver.

**serial_addr**

This parameter specifies the serial address of the target I²C device.

**read_buffer**

Pointer to a buffer where the data received from the target device will be stored.

Care must be taken not to release the memory used by this buffer before the read transaction completes. For example, it is not appropriate to return from a function allocating this buffer as an array variable before the read transaction completes as this would result in the buffer's memory being de-allocated from the stack when the function returns. This memory could then be subsequently reallocated resulting in the read transaction corrupting the newly allocated memory.

**read_size**

This parameter is the number of bytes to read from the target device. This size must not exceed the size of the `read_buffer` buffer.

Options

The options parameter is used to indicate if the I²C bus should be released on completion of the read transaction.

Using the MSS_I2C_RELEASE_BUS constant for the options parameter causes a STOP bit to be generated at the end of the read transaction causing the bus to be released for other I²C devices to use. Using the MSS_I2C_HOLD_BUS constant as options parameter prevents a STOP bit from being generated at the end of the read transaction, preventing other I²C devices from initiating a bus transaction.

Return Value

This function does not return a value.
MSS_I2C_write_read

Prototype

void MSS_I2C_write_read
{
    mss_i2c_instance_t * this_i2c,
    uint8_t serial_addr,
    const uint8_t * addr_offset,
    uint16_t offset_size,
    uint8_t * read_buffer,
    uint16_t read_size,
    uint8_t options
};

Description

This function initiates an I2C write-read transaction where data is first written to the target device before issuing a restart condition and changing the direction of the I2C transaction in order to read from the target device.

Parameters

this_i2c
The this_i2c parameter is a pointer to an mss_i2c_instance_t structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, g_mss_i2c0 and g_mss_i2c1, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the g_mss_i2c0 or g_mss_i2c1 global data structure defined within the I2C driver.

serial_addr
This parameter specifies the serial address of the target I2C device.

addr_offset
This parameter is a pointer to the buffer containing the data that will be sent to the slave during the write phase of the write-read transaction. This data is typically used to specify an address offset specifying to the I2C slave device what data it must return during the read phase of the write-read transaction.

offset_size
This parameter specifies the number of offset bytes to be written during the write phase of the write-read transaction. This is typically the size of the buffer pointed to by the addr_offset parameter.

read_buffer
This parameter is a pointer to the buffer where the data read from the I2C slave will be stored.

read_size
This parameter specifies the number of bytes to read from the target I2C slave device. This size must not exceed the size of the buffer pointed to by the read_buffer parameter.

Options

The options parameter is used to indicate if the I2C bus should be released on completion of the write-read transaction. Using the MSS_I2C_RELEASE_BUS constant for the options parameter causes a STOP bit to be generated at the end of the write-read transaction causing the bus to be released for other I2C devices to use. Using the MSS_I2C_HOLD_BUS constant as options parameter prevents a STOP bit from being generated at the end of the write-read transaction, preventing other I2C devices from initiating a bus transaction.
Return Value
This function does not return a value.

MSS_I2C_get_status

Prototype

```
mss_i2c_status_t MSS_I2C_get_status
(
    mss_i2c_instance_t * this_i2c
);
```

Description
This function indicates the current state of a MSS I2C instance.

Parameters

```
this_i2c
```

The *this_i2c* parameter is a pointer to an *mss_i2c_instance_t* structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, *g_mss_i2c0* and *g_mss_i2c1*, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the *g_mss_i2c0* or *g_mss_i2c1* global data structure defined within the I2C driver.

Return Value
The return value indicates the current state of a MSS I2C instance or the outcome of the previous transaction if no transaction is in progress. Possible return values are:

```
SUCCESS
```
The last I2C transaction has completed successfully.

```
IN_PROGRESS
```
There is an I2C transaction in progress.

```
FAILED
```
The last I2C transaction failed.
**MSS_I2C_wait_complete**

**Prototype**

```c
mss_i2c_status_t MSS_I2C_wait_complete
{
    mss_i2c_instance_t * this_i2c
};
```

**Description**

This function waits for the current I2C transaction to complete. The return value indicates whether the last I2C transaction was successful, or is still in progress, or failed.

**Parameters**

- **this_i2c**
  
The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

**Return Value**

The return value indicates the outcome of the last I2C transaction. It can be one of the following:

- **MSS_I2C_SUCCESS**
  
The last I2C transaction has completed successfully.

- **MSS_I2C_IN_PROGRESS**
  
The current I2C transaction is still in progress.

- **MSS_I2C_FAILED**
  
The last I2C transaction failed.
MSS_I2C_set_slave_tx_buffer

Prototype

```c
void MSS_I2C_set_slave_tx_buffer
(
    mss_i2c_instance_t * this_i2c,
    uint8_t * tx_buffer,
    uint16_t tx_size
);
```

Description

This function specifies the memory buffer holding the data that will be sent to the I²C master when this MSS I²C instance is the target of an I²C read or write-read transaction.

Parameters

- **this_i2c**
  The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I²C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I²C 0 and MSS I²C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I²C driver.

- **tx_buffer**
  This parameter is a pointer to the memory buffer holding the data to be returned to the I²C master when this MSS I²C instance is the target of an I²C read or write-read transaction.

- **tx_size**
  Size of the transmit buffer pointed to by the `tx_buffer` parameter.

Return Value

This function does not return a value.
MSS_I2C_set_slave_rx_buffer

Prototype

```c
void MSS_I2C_set_slave_rx_buffer
{
    mss_i2c_instance_t * this_i2c,
    uint8_t * rx_buffer,
    uint16_t rx_size
};
```

Description

This function specifies the memory buffer that will be used by the MSS I2C instance to receive data when it is a slave. This buffer is the memory where data will be stored when the MSS I2C is the target of an I2C master write or write-read transaction (i.e. when it is the slave).

Parameters

this_i2c

The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

rx_buffer

This parameter is a pointer to the memory buffer allocated by the caller software to be used as a slave receive buffer.

rx_size

Size of the slave receive buffer. This is the amount of memory that is allocated to the buffer pointed to by `rx_buffer`.

Note: This buffer size will indirectly specify the maximum I2C write transaction length this MSS I2C instance can be the target of. This is because this MSS I2C instance will respond to further received bytes with a non-acknowledge bit (NACK) as soon as its receive buffer is full. This will cause the write transaction to fail.

Return Value

This function does not return a value.
### MSS_I2C_set_slave_mem_offset_length

**Prototype**

```c
void MSS_I2C_set_slave_mem_offset_length
(  
    mss_i2c_instance_t * this_i2c,  
    uint8_t offset_length
);
```

**Description**

This function is used as part of the configuration of a MSS I2C instance for operation as a slave supporting write-read transactions. It specifies the number of bytes expected as part of the write phase of a write-read transaction. The bytes received during the write phase of a write-read transaction will be interpreted as an offset into the slave's transmit buffer. This allows random access into the I2C slave transmit buffer from a remote I2C master.

**Parameters**

- **this_i2c**
  
  The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

- **offset_length**
  
  The `offset_length` parameter configures the number of bytes to be interpreted by the MSS I2C slave as a memory offset value during the write phase of write-read transactions.

**Return Value**

This function does not return a value.
**MSS_I2C_register_write_handler**

**Prototype**
```c
void MSS_I2C_register_write_handler(
    mss_i2c_instance_t * this_i2c,
    mss_i2c_slave_wr_handler_t handler
);
```

**Description**
Register the function that will be called to process the data written to this MSS I2C instance when it is the slave in an I2C write transaction.

**Note:** The write handler is not called as a result of a write-read transaction. The write data of a write read transaction is interpreted as an offset into the slave's transmit buffer and handled by the driver.

**Parameters**
- **this_i2c**
  The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

- **handler**
  Pointer to the function that will process the I2C write request.

**Return Value**
This function does not return a value.
**MSS_I2C_enable_slave_rx**

**Prototype**

```c
void MSS_I2C_enable_slave_rx
  (  
    mss_i2c_instance_t * this_i2c  
  );
```

**Description**

Enables the MSS I2C instance identified through the `this_i2c` parameter, to receive data when it is the target of an I2C read, write or write-read transaction.

**Parameters**

**this_i2c**

The `this_i2c` parameter is a pointer to an `mss_i2c_instance_t` structure identifying the MSS I2C hardware block that will perform the requested function. There are two such data structures, `g_mss_i2c0` and `g_mss_i2c1`, associated with MSS I2C 0 and MSS I2C 1 respectively. This parameter must point to either the `g_mss_i2c0` or `g_mss_i2c1` global data structure defined within the I2C driver.

**Return Value**

This function does not return a value.
Product Support

Actel backs its products with various support services including Customer Service, a Customer Technical Support Center, a web site, an FTP site, electronic mail, and worldwide sales offices. This appendix contains information about contacting Actel and using these support services.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From Northeast and North Central U.S.A., call 650.318.4480
From Southeast and Southwest U.S.A., call 650.318.4480
From South Central U.S.A., call 650.318.4434
From Northwest U.S.A., call 650.318.4434
From Canada, call 650.318.4480
From Europe, call 650.318.4252 or +44 (0) 1276 401 500
From Japan, call 650.318.4743
From the rest of the world, call 650.318.4743
Fax, from anywhere in the world 650.318.8044

Actel Customer Technical Support Center

Actel staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions. The Customer Technical Support Center spends a great deal of time creating application notes and answers to FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

Actel Technical Support

Visit the Actel Customer Support website (http://www.actel.com/support/search/default.aspx) for more information and support. Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on the Actel web site.

Website

You can browse a variety of technical and non-technical information on Actel’s home page, at http://www.actel.com/.

Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. Several ways of contacting the Center follow:

Email

You can communicate your technical questions to our email address and receive answers back by email, fax, or phone. Also, if you have design problems, you can email your design files to receive assistance. We constantly monitor the email account throughout the day. When sending your request to us, please be sure to include your full name, company name, and your contact information for efficient processing of your request.
The technical support email address is tech@actel.com.

**Phone**

Our Technical Support Center answers all calls. The center retrieves information, such as your name, company name, phone number and your question, and then issues a case number. The Center then forwards the information to a queue where the first available application engineer receives the data and returns your call. The phone hours are from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. The Technical Support numbers are:

**650.318.4460**  
**800.262.1060**

Customers needing assistance outside the US time zones can either contact technical support via email (tech@actel.com) or contact a local sales office. Sales office listings can be found at www.actel.com/company/contact/default.aspx.
Actel is the leader in low-power FPGAs and mixed-signal FPGAs and offers the most comprehensive portfolio of system and power management solutions. Power Matters. Learn more at http://www.actel.com.