**SMART ARM-based Microcontrollers**

# AT03250: SAM D/R/L/C I2C Master Mode (SERCOM I2C) Driver

**APPLICATION NOTE**

## Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's SERCOM I$^2$C module, for the transfer of data via an I$^2$C bus. The following driver API modes are covered by this manual:

- Master Mode Polled APIs
- Master Mode Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information
- Examples
- API Overview

## Table of Contents

Introduction....1

1. Software License....4

2. Prerequisites....5

3. Module Overview....6

3.1. Driver Feature Macro Definition....6
3.2. Functional Description....6
3.3. Bus Topology....7
3.4. Transactions....7
3.4.1. Address Packets....7
3.4.2. Data Packets....8
3.4.3. Transaction Examples....8
3.4.4. Packet Timeout....8
3.4.5. Repeated Start....8
3.5. Multi Master....8
3.5.1. Arbitration....9
3.5.2. Clock Synchronization....9
3.6. Bus States....9
3.7. Bus Timing....10
3.7.1. Unknown Bus State Timeout....10
3.7.2. SDA Hold Timeout....10
3.8. Operation in Sleep Modes....10

4. Special Considerations....12

4.1. Interrupt-driven Operation....12

5. Extra Information....13

6. Examples....14

7. API Overview....15

7.1. Structure Definitions....15
7.1.1. Struct i2c_master_config....15
7.1.2. Struct i2c_master_module....16
7.1.3. Struct i2c_master_packet....16
7.2. Macro Definitions....16
7.2.1. Driver Feature Definition....16
7.3. Function Definitions....17
7.3.1. Lock/Unlock....17
7.3.2. Configuration and Initialization....18
7.3.3. Read and Write....20
7.3.4. SERCOM I$^2$C Master with DMA Interfaces....25
7.3.5. Callbacks....26
7.3.6. Read and Write, Interrupt-driven....27

7.4. Enumeration Definitions........ 31
7.4.1. Enum i2c_master_baud_rate........ 31
7.4.2. Enum i2c_master_callback........ 32
7.4.3. Enum i2c_master_inactive_timeout........ 32
7.4.4. Enum i2c_master_interrupt_flag........ 32
7.4.5. Enum i2c_master_start_hold_time........ 32
7.4.6. Enum i2c_master_transfer_speed........ 33
7.4.7. Enum i2c_transfer_direction........ 33

8. Extra Information for SERCOM I$^2$C Driver........ 34
8.1. Acronyms........ 34
8.2. Dependencies........ 34
8.3. Errata........ 34
8.4. Module History........ 34

9. Examples for SERCOM I$^2$C Driver........ 35
9.1. Quick Start Guide for SERCOM I$^2$C Master - Basic........ 35
9.1.1. Prerequisites........ 35
9.1.2. Setup........ 35
9.1.3. Implementation........ 37
9.2. Quick Start Guide for SERCOM I$^2$C Master - Callback........ 38
9.2.1. Prerequisites........ 38
9.2.2. Setup........ 38
9.2.3. Implementation........ 40
9.2.4. Callback........ 41
9.3. Quick Start Guide for Using DMA with SERCOM I$^2$C Master........ 41
9.3.1. Prerequisites........ 41
9.3.2. Setup........ 41
9.3.3. Implementation........ 44

10. Document Revision History........ 46

## 1. Software License

## 2. Prerequisites

There are no prerequisites.

# 3. Module Overview

The outline of this section is as follows:

- Driver Feature Macro Definition
- Functional Description
- Bus Topology
- Transactions
- Multi Master
- Bus States
- Bus Timing
- Operation in Sleep Modes

## 3.1. Driver Feature Macro Definition

| Driver Feature Macro | Supported devices |
|---|---|
| FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED | SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21 |
| FEATURE_I2C_10_BIT_ADDRESS | SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21 |
| FEATURE_I2C_SCL_STRETCH_MODE | SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21 |
| FEATURE_I2C_SCL_EXTEND_TIMEOUT | SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21 |

**Note:** The specific features are only available in the driver when the selected device supports those features.

## 3.2. Functional Description

The $I^2C$ provides a simple two-wire bidirectional bus consisting of a wired-AND type serial clock line (SCL) and a wired-AND type serial data line (SDA).

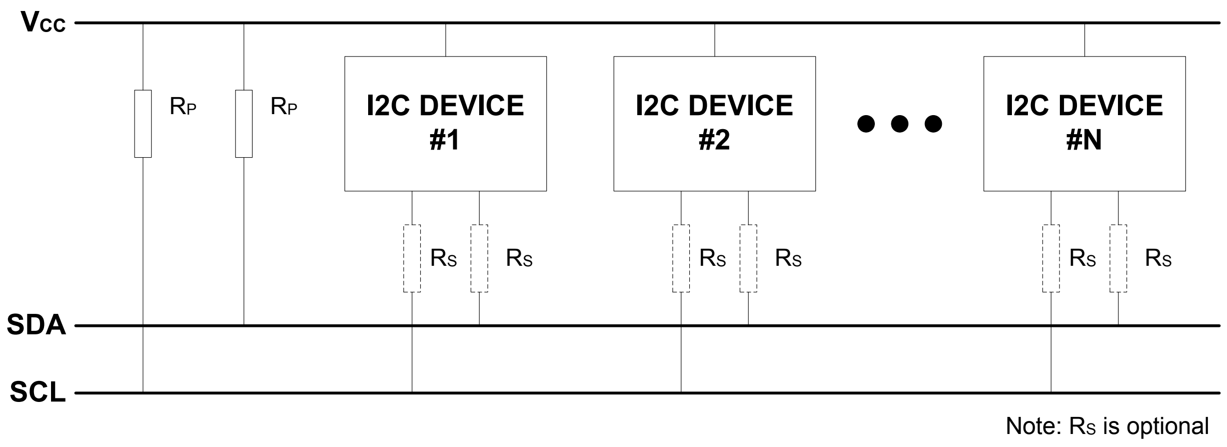
The $I^2C$ bus provides a simple, but efficient method of interconnecting multiple master and slave devices. An arbitration mechanism is provided for resolving bus ownership between masters, as only one master device may own the bus at any given time. The arbitration mechanism relies on the wired-AND connections to avoid bus drivers short-circuiting.

A unique address is assigned to all slave devices connected to the bus. A device can contain both master and slave logic, and can emulate multiple slave devices by responding to more than one address.

## 3.3. Bus Topology

The I$^2$C bus topology is illustrated in Figure 3-1 I2C Bus Topology on page 7. The pull-up resistors (Rs) will provide a high level on the bus lines when none of the I$^2$C devices are driving the bus. These are optional, and can be replaced with a constant current source.

**Figure 3-1 I$^2$C Bus Topology**



## 3.4. Transactions

The I$^2$C standard defines three fundamental transaction formats:

- Master Write
  - The master transmits data packets to the slave after addressing it
- Master Read
  - The slave transmits data packets to the master after being addressed
- Combined Read/Write
  - A combined transaction consists of several write and read transactions

A data transfer starts with the master issuing a **Start** condition on the bus, followed by the address of the slave together with a bit to indicate whether the master wants to read from or write to the slave. The addressed slave must respond to this by sending an **ACK** back to the master.

After this, data packets are sent from the master or slave, according to the read/write bit. Each packet must be acknowledged (ACK) or not acknowledged (NACK) by the receiver.

If a slave responds with a NACK, the master must assume that the slave cannot receive any more data and cancel the write operation.

The master completes a transaction by issuing a **Stop** condition.

A master can issue multiple **Start** conditions during a transaction; this is then called a **Repeated Start** condition.

### 3.4.1. Address Packets

The slave address consists of seven bits. The 8th bit in the transfer determines the data direction (read or write). An address packet always succeeds a **Start** or **Repeated Start** condition. The 8th bit is handled in the driver, and the user will only have to provide the 7-bit address.

### 3.4.2. Data Packets

Data packets are nine bits long, consisting of one 8-bit data byte, and an acknowledgement bit. Data packets follow either an address packet or another data packet on the bus.

### 3.4.3. Transaction Examples

The gray bits in the following examples are sent from master to slave, and the white bits are sent from slave to master. Example of a read transaction is shown in Figure 3-2 I2C Packet Read on page 8. Here, the master first issues a **Start** condition and gets ownership of the bus. An address packet with the direction flag set to read is then sent and acknowledged by the slave. Then the slave sends one data packet which is acknowledged by the master. The slave sends another packet, which is not acknowledged by the master and indicates that the master will terminate the transaction. In the end, the transaction is terminated by the master issuing a **Stop** condition.

**Figure 3-2 I$^2$C Packet Read**

| Bit 0 | Bit 1 | Bit 2 | Bit 3 | Bit 4 | Bit 5 | Bit 6 | Bit 7 | Bit 8 | Bit 9 | Bit 10 | Bit 11 | Bit 12 | Bit 13 | Bit 14 | Bit 15 | Bit 16 | Bit 17 | Bit 18 | Bit 19 | Bit 20 | Bit 21 | Bit 22 | Bit 23 | Bit 24 | Bit 25 | Bit 26 | Bit 27 | Bit 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| START | ADDRESS | | | | | | | READ | ACK | DATA | | | | | | | | ACK | DATA | | | | | | | | NACK | STOP |

Example of a write transaction is shown in Figure 3-3 I2C Packet Write on page 8. Here, the master first issues a **Start** condition and gets ownership of the bus. An address packet with the dir flag set to write is then sent and acknowledged by the slave. Then the master sends two data packets, each acknowledged by the slave. In the end, the transaction is terminated by the master issuing a **Stop** condition.

**Figure 3-3 I$^2$C Packet Write**

| Bit 0 | Bit 1 | Bit 2 | Bit 3 | Bit 4 | Bit 5 | Bit 6 | Bit 7 | Bit 8 | Bit 9 | Bit 10 | Bit 11 | Bit 12 | Bit 13 | Bit 14 | Bit 15 | Bit 16 | Bit 17 | Bit 18 | Bit 19 | Bit 20 | Bit 21 | Bit 22 | Bit 23 | Bit 24 | Bit 25 | Bit 26 | Bit 27 | Bit 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| START | ADDRESS | | | | | | | WRITE | ACK | DATA | | | | | | | | ACK | DATA | | | | | | | | ACK | STOP |

### 3.4.4. Packet Timeout

When a master sends an I$^2$C packet, there is no way of being sure that a slave will acknowledge the packet. To avoid stalling the device forever while waiting for an acknowledge, a user selectable timeout is provided in the i2c_master_config struct which lets the driver exit a read or write operation after the specified time. The function will then return the STATUS_ERR_TIMEOUT flag.

This is also the case for the slave when using the functions postfixed `_wait`.

The time before the timeout occurs, will be the same as for unknown bus state timeout.

### 3.4.5. Repeated Start

To issue a **Repeated Start**, the functions postfixed `_no_stop` must be used. These functions will not send a **Stop** condition when the transfer is done, thus the next transfer will start with a **Repeated Start**. To end the transaction, the functions without the `_no_stop` postfix must be used for the last read/write.

## 3.5. Multi Master

In a multi master environment, arbitration of the bus is important, as only one master can own the bus at any point.

### 3.5.1. Arbitration

| | |
|---|---|
| **Clock stretching** | The serial clock line is always driven by a master device. However, all devices connected to the bus are allowed stretch the low period of the clock to slow down the overall clock frequency or to insert wait states while processing data. Both master and slave can randomly stretch the clock, which will force the other device into a wait-state until the clock line goes high again. |
| **Arbitration on the data line** | If two masters start transmitting at the same time, they will both transmit until one master detects that the other master is pulling the data line low. When this is detected, the master not pulling the line low, will stop the transmission and wait until the bus is idle. As it is the master trying to contact the slave with the lowest address that will get the bus ownership, this will create an arbitration scheme always prioritizing the slaves with the lowest address in case of a bus collision. |

### 3.5.2. Clock Synchronization

In situations where more than one master is trying to control the bus clock line at the same time, a clock synchronization algorithm based on the same principles used for clock stretching is necessary.

## 3.6. Bus States

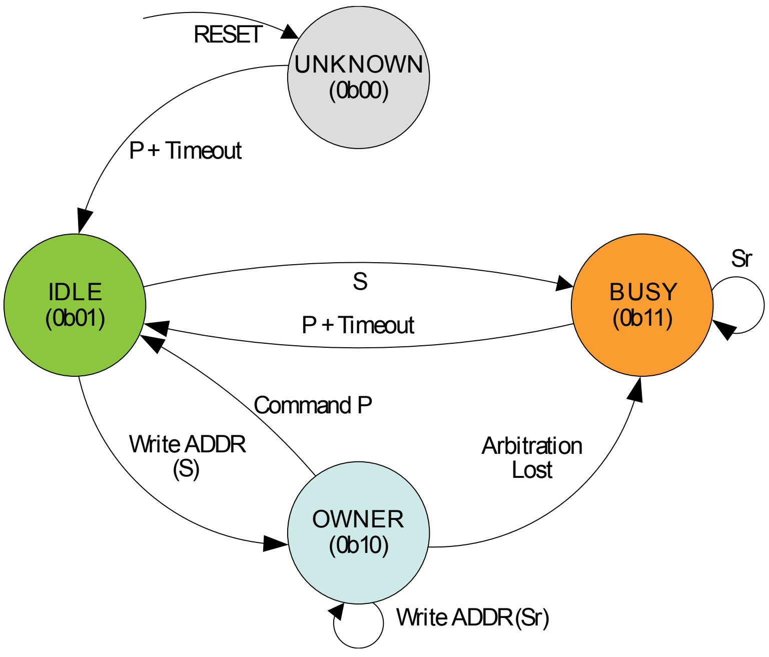
As the I$^2$C bus is limited to one transaction at the time, a master that wants to perform a bus transaction must wait until the bus is free. Because of this, it is necessary for all masters in a multi-master system to know the current status of the bus to be able to avoid conflicts and to ensure data integrity.

- **IDLE** No activity on the bus (between a **Stop** and a new **Start** condition)
- **OWNER** If the master initiates a transaction successfully
- **BUSY** If another master is driving the bus
- **UNKNOWN** If the master has recently been enabled or connected to the bus. Is forced to **IDLE** after given timeout when the master module is enabled

The bus state diagram can be seen in Figure 3-4 I2C Bus State Diagram on page 10.

- S: Start condition
- P: Stop condition
- Sr: Repeated start condition

**Figure 3-4  I²C Bus State Diagram**



## 3.7. Bus Timing

Inactive bus timeout for the master and SDA hold time is configurable in the drivers.

### 3.7.1. Unknown Bus State Timeout

When a master is enabled or connected to the bus, the bus state will be unknown until either a given timeout or a stop command has occurred. The timeout is configurable in the i2c_master_config struct. The timeout time will depend on toolchain and optimization level used, as the timeout is a loop incrementing a value until it reaches the specified timeout value.

### 3.7.2. SDA Hold Timeout

When using the I²C in slave mode, it will be important to set a SDA hold time which assures that the master will be able to pick up the bit sent from the slave. The SDA hold time makes sure that this is the case by holding the data line low for a given period after the negative edge on the clock.

The SDA hold time is also available for the master driver, but is not a necessity.

## 3.8. Operation in Sleep Modes

The I²C module can operate in all sleep modes by setting the run_in_standby Boolean in the i2c_master_config or i2c_slave_config struct. The operation in slave and master mode is shown in Table 3-1  I2C Standby Operations on page 11.

**Table 3-1 I²C Standby Operations**

| Run in standby | Slave | Master |
|---|---|---|
| false | Disabled, all reception is dropped | Generic Clock (GCLK) disabled when master is idle |
| true | Wake on address match when enabled | GCLK enabled while in sleep modes |

# 4. Special Considerations

## 4.1. Interrupt-driven Operation

While an interrupt-driven operation is in progress, subsequent calls to a write or read operation will return the STATUS_BUSY flag, indicating that only one operation is allowed at any given time.

To check if another transmission can be initiated, the user can either call another transfer operation, or use the i2c_master_get_job_status/i2c_slave_get_job_status functions depending on mode.

If the user would like to get callback from operations while using the interrupt-driven driver, the callback must be registered and then enabled using the "register_callback" and "enable_callback" functions.

## 5. Extra Information

For extra information, see Extra Information for SERCOM I2C Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

## 6. Examples

For a list of examples related to this driver, see Examples for SERCOM I2C Driver.

# 7. API Overview

## 7.1. Structure Definitions

### 7.1.1. Struct i2c_master_config

This is the configuration structure for the I$^2$C Master device. It is used as an argument for i2c_master_init to provide the desired configurations for the module. The structure should be initialized using the i2c_master_get_config_defaults.

**Table 7-1 Members**

| Type | Name | Description |
|---|---|---|
| uint32_t | baud_rate | Baud rate (in KHz) for I$^2$C operations in standard-mode, Fast-mode, and Fast-mode Plus Transfers, i2c_master_baud_rate |
| uint32_t | baud_rate_high_speed | Baud rate (in KHz) for I$^2$C operations in High-speed mode, i2c_master_baud_rate |
| uint16_t | buffer_timeout | Timeout for packet write to wait for slave |
| enum gclk_generator | generator_source | GCLK generator to use as clock source |
| enum i2c_master_inactive_timeout | inactive_timeout | Inactive bus time out |
| bool | master_scl_low_extend_timeout | Set to enable maser SCL low extend time-out |
| uint32_t | pinmux_pad0 | PAD0 (SDA) pinmux |
| uint32_t | pinmux_pad1 | PAD1 (SCL) pinmux |
| bool | run_in_standby | Set to keep module active in sleep modes |
| bool | scl_low_timeout | Set to enable SCL low time-out |
| bool | scl_stretch_only_after_ack_bit | Set to enable SCL stretch only after ACK bit (required for high speed) |
| uint16_t | sda_scl_rise_time_ns | Get more accurate BAUD, considering rise time(required for standard-mode and Fast-mode) |
| bool | slave_scl_low_extend_timeout | Set to enable slave SCL low extend time-out |

| Type | Name | Description |
|---|---|---|
| enum i2c_master_start_hold_time | start_hold_time | Bus hold time after start signal on data line |
| enum i2c_master_transfer_speed | transfer_speed | Transfer speed mode |
| uint16_t | unknown_bus_state_timeout | Unknown bus state timeout |

### 7.1.2. Struct i2c_master_module

SERCOM I$^2$C Master driver software instance structure, used to retain software state information of an associated hardware module instance.

**Note:** The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

### 7.1.3. Struct i2c_master_packet

Structure to be used when transferring I$^2$C master packets.

**Table 7-2 Members**

| Type | Name | Description |
|---|---|---|
| uint16_t | address | Address to slave device |
| uint8_t * | data | Data array containing all data to be transferred |
| uint16_t | data_length | Length of data array |
| bool | high_speed | Use high speed transfer. Set to false if the feature is not supported by the device |
| uint8_t | hs_master_code | High speed mode master code (0000 1XXX), valid when high_speed is true |
| bool | ten_bit_address | Use 10-bit addressing. Set to false if the feature is not supported by the device |

## 7.2. Macro Definitions

### 7.2.1. Driver Feature Definition

Define SERCOM I$^2$C driver features set according to different device family.

#### 7.2.1.1. Macro FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED

```
#define FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED
```

Fast mode plus and high speed support.

#### 7.2.1.2. Macro FEATURE_I2C_10_BIT_ADDRESS

```
#define FEATURE_I2C_10_BIT_ADDRESS
```

10-bit address support

#### 7.2.1.3. Macro FEATURE_I2C_SCL_STRETCH_MODE

```
#define FEATURE_I2C_SCL_STRETCH_MODE
```

SCL stretch mode support

#### 7.2.1.4. Macro FEATURE_I2C_SCL_EXTEND_TIMEOUT

```
#define FEATURE_I2C_SCL_EXTEND_TIMEOUT
```

SCL extend timeout support

#### 7.2.1.5. Macro FEATURE_I2C_DMA_SUPPORT

```
#define FEATURE_I2C_DMA_SUPPORT
```

## 7.3. Function Definitions

### 7.3.1. Lock/Unlock

#### 7.3.1.1. Function i2c_master_lock()

Attempt to get lock on driver instance.

```
enum status_code i2c_master_lock(
        struct i2c_master_module *const module)
```

This function checks the instance's lock, which indicates whether or not it is currently in use, and sets the lock if it was not already set.

The purpose of this is to enable exclusive access to driver instances, so that, e.g., transactions by different services will not interfere with each other.

**Table 7-3 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the driver instance to lock |

**Table 7-4 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | If the module was locked |
| STATUS_BUSY | If the module was already locked |

#### 7.3.1.2. Function i2c_master_unlock()

Unlock driver instance.

```
void i2c_master_unlock(
        struct i2c_master_module *const module)
```

This function clears the instance lock, indicating that it is available for use.

**Table 7-5 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the driver instance to lock |

**Table 7-6 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | If the module was locked |
| STATUS_BUSY | If the module was already locked |

## 7.3.2. Configuration and Initialization

### 7.3.2.1. Function i2c_master_is_syncing()

Returns the synchronization status of the module.

```
bool i2c_master_is_syncing(
        const struct i2c_master_module *const module)
```

Returns the synchronization status of the module.

**Table 7-7 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in]** | module | Pointer to software module structure |

**Returns**
Status of the synchronization.

**Table 7-8 Return Values**

| Return value | Description |
|---|---|
| true | Module is busy synchronizing |
| false | Module is not synchronizing |

### 7.3.2.2. Function i2c_master_get_config_defaults()

Gets the I2C master default configurations.

```
void i2c_master_get_config_defaults(
        struct i2c_master_config *const config)
```

Use to initialize the configuration structure to known default values.

The default configuration is as follows:

- Baudrate 100KHz
- GCLK generator 0
- Do not run in standby
- Start bit hold time 300ns - 600ns
- Buffer timeout = 65535

- Unknown bus status timeout = 65535
- Do not run in standby
- PINMUX_DEFAULT for SERCOM pads

Those default configuration only available if the device supports it:

- High speed baudrate 3.4MHz
- Standard-mode and Fast-mode transfer speed
- SCL stretch disabled
- Slave SCL low extend time-out disabled
- Master SCL low extend time-out disabled

**Table 7-9 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[out]** | config | Pointer to configuration structure to be initiated |

### 7.3.2.3. Function i2c_master_init()

Initializes the requested I2C hardware module.

```
enum status_code i2c_master_init(
        struct i2c_master_module *const module,
        Sercom *const hw,
        const struct i2c_master_config *const config)
```

Initializes the SERCOM I$^2$C master device requested and sets the provided software module struct. Run this function before any further use of the driver.

**Table 7-10 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[out]** | module | Pointer to software module struct |
| **[in]** | hw | Pointer to the hardware instance |
| **[in]** | config | Pointer to the configuration struct |

**Returns**
Status of initialization.

**Table 7-11 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | Module initiated correctly |
| STATUS_ERR_DENIED | If module is enabled |
| STATUS_BUSY | If module is busy resetting |
| STATUS_ERR_ALREADY_INITIALIZED | If setting other GCLK generator than previously set |
| STATUS_ERR_BAUDRATE_UNAVAILABLE | If given baudrate is not compatible with set GCLK frequency |

#### 7.3.2.4. Function i2c_master_enable()

Enables the I2C module.

```
void i2c_master_enable(
        const struct i2c_master_module *const module)
```

Enables the requested I$^2$C module and set the bus state to IDLE after the specified timeout period if no stop bit is detected.

**Table 7-12 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in]** | module | Pointer to the software module struct |

#### 7.3.2.5. Function i2c_master_disable()

Disable the I2C module.

```
void i2c_master_disable(
        const struct i2c_master_module *const module)
```

Disables the requested I$^2$C module.

**Table 7-13 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in]** | module | Pointer to the software module struct |

#### 7.3.2.6. Function i2c_master_reset()

Resets the hardware module.

```
void i2c_master_reset(
        struct i2c_master_module *const module)
```

Reset the module to hardware defaults.

**Table 7-14 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module structure |

### 7.3.3. Read and Write

#### 7.3.3.1. Function i2c_master_read_packet_wait()

Reads data packet from slave.

```
enum status_code i2c_master_read_packet_wait(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I$^2$C bus and sends a stop condition when finished.

**Note:** This will stall the device from any other operation. For interrupt-driven operation, see i2c_master_read_packet_job.

**Table 7-15 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of reading packet.

**Table 7-16 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | The packet was read successfully |
| STATUS_ERR_TIMEOUT | If no response was given within specified timeout period |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |

#### 7.3.3.2. Function i2c_master_read_packet_wait_no_stop()

Reads data packet from slave without sending a stop condition when done.

```
enum status_code i2c_master_read_packet_wait_no_stop(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I$^2$C bus without sending a stop condition when done, thus retaining ownership of the bus when done. To end the transaction, a read or write with stop condition must be performed.

**Note:** This will stall the device from any other operation. For interrupt-driven operation, see i2c_master_read_packet_job.

**Table 7-17 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of reading packet.

Table 7-18 Return Values

| Return value | Description |
|---|---|
| STATUS_OK | The packet was read successfully |
| STATUS_ERR_TIMEOUT | If no response was given within specified timeout period |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |

### 7.3.3.3. Function i2c_master_write_packet_wait()

Writes data packet to slave.

```
enum status_code i2c_master_write_packet_wait(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I$^2$C bus and sends a stop condition when finished.

**Note:** This will stall the device from any other operation. For interrupt-driven operation, see i2c_master_read_packet_job.

Table 7-19 Parameters

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of write packet.

Table 7-20 Return Values

| Return value | Description |
|---|---|
| STATUS_OK | If packet was write successfully |
| STATUS_BUSY | If master module is busy with a job |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |
| STATUS_ERR_TIMEOUT | If timeout occurred |
| STATUS_ERR_OVERFLOW | If slave did not acknowledge last sent data, indicating that slave does not want more data and was not able to read last data sent |

### 7.3.3.4. Function i2c_master_write_packet_wait_no_stop()

Writes data packet to slave without sending a stop condition when done.

```
enum status_code i2c_master_write_packet_wait_no_stop(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I$^2$C bus without sending a stop condition, thus retaining ownership of the bus when done. To end the transaction, a read or write with stop condition or sending a stop with the i2c_master_send_stop function must be performed.

**Note:** This will stall the device from any other operation. For interrupt-driven operation, see i2c_master_read_packet_job.

**Table 7-21 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of write packet.

**Table 7-22 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | If packet was write successfully |
| STATUS_BUSY | If master module is busy |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |
| STATUS_ERR_TIMEOUT | If timeout occurred |
| STATUS_ERR_OVERFLOW | If slave did not acknowledge last sent data, indicating that slave do not want more data |

### 7.3.3.5. Function i2c_master_send_stop()

Sends stop condition on bus.

```
void i2c_master_send_stop(
        struct i2c_master_module *const module)
```

Sends a stop condition on bus.

**Note:** This function can only be used after the i2c_master_write_packet_wait_no_stop function. If a stop condition is to be sent after a read, the i2c_master_read_packet_wait function must be used.

**Table 7-23 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the software instance struct |

### 7.3.3.6. Function i2c_master_send_nack()

Sends nack signal on bus.

```
void i2c_master_send_nack(
        struct i2c_master_module *const module)
```

Sends a nack signal on bus.

**Note:** This function can only be used after the i2c_master_write_packet_wait_no_nack function, or i2c_master_read_byte function.

**Table 7-24 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the software instance struct |

### 7.3.3.7. Function i2c_master_read_byte()

Reads one byte data from slave.

```
enum status_code i2c_master_read_byte(
        struct i2c_master_module *const module,
        uint8_t * byte)
```

**Table 7-25 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[out]** | byte | Read one byte data to slave |

**Returns**
Status of reading byte.

**Table 7-26 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | One byte was read successfully |
| STATUS_ERR_TIMEOUT | If no response was given within specified timeout period |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |

#### 7.3.3.8. Function i2c_master_write_byte()

Write one byte data to slave.

```
enum status_code i2c_master_write_byte(
        struct i2c_master_module *const module,
        uint8_t byte)
```

**Table 7-27 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in]** | byte | Send one byte data to slave |

**Returns**
Status of writing byte.

**Table 7-28 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | One byte was write successfully |
| STATUS_ERR_TIMEOUT | If no response was given within specified timeout period |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |

#### 7.3.3.9. Function i2c_master_read_packet_wait_no_nack()

```
enum status_code i2c_master_read_packet_wait_no_nack(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

### 7.3.4. SERCOM I²C Master with DMA Interfaces

#### 7.3.4.1. Function i2c_master_dma_set_transfer()

Set I2C for DMA transfer with slave address and transfer size.

```
void i2c_master_dma_set_transfer(
        struct i2c_master_module *const module,
        uint16_t addr,
        uint8_t length,
        enum i2c_transfer_direction direction)
```

This function will set the slave address, transfer size and enable the auto transfer mode for DMA.

**Table 7-29 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the driver instance to lock |
| **[in]** | addr | I$^2$C slave address |
| **[in]** | length | I$^2$C transfer length with DMA |
| **[in]** | direction | I$^2$C transfer direction |

### 7.3.5. Callbacks

#### 7.3.5.1. Function i2c_master_register_callback()

Registers callback for the specified callback type.

```
void i2c_master_register_callback(
        struct i2c_master_module *const module,
        i2c_master_callback_t callback,
        enum i2c_master_callback callback_type)
```

Associates the given callback function with the specified callback type.

To enable the callback, the i2c_master_enable_callback function must be used.

**Table 7-30 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the software module struct |
| **[in]** | callback | Pointer to the function desired for the specified callback |
| **[in]** | callback_type | Callback type to register |

#### 7.3.5.2. Function i2c_master_unregister_callback()

Unregisters callback for the specified callback type.

```
void i2c_master_unregister_callback(
        struct i2c_master_module *const module,
        enum i2c_master_callback callback_type)
```

When called, the currently registered callback for the given callback type will be removed.

**Table 7-31 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the software module struct |
| **[in]** | callback_type | Specifies the callback type to unregister |

#### 7.3.5.3. Function i2c_master_enable_callback()

Enables callback.

```
void i2c_master_enable_callback(
        struct i2c_master_module *const module,
        enum i2c_master_callback callback_type)
```

Enables the callback specified by the callback_type.

**Table 7-32  Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the software module struct |
| **[in]** | callback_type | Callback type to enable |

#### 7.3.5.4. Function i2c_master_disable_callback()

Disables callback.

```
void i2c_master_disable_callback(
        struct i2c_master_module *const module,
        enum i2c_master_callback callback_type)
```

Disables the callback specified by the callback_type.

**Table 7-33  Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to the software module struct |
| **[in]** | callback_type | Callback type to disable |

### 7.3.6. Read and Write, Interrupt-driven

#### 7.3.6.1. Function i2c_master_read_bytes()

```
enum status_code i2c_master_read_bytes(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

#### 7.3.6.2. Function i2c_master_read_packet_job()

Initiates a read packet operation.

```
enum status_code i2c_master_read_packet_job(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I$^2$C bus. This is the non-blocking equivalent of i2c_master_read_packet_wait.

Table 7-34 Parameters

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of starting reading I$^2$C packet.

Table 7-35 Return Values

| Return value | Description |
|---|---|
| STATUS_OK | If reading was started successfully |
| STATUS_BUSY | If module is currently busy with another transfer |

#### 7.3.6.3. Function i2c_master_read_packet_job_no_stop()

Initiates a read packet operation without sending a STOP condition when done.

```
enum status_code i2c_master_read_packet_job_no_stop(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I$^2$C bus without sending a stop condition, thus retaining ownership of the bus when done. To end the transaction, a read or write with stop condition must be performed.

This is the non-blocking equivalent of i2c_master_read_packet_wait_no_stop.

Table 7-36 Parameters

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of starting reading I$^2$C packet.

Table 7-37 Return Values

| Return value | Description |
|---|---|
| STATUS_OK | If reading was started successfully |
| STATUS_BUSY | If module is currently busy with another operation |

### 7.3.6.4. Function i2c_master_read_packet_job_no_nack()

Initiates a read packet operation without sending a NACK signal and a STOP condition when done.

```
enum status_code i2c_master_read_packet_job_no_nack(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I²C bus without sending a nack and a stop condition, thus retaining ownership of the bus when done. To end the transaction, a read or write with stop condition must be performed.

This is the non-blocking equivalent of i2c_master_read_packet_wait_no_stop.

**Table 7-38 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I²C packet to transfer |

**Returns**
Status of starting reading I²C packet.

**Table 7-39 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | If reading was started successfully |
| STATUS_BUSY | If module is currently busy with another operation |

### 7.3.6.5. Function i2c_master_write_bytes()

```
enum status_code i2c_master_write_bytes(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

### 7.3.6.6. Function i2c_master_write_packet_job()

Initiates a write packet operation.

```
enum status_code i2c_master_write_packet_job(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I²C bus. This is the non-blocking equivalent of i2c_master_write_packet_wait.

**Table 7-40 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I²C packet to transfer |

**Returns**
Status of starting writing I$^2$C packet job.

**Table 7-41 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | If writing was started successfully |
| STATUS_BUSY | If module is currently busy with another transfer |

### 7.3.6.7. Function i2c_master_write_packet_job_no_stop()

Initiates a write packet operation without sending a STOP condition when done.

```
enum status_code i2c_master_write_packet_job_no_stop(
        struct i2c_master_module *const module,
        struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I$^2$C bus without sending a stop condition, thus retaining ownership of the bus when done. To end the transaction, a read or write with stop condition or sending a stop with the i2c_master_send_stop function must be performed.

This is the non-blocking equivalent of i2c_master_write_packet_wait_no_stop.

**Table 7-42 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module struct |
| **[in, out]** | packet | Pointer to I$^2$C packet to transfer |

**Returns**
Status of starting writing I$^2$C packet job.

**Table 7-43 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | If writing was started successfully |
| STATUS_BUSY | If module is currently busy with another |

### 7.3.6.8. Function i2c_master_cancel_job()

Cancel any currently ongoing operation.

```
void i2c_master_cancel_job(
        struct i2c_master_module *const module)
```

Terminates the running transfer operation.

**Table 7-44 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in, out]** | module | Pointer to software module structure |

### 7.3.6.9. Function i2c_master_get_job_status()

Get status from ongoing job.

```
enum status_code i2c_master_get_job_status(
        struct i2c_master_module *const module)
```

Will return the status of a transfer operation.

**Table 7-45 Parameters**

| Data direction | Parameter name | Description |
|---|---|---|
| **[in]** | module | Pointer to software module structure |

**Returns**
Last status code from transfer operation.

**Table 7-46 Return Values**

| Return value | Description |
|---|---|
| STATUS_OK | No error has occurred |
| STATUS_BUSY | If transfer is in progress |
| STATUS_BUSY | If master module is busy |
| STATUS_ERR_DENIED | If error on bus |
| STATUS_ERR_PACKET_COLLISION | If arbitration is lost |
| STATUS_ERR_BAD_ADDRESS | If slave is busy, or no slave acknowledged the address |
| STATUS_ERR_TIMEOUT | If timeout occurred |
| STATUS_ERR_OVERFLOW | If slave did not acknowledge last sent data, indicating that slave does not want more data and was not able to read |

## 7.4. Enumeration Definitions

### 7.4.1. Enum i2c_master_baud_rate

Values for I$^2$C speeds supported by the module. The driver will also support setting any other value, in which case set the value in the i2c_master_config at desired value divided by 1000.

Example: If 10KHz operation is required, give baud_rate in the configuration structure the value 10.

**Table 7-47 Members**

| Enum value | Description |
|---|---|
| I2C_MASTER_BAUD_RATE_100KHZ | Baud rate at 100KHz (Standard-mode) |
| I2C_MASTER_BAUD_RATE_400KHZ | Baud rate at 400KHz (Fast-mode) |

| Enum value | Description |
|---|---|
| I2C_MASTER_BAUD_RATE_1000KHZ | Baud rate at 1MHz (Fast-mode Plus) |
| I2C_MASTER_BAUD_RATE_3400KHZ | Baud rate at 3.4MHz (High-speed mode) |

### 7.4.2. Enum i2c_master_callback

The available callback types for the I$^2$C master module.

**Table 7-48 Members**

| Enum value | Description |
|---|---|
| I2C_MASTER_CALLBACK_WRITE_COMPLETE | Callback for packet write complete |
| I2C_MASTER_CALLBACK_READ_COMPLETE | Callback for packet read complete |
| I2C_MASTER_CALLBACK_ERROR | Callback for error |

### 7.4.3. Enum i2c_master_inactive_timeout

If the inactive bus time-out is enabled and the bus is inactive for longer than the time-out setting, the bus state logic will be set to idle.

**Table 7-49 Members**

| Enum value | Description |
|---|---|
| I2C_MASTER_INACTIVE_TIMEOUT_DISABLED | Inactive bus time-out disabled |
| I2C_MASTER_INACTIVE_TIMEOUT_55US | Inactive bus time-out 5-6 SCL cycle time-out |
| I2C_MASTER_INACTIVE_TIMEOUT_105US | Inactive bus time-out 10-11 SCL cycle time-out |
| I2C_MASTER_INACTIVE_TIMEOUT_205US | Inactive bus time-out 20-21 SCL cycle time-out |

### 7.4.4. Enum i2c_master_interrupt_flag

Flags used when reading or setting interrupt flags.

**Table 7-50 Members**

| Enum value | Description |
|---|---|
| I2C_MASTER_INTERRUPT_WRITE | Interrupt flag used for write |
| I2C_MASTER_INTERRUPT_READ | Interrupt flag used for read |

### 7.4.5. Enum i2c_master_start_hold_time

Values for the possible I$^2$C master mode SDA internal hold times after start bit has been sent.

**Table 7-51 Members**

| Enum value | Description |
|---|---|
| I2C_MASTER_START_HOLD_TIME_DISABLED | Internal SDA hold time disabled |
| I2C_MASTER_START_HOLD_TIME_50NS_100NS | Internal SDA hold time 50ns - 100ns |
| I2C_MASTER_START_HOLD_TIME_300NS_600NS | Internal SDA hold time 300ns - 600ns |
| I2C_MASTER_START_HOLD_TIME_400NS_800NS | Internal SDA hold time 400ns - 800ns |

### 7.4.6. Enum i2c_master_transfer_speed

Enum for the transfer speed.

**Table 7-52 Members**

| Enum value | Description |
|---|---|
| I2C_MASTER_SPEED_STANDARD_AND_FAST | Standard-mode (Sm) up to 100KHz and Fast-mode (Fm) up to 400KHz |
| I2C_MASTER_SPEED_FAST_MODE_PLUS | Fast-mode Plus (Fm+) up to 1MHz |
| I2C_MASTER_SPEED_HIGH_SPEED | High-speed mode (Hs-mode) up to 3.4MHz |

### 7.4.7. Enum i2c_transfer_direction

For master: transfer direction or setting direction bit in address. For slave: direction of request from master.

**Table 7-53 Members**

| Enum value | Description |
|---|---|
| I2C_TRANSFER_WRITE | Master write operation is in progress |
| I2C_TRANSFER_READ | Master read operation is in progress |

# 8. Extra Information for SERCOM I$^2$C Driver

## 8.1. Acronyms

Table 8-1 Acronyms on page 34 is a table listing the acronyms used in this module, along with their intended meanings.

**Table 8-1 Acronyms**

| Acronym | Description |
|---|---|
| SDA | Serial Data Line |
| SCL | Serial Clock Line |
| SERCOM | Serial Communication Interface |
| DMA | Direct Memory Access |

## 8.2. Dependencies

The I$^2$C driver has the following dependencies:

- System Pin Multiplexer Driver

## 8.3. Errata

There are no errata related to this driver.

## 8.4. Module History

Table 8-2 Module History on page 34 is an overview of the module history, detailing enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version listed in Table 8-2 Module History on page 34.

**Table 8-2 Module History**

| Changelog |
|---|
| • Added 10-bit addressing and high speed support in SAM D21<br>• Separate structure i2c_packet into i2c_master_packet and i2c_slave packet |
| • Added support for SCL stretch and extended timeout hardware features in SAM D21<br>• Added fast mode plus support in SAM D21 |
| Fixed incorrect logical mask for determining if a bus error has occurred in I$^2$C Slave mode |
| Initial Release |

# 9. Examples for SERCOM I$^2$C Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM I2C Master Mode (SERCOM I2C) Driver. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for the I2C Master module - Basic Use Case
  - Quick Start Guide for the I2C Master module - Callback Use Case
- Quick Start Guide for the I2C Master module - DMA Use Case

## 9.1. Quick Start Guide for SERCOM I$^2$C Master - Basic

In this use case, the I$^2$C will used and set up as follows:

- Master mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value
- 65535 unknown bus state timeout value

### 9.1.1. Prerequisites

The device must be connected to an I$^2$C slave.

### 9.1.2. Setup

#### 9.1.2.1. Code

The following must be added to the user application:

- A sample buffer to send, a sample buffer to read:

```
#define DATA_LENGTH 10
static uint8_t write_buffer[DATA_LENGTH] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
};

static uint8_t read_buffer[DATA_LENGTH];
```

- Slave address to access:

```
#define SLAVE_ADDRESS 0x12
```

- Number of times to try to send packet if it fails:

```
#define TIMEOUT 1000
```

- Globally accessible module structure:

```
struct i2c_master_module i2c_master_instance;
```

- Function for setting up the module:

```
void configure_i2c_master(void)
{
    /* Initialize config structure and software module. */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer. */
    config_i2c_master.buffer_timeout = 10000;

    /* Initialize and enable device with config. */
    i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}
```

- Add to user application `main()`:

```
/* Configure device and enable. */
configure_i2c_master();

/* Timeout counter. */
uint16_t timeout = 0;

/* Init i2c packet. */
struct i2c_master_packet packet = {
    .address     = SLAVE_ADDRESS,
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
    .ten_bit_address = false,
    .high_speed      = false,
    .hs_master_code  = 0x0,
};
```

#### 9.1.2.2. Workflow

1. Configure and enable module.

```
void configure_i2c_master(void)
{
    /* Initialize config structure and software module. */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer. */
    config_i2c_master.buffer_timeout = 10000;

    /* Initialize and enable device with config. */
    i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}
```

   1. Create and initialize configuration structure.

   ```
   struct i2c_master_config config_i2c_master;
   i2c_master_get_config_defaults(&config_i2c_master);
   ```

   2. Change settings in the configuration.

   ```
   config_i2c_master.buffer_timeout = 10000;
   ```

   3. Initialize the module with the set configurations.

   ```
   i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
   &config_i2c_master);
   ```

4. Enable the module.

```
i2c_master_enable(&i2c_master_instance);
```

2. Create a variable to see when we should stop trying to send packet.

```
uint16_t timeout = 0;
```

3. Create a packet to send.

```
struct i2c_master_packet packet = {
    .address      = SLAVE_ADDRESS,
    .data_length = DATA_LENGTH,
    .data         = write_buffer,
    .ten_bit_address = false,
    .high_speed      = false,
    .hs_master_code  = 0x0,
};
```

### 9.1.3. Implementation

#### 9.1.3.1. Code

Add to user application `main()`:

```
/* Write buffer to slave until success. */
while (i2c_master_write_packet_wait(&i2c_master_instance, &packet) !=
        STATUS_OK) {
    /* Increment timeout counter and check if timed out. */
    if (timeout++ == TIMEOUT) {
        break;
    }
}

/* Read from slave until success. */
packet.data = read_buffer;
while (i2c_master_read_packet_wait(&i2c_master_instance, &packet) !=
        STATUS_OK) {
    /* Increment timeout counter and check if timed out. */
    if (timeout++ == TIMEOUT) {
        break;
    }
}
```

#### 9.1.3.2. Workflow

1. Write packet to slave.

```
while (i2c_master_write_packet_wait(&i2c_master_instance, &packet) !=
        STATUS_OK) {
    /* Increment timeout counter and check if timed out. */
    if (timeout++ == TIMEOUT) {
        break;
    }
}
```

The module will try to send the packet TIMEOUT number of times or until it is successfully sent.

2. Read packet from slave.

```
packet.data = read_buffer;
while (i2c_master_read_packet_wait(&i2c_master_instance, &packet) !=
        STATUS_OK) {
```

```
    /* Increment timeout counter and check if timed out. */
    if (timeout++ == TIMEOUT) {
        break;
    }
}
```

The module will try to read the packet TIMEOUT number of times or until it is successfully read.

## 9.2. Quick Start Guide for SERCOM I$^2$C Master - Callback

In this use case, the I$^2$C will used and set up as follows:

- Master mode
- 100KHz operation speed
- Not operational in standby
- 65535 unknown bus state timeout value

### 9.2.1. Prerequisites

The device must be connected to an I$^2$C slave.

### 9.2.2. Setup

#### 9.2.2.1. Code

The following must be added to the user application:

A sample buffer to write from, a reversed buffer to write from and length of buffers.

```
#define DATA_LENGTH 8

static uint8_t wr_buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07
};

static uint8_t wr_buffer_reversed[DATA_LENGTH] = {
    0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00
};

static uint8_t rd_buffer[DATA_LENGTH];
```

Address of slave:

```
#define SLAVE_ADDRESS 0x12
```

Globally accessible module structure:

```
struct i2c_master_module i2c_master_instance;
```

Globally accessible packet:

```
struct i2c_master_packet wr_packet;
struct i2c_master_packet rd_packet;
```

Function for setting up module:

```
void configure_i2c(void)
{
    /* Initialize config structure and software module */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer */
    config_i2c_master.buffer_timeout = 65535;

    /* Initialize and enable device with config */
    while(i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master)     \
            != STATUS_OK);

    i2c_master_enable(&i2c_master_instance);
}
```

Callback function for write complete:

```
void i2c_write_complete_callback(
        struct i2c_master_module *const module)
{
    /* Initiate new packet read */
    i2c_master_read_packet_job(&i2c_master_instance,&rd_packet);
}
```

Function for setting up the callback functionality of the driver:

```
void configure_i2c_callbacks(void)
{
    /* Register callback function. */
    i2c_master_register_callback(&i2c_master_instance,
i2c_write_complete_callback,
            I2C_MASTER_CALLBACK_WRITE_COMPLETE);
    i2c_master_enable_callback(&i2c_master_instance,
            I2C_MASTER_CALLBACK_WRITE_COMPLETE);
}
```

Add to user application `main()`:

```
/* Configure device and enable. */
configure_i2c();
/* Configure callbacks and enable. */
configure_i2c_callbacks();
```

#### 9.2.2.2. Workflow

1. Configure and enable module.

   ```
   configure_i2c();
   ```

   1. Create and initialize configuration structure.

      ```
      struct i2c_master_config config_i2c_master;
      i2c_master_get_config_defaults(&config_i2c_master);
      ```

   2. Change settings in the configuration.

      ```
      config_i2c_master.buffer_timeout = 65535;
      ```

   3. Initialize the module with the set configurations.

      ```
      while(i2c_master_init(&i2c_master_instance,
      CONF_I2C_MASTER_MODULE, &config_i2c_master)     \
              != STATUS_OK);
      ```

4. Enable the module.

```
i2c_master_enable(&i2c_master_instance);
```

2. Configure callback functionality.

```
configure_i2c_callbacks();
```

1. Register write complete callback.

```
i2c_master_register_callback(&i2c_master_instance,
i2c_write_complete_callback,
        I2C_MASTER_CALLBACK_WRITE_COMPLETE);
```

2. Enable write complete callback.

```
i2c_master_enable_callback(&i2c_master_instance,
        I2C_MASTER_CALLBACK_WRITE_COMPLETE);
```

3. Create a packet to send to slave.

```
wr_packet.address     = SLAVE_ADDRESS;
wr_packet.data_length = DATA_LENGTH;
wr_packet.data        = wr_buffer;
```

### 9.2.3. Implementation

#### 9.2.3.1. Code

Add to user application `main()`:

```
while (true) {
    /* Infinite loop */
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {
        while (!port_pin_get_input_level(BUTTON_0_PIN)) {
            /* Waiting for button steady */
        }
        /* Send every other packet with reversed data */
        if (wr_packet.data[0] == 0x00) {
            wr_packet.data = &wr_buffer_reversed[0];
        } else {
            wr_packet.data = &wr_buffer[0];
        }
        i2c_master_write_packet_job(&i2c_master_instance, &wr_packet);
    }
}
```

#### 9.2.3.2. Workflow

1. Write packet to slave.

```
wr_packet.address     = SLAVE_ADDRESS;
wr_packet.data_length = DATA_LENGTH;
wr_packet.data        = wr_buffer;
```

2. Infinite while loop, while waiting for interaction with slave.

```
while (true) {
    /* Infinite loop */
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {
        while (!port_pin_get_input_level(BUTTON_0_PIN)) {
            /* Waiting for button steady */
        }
        /* Send every other packet with reversed data */
        if (wr_packet.data[0] == 0x00) {
            wr_packet.data = &wr_buffer_reversed[0];
        } else {
            wr_packet.data = &wr_buffer[0];
        }
        i2c_master_write_packet_job(&i2c_master_instance, &wr_packet);
    }
}
```

### 9.2.4. Callback

Each time a packet is sent, the callback function will be called.

#### 9.2.4.1. Workflow

- Write complete callback:
  1. Send every other packet in reversed order.
     ```
     if (wr_packet.data[0] == 0x00) {
         wr_packet.data = &wr_buffer_reversed[0];
     } else {
         wr_packet.data = &wr_buffer[0];
     }
     ```
  2. Write new packet to slave.
     ```
     wr_packet.address     = SLAVE_ADDRESS;
     wr_packet.data_length = DATA_LENGTH;
     wr_packet.data        = wr_buffer;
     ```

## 9.3. Quick Start Guide for Using DMA with SERCOM I$^2$C Master

The supported board list:

- SAMD21 Xplained Pro
- SAMR21 Xplained Pro
- SAML21 Xplained Pro
- SAML22 Xplained Pro
- SAMDA1 Xplained Pro
- SAMC21 Xplained Pro

In this use case, the I$^2$C will used and set up as follows:

- Master mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value
- 65535 unknown bus state timeout value

### 9.3.1. Prerequisites

The device must be connected to an I$^2$C slave.

### 9.3.2. Setup

#### 9.3.2.1. Code

The following must be added to the user application:

- A sample buffer to send, number of entries to send and address of slave:

```
#define DATA_LENGTH 10
static uint8_t buffer[DATA_LENGTH] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
};

#define SLAVE_ADDRESS 0x12
```

  Number of times to try to send packet if it fails:

```
#define TIMEOUT 1000
```

- Globally accessible module structure:

```
struct i2c_master_module i2c_master_instance;
```

- Function for setting up the module:

```
static void configure_i2c_master(void)
{
    /* Initialize config structure and software module. */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer. */
    config_i2c_master.buffer_timeout = 10000;

    /* Initialize and enable device with config. */
    i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}
```

- Globally accessible DMA module structure:

```
struct dma_resource example_resource;
```

- Globally transfer done flag:

```
static volatile bool transfer_is_done = false;
```

- Globally accessible DMA transfer descriptor:

```
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor;
```

- Function for transfer done callback:

```
static void transfer_done(struct dma_resource* const resource )
{
    UNUSED(resource);

    transfer_is_done = true;
}
```

- Function for setting up the DMA resource:

```
static void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.peripheral_trigger = CONF_I2C_DMA_TRIGGER;
    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;

    dma_allocate(resource, &config);
}
```

- Function for setting up the DMA transfer descriptor:

```
static void setup_dma_descriptor(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.block_transfer_count = DATA_LENGTH;
    descriptor_config.source_address = (uint32_t)buffer + DATA_LENGTH;
    descriptor_config.destination_address =
            (uint32_t)(&i2c_master_instance.hw->I2CM.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}
```

- Add to user application `main()`:

```
configure_i2c_master();

configure_dma_resource(&example_resource);
setup_dma_descriptor(&example_descriptor);
dma_add_descriptor(&example_resource, &example_descriptor);
dma_register_callback(&example_resource, transfer_done,
        DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource, DMA_CALLBACK_TRANSFER_DONE);
```

#### 9.3.2.2. Workflow

1. Configure and enable module:

   ```
   configure_i2c_master();
   ```

   1. Create and initialize configuration structure.

      ```
      struct i2c_master_config config_i2c_master;
      i2c_master_get_config_defaults(&config_i2c_master);
      ```

   2. Change settings in the configuration.

      ```
      config_i2c_master.buffer_timeout = 10000;
      ```

   3. Initialize the module with the set configurations.

      ```
      i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
      &config_i2c_master);
      ```

   4. Enable the module.

      ```
      i2c_master_enable(&i2c_master_instance);
      ```

2. Configure DMA

   1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

      ```
      struct dma_resource_config config;
      ```

2. Initialize the DMA resource configuration struct with the module's default values.

   ```
   dma_get_config_defaults(&config);
   ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
3. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM TX trigger causes a transaction transfer in this example.

   ```
   config.peripheral_trigger = CONF_I2C_DMA_TRIGGER;
   config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
   ```
4. Allocate a DMA resource with the configurations.

   ```
   dma_allocate(resource, &config);
   ```
5. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

   ```
   struct dma_descriptor_config descriptor_config;
   ```
6. Initialize the DMA transfer descriptor configuration struct with the module's default values.

   ```
   dma_descriptor_get_config_defaults(&descriptor_config);
   ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
7. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

   ```
   descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
   descriptor_config.dst_increment_enable = false;
   descriptor_config.block_transfer_count = DATA_LENGTH;
   descriptor_config.source_address = (uint32_t)buffer + DATA_LENGTH;
   descriptor_config.destination_address =
           (uint32_t)(&i2c_master_instance.hw->I2CM.DATA.reg);
   ```
8. Create the DMA transfer descriptor.

   ```
   dma_descriptor_create(descriptor, &descriptor_config);
   ```

### 9.3.3. Implementation

#### 9.3.3.1. Code

Add to user application `main()`:

```
dma_start_transfer_job(&example_resource);

i2c_master_dma_set_transfer(&i2c_master_instance, SLAVE_ADDRESS,
        DATA_LENGTH, I2C_TRANSFER_WRITE);

while (!transfer_is_done) {
    /* Wait for transfer done */
}

while (true) {
}
```

#### 9.3.3.2. Workflow

1. Start the DMA transfer job.

```
dma_start_transfer_job(&example_resource);
```

2. Set the auto address length and enable flag.

```
i2c_master_dma_set_transfer(&i2c_master_instance, SLAVE_ADDRESS,
        DATA_LENGTH, I2C_TRANSFER_WRITE);
```

3. Waiting for transfer complete.

```
while (!transfer_is_done) {
    /* Wait for transfer done */
}
```

4. Enter an infinite loop once transfer complete.

```
while (true) {
}
```

# 10. Document Revision History

| Doc. Rev. | Date | Comments |
|---|---|---|
| 42117E | 12/2015 | Added support for SAM L21/L22, SAM DA1, SAM D09, and SAM C21 |
| 42117D | 12/2014 | Added support for 10-bit addressing and high speed in SAM D21. Added support for SAM R21 and SAM D10/D11. |
| 42117C | 01/2014 | Added support for SAM D21 |
| 42117B | 06/2013 | Corrected documentation typos. Updated I$^2$C Bus State Diagram. |
| 42117A | 06/2013 | Initial release |

Atmel | Enabling Unlimited Possibilities®

**Atmel Corporation** 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**