SMART ARM-based Microcontrollers

AT03254: SAM D/R/L/C I2C Slave 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^2C module, for the transfer of data via an I^2C bus. The following driver API modes are covered by this manual:

- Slave Mode Polled APIs
- Slave 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:

Table of Contents

1. Software License

2. Prerequisites

There are no prerequisites.

3. Module Overview

The outline of this section is as follows:

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).

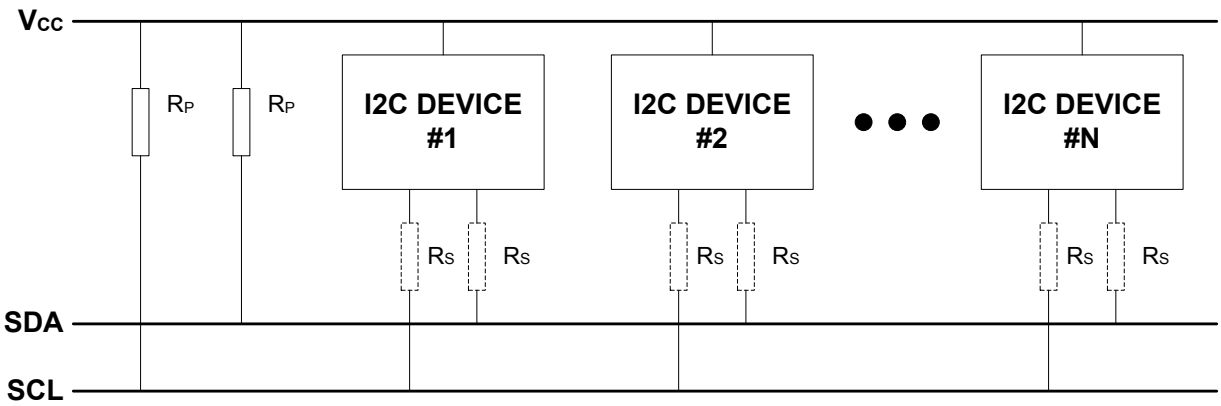
The I^2C bus provides a simple, but efficient method of interconnecting multiple master and slave devices. An arbitration mechanism is provided for resolving bus ownership between masters, as only one master device may own the bus at any given time. The arbitration mechanism relies on the wired-AND connections 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 I²C 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 I²C devices are driving the bus. These are optional, and can be replaced with a constant current source.

Figure 3-1 I²C Bus Topology



Note: RS is optional

3.4. Transactions

The I²C 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.

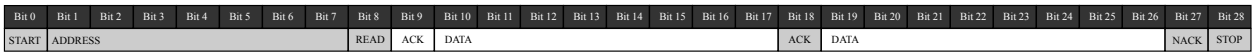
3.4.2. Data Packets

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 I²C Packet Read



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 I²C Packet Write

3.4.4. Packet Timeout

When a master sends an I²C 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

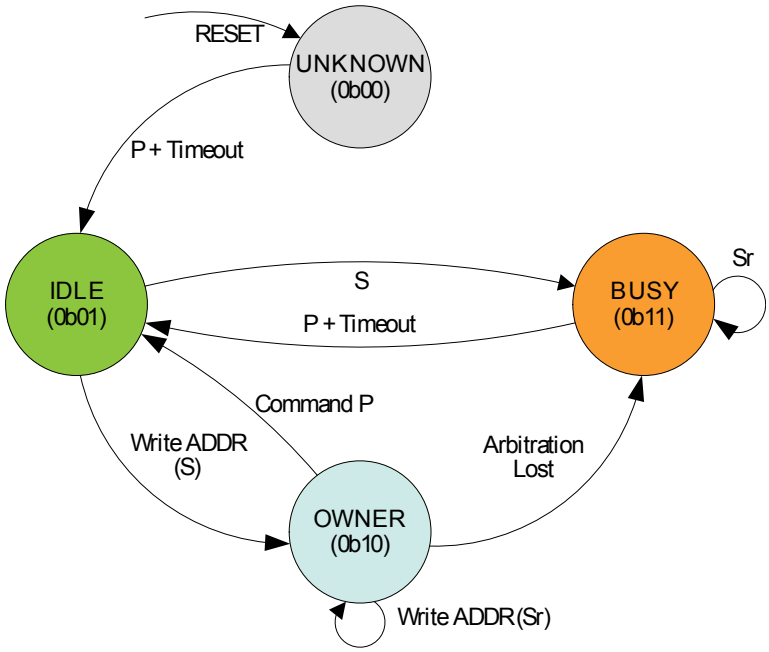
As the I^2C bus is limited to one transaction at the time, a master that wants to perform a bus transaction must wait until the bus is free. Because of this, it is necessary for all masters in a multi-master system to know the current status of the bus to be able to avoid conflicts and to ensure data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

This is the configuration structure for the I²C slave device. It is used as an argument for i2c_slave_init to provide the desired configurations for the module. The structure should be initialized using the i2c_slave_get_config_defaults.

Table 7-1 Members

Type	Name	Description
uint16_t	address	Address or upper limit of address range
uint16_t	address_mask	Address mask, second address, or lower limit of address range
enum i2c_slave_address_mode	address_mode	Addressing mode
uint16_t	buffer_timeout	Timeout to wait for master in polled functions
bool	enable_general_call_address	Enable general call address recognition (general call address is defined as 0000000 with direction bit 0).
bool	enable_nack_on_address	Enable NACK on address match (this can be changed after initialization via the i2c_slave_enable_nack_on_address and i2c_slave_disable_nack_on_address functions).
bool	enable_scl_low_timeout	Set to enable the SCL low timeout
enum gclk_generator	generator_source	GCLK generator to use as clock source
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)
enum i2c_slave_sda_hold_time	sda_hold_time	SDA hold time with respect to the negative edge of SCL

Type	Name	Description
bool	slave_scl_low_extend_timeout	Set to enable slave SCL low extend time-out
bool	ten_bit_address	Enable 10-bit addressing
enum i2c_slave_transfer_speed	transfer_speed	Transfer speed mode

7.1.2. Struct i2c_slave_module

SERCOM I²C slave 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_slave_packet

Structure to be used when transferring I²C slave packets.

Table 7-2 Members

Type	Name	Description
uint8_t *	data	Data array containing all data to be transferred
uint16_t	data_length	Length of data array

7.2. Macro Definitions

7.2.1. Driver Feature Definition

Define SERCOM I²C 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
```

7.2.2. I²C Slave Status Flags

I²C slave status flags, returned by i2c_slave_get_status() and cleared by i2c_slave_clear_status().

7.2.2.1. Macro I2C_SLAVE_STATUS_ADDRESS_MATCH

```
#define I2C_SLAVE_STATUS_ADDRESS_MATCH
```

Address Match.
Note: Should only be cleared internally by driver.

7.2.2.2. Macro I2C_SLAVE_STATUS_DATA_READY

```
#define I2C_SLAVE_STATUS_DATA_READY
```

Data Ready.

7.2.2.3. Macro I2C_SLAVE_STATUS_STOP_RECEIVED

```
#define I2C_SLAVE_STATUS_STOP_RECEIVED
```

Stop Received.

7.2.2.4. Macro I2C_SLAVE_STATUS_CLOCK_HOLD

```
#define I2C_SLAVE_STATUS_CLOCK_HOLD
```

Clock Hold.
Note: Cannot be cleared, only valid when I2C_SLAVE_STATUS_ADDRESS_MATCH is set.

7.2.2.5. Macro I2C_SLAVE_STATUS_SCL_LOW_TIMEOUT

```
#define I2C_SLAVE_STATUS_SCL_LOW_TIMEOUT
```

SCL Low Timeout.

7.2.2.6. Macro I2C_SLAVE_STATUS_REPEATED_START

```
#define I2C_SLAVE_STATUS_REPEATED_START
```

Repeated Start.
Note: Cannot be cleared, only valid when I2C_SLAVE_STATUS_ADDRESS_MATCH is set.

7.2.2.7. Macro I2C_SLAVE_STATUS_RECEIVED_NACK

```
#define I2C_SLAVE_STATUS_RECEIVED_NACK
```

Received not acknowledge.
Note: Cannot be cleared.

7.2.2.8. Macro I2C_SLAVE_STATUS_COLLISION

```
#define I2C_SLAVE_STATUS_COLLISION
```

Transmit Collision.

7.2.2.9. Macro I2C_SLAVE_STATUS_BUS_ERROR

```
#define I2C_SLAVE_STATUS_BUS_ERROR
```

Bus error.

7.3. Function Definitions

7.3.1. Lock/Unlock

7.3.1.1. Function i2c_slave_lock()

Attempt to get lock on driver instance.

```
enum status_code i2c_slave_lock(
        struct i2c_slave_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_slave_unlock()

Unlock driver instance.

```
void i2c_slave_unlock(
        struct i2c_slave_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_slave_is_syncing()

Returns the synchronization status of the module.

```
bool i2c_slave_is_syncing(
        const struct i2c_slave_module *const module)
```

Returns the synchronization status of the module.

Table 7-7 Parameters

Data direction	Parameter name	Description
[out]	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_slave_get_config_defaults()

Gets the I2C slave default configurations.

```
void i2c_slave_get_config_defaults(
        struct i2c_slave_config *const config)
```

This will initialize the configuration structure to known default values.

The default configuration is as follows:

- Disable SCL low timeout
- 300ns - 600ns SDA hold time
- Buffer timeout = 65535
- Address with mask
- Address = 0
- Address mask = 0 (one single address)
- General call address disabled
- Address nack disabled if the interrupt driver is used
- GCLK generator 0

- Do not run in standby
- PINMUX_DEFAULT for SERCOM pads

Those default configuration only available if the device supports it:

- Not using 10-bit addressing
- Standard-mode and Fast-mode transfer speed
- SCL stretch disabled
- Slave SCL low extend time-out disabled

Table 7-9 Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to configuration structure to be initialized

7.3.2.3. Function i2c_slave_init()

Initializes the requested I2C hardware module.

```
enum status_code i2c_slave_init(
        struct i2c_slave_module *const module,
        Sercom *const hw,
        const struct i2c_slave_config *const config)
```

Initializes the SERCOM I^2C slave 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

7.3.2.4. Function i2c_slave_enable()

Enables the I2C module.

```
void i2c_slave_enable(
        const struct i2c_slave_module *const module)
```

This will enable the requested I²C module.

Table 7-12 Parameters

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

7.3.2.5. Function i2c_slave_disable()

Disables the I2C module.

```
void i2c_slave_disable(
        const struct i2c_slave_module *const module)
```

This will disable the I²C module specified in the provided software module structure.

Table 7-13 Parameters

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

7.3.2.6. Function i2c_slave_reset()

Resets the hardware module.

```
void i2c_slave_reset(
        struct i2c_slave_module *const module)
```

This will 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_slave_write_packet_wait()

Writes a packet to the master.

```
enum status_code i2c_slave_write_packet_wait(
        struct i2c_slave_module *const module,
        struct i2c_slave_packet *const packet)
```

Writes a packet to the master. This will wait for the master to issue a request.

Table 7-15 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure
[in]	packet	Packet to write to master

Returns
Status of packet write.

Table 7-16 Return Values

Return value	Description
STATUS_OK	Packet was written successfully
STATUS_ERR_DENIED	Start condition not received, another interrupt flag is set
STATUS_ERR_IO	There was an error in the previous transfer
STATUS_ERR_BAD_FORMAT	Master wants to write data
STATUS_ERR_INVALID_ARG	Invalid argument(s) was provided
STATUS_ERR_BUSY	The I^2C module is busy with a job
STATUS_ERR_ERR_OVERFLOW	Master NACKed before entire packet was transferred
STATUS_ERR_TIMEOUT	No response was given within the timeout period

7.3.3.2. Function i2c_slave_read_packet_wait()

Reads a packet from the master.

```
enum status_code i2c_slave_read_packet_wait(
        struct i2c_slave_module *const module,
        struct i2c_slave_packet *const packet)
```

Reads a packet from the master. This will wait for the master to issue a request.

Table 7-17 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure
[out]	packet	Packet to read from master

Returns
Status of packet read.

Table 7-18 Return Values

Return value	Description
STATUS_OK	Packet was read successfully
STATUS_ABORTED	Master sent stop condition or repeated start before specified length of bytes was received
STATUS_ERR_IO	There was an error in the previous transfer
STATUS_ERR_DENIED	Start condition not received, another interrupt flag is set
STATUS_ERR_INVALID_ARG	Invalid argument(s) was provided
STATUS_ERR_BUSY	The I^2C module is busy with a job
STATUS_ERR_BAD_FORMAT	Master wants to read data
STATUS_ERR_ERR_OVERFLOW	Last byte received overflows buffer

7.3.3.3. Function i2c_slave_get_direction_wait()

Waits for a start condition on the bus.

```
enum i2c_slave_direction i2c_slave_get_direction_wait(
        struct i2c_slave_module *const module)
```

Note: This function is only available for 7-bit slave addressing.

Waits for the master to issue a start condition on the bus.

Note: This function does not check for errors in the last transfer, this will be discovered when reading or writing.

Table 7-19 Parameters

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

Returns
Direction of the current transfer, when in slave mode.

Table 7-20 Return Values

Return value	Description
I2C_SLAVE_DIRECTION_NONE	No request from master within timeout period
I2C_SLAVE_DIRECTION_READ	Write request from master
I2C_SLAVE_DIRECTION_WRITE	Read request from master

7.3.4. Status Management

7.3.4.1. Function i2c_slave_get_status()

Retrieves the current module status.

```
uint32_t i2c_slave_get_status(
        struct i2c_slave_module *const module)
```

Checks the status of the module and returns it as a bitmask of status flags.

Table 7-21 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the I²C slave software device struct

Returns
Bitmask of status flags.

Table 7-22 Return Values

Return value	Description
I2C_SLAVE_STATUS_ADDRESS_MATCH	A valid address has been received
I2C_SLAVE_STATUS_DATA_READY	A I²C slave byte transmission is successfully completed
I2C_SLAVE_STATUS_STOP_RECEIVED	A stop condition is detected for a transaction being processed
I2C_SLAVE_STATUS_CLOCK_HOLD	The slave is holding the SCL line low
I2C_SLAVE_STATUS_SCL_LOW_TIMEOUT	An SCL low time-out has occurred
I2C_SLAVE_STATUS_REPEATED_START	Indicates a repeated start, only valid if I2C_SLAVE_STATUS_ADDRESS_MATCH is set
I2C_SLAVE_STATUS_RECEIVED_NACK	The last data packet sent was not acknowledged
I2C_SLAVE_STATUS_COLLISION	The I²C slave was not able to transmit a high data or NACK bit
I2C_SLAVE_STATUS_BUS_ERROR	An illegal bus condition has occurred on the bus

7.3.4.2. Function i2c_slave_clear_status()

Clears a module status flag.

```
void i2c_slave_clear_status(
        struct i2c_slave_module *const module,
        uint32_t status_flags)
```

Clears the given status flag of the module.

Note: Not all status flags can be cleared.

Table 7-23 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the I²C software device struct
[in]	status_flags	Bit mask of status flags to clear

7.3.5. SERCOM I²C slave with DMA Interfaces

7.3.5.1. Function i2c_slave_dma_read_interrupt_status()

Read SERCOM I2C interrupt status.

```
uint8_t i2c_slave_dma_read_interrupt_status(
        struct i2c_slave_module *const module)
```

Read I²C interrupt status for DMA transfer.

Table 7-24 Parameters

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

7.3.5.2. Function i2c_slave_dma_write_interrupt_status()

Write SERCOM I2C interrupt status.

```
void i2c_slave_dma_write_interrupt_status(
        struct i2c_slave_module *const module,
        uint8_t flag)
```

Write I²C interrupt status for DMA transfer.

Table 7-25 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock
[in]	flag	Interrupt flag status

7.3.6. Address Match Functionality

7.3.6.1. Function i2c_slave_enable_nack_on_address()

Enables sending of NACK on address match.

```
void i2c_slave_enable_nack_on_address(
        struct i2c_slave_module *const module)
```

Enables sending of NACK on address match, thus discarding any incoming transaction.

Table 7-26 Parameters

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

7.3.6.2. Function i2c_slave_disable_nack_on_address()

Disables sending NACK on address match.

```
void i2c_slave_disable_nack_on_address(
        struct i2c_slave_module *const module)
```

Disables sending of NACK on address match, thus acknowledging incoming transactions.

Table 7-27 Parameters

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

7.3.7. Callbacks

7.3.7.1. Function i2c_slave_register_callback()

Registers callback for the specified callback type.

```
void i2c_slave_register_callback(
        struct i2c_slave_module *const module,
        i2c_slave_callback_t callback,
        enum i2c_slave_callback callback_type)
```

Associates the given callback function with the specified callback type. To enable the callback, the i2c_slave_enable_callback function must be used.

Table 7-28 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

7.3.7.2. Function i2c_slave_unregister_callback()

Unregisters callback for the specified callback type.

```
void i2c_slave_unregister_callback(
        struct i2c_slave_module *const module,
        enum i2c_slave_callback callback_type)
```

Removes the currently registered callback for the given callback type.

Table 7-29 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Callback type to unregister

7.3.7.3. Function i2c_slave_enable_callback()

Enables callback.

```
void i2c_slave_enable_callback(
        struct i2c_slave_module *const module,
        enum i2c_slave_callback callback_type)
```

Enables the callback specified by the callback_type.

Table 7-30 Parameters

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

7.3.7.4. Function i2c_slave_disable_callback()

Disables callback.

```
void i2c_slave_disable_callback(
        struct i2c_slave_module *const module,
        enum i2c_slave_callback callback_type)
```

Disables the callback specified by the callback_type.

Table 7-31 Parameters

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

7.3.8. Read and Write, Interrupt-Driven

7.3.8.1. Function i2c_slave_read_packet_job()

Initiates a reads packet operation.

```
enum status_code i2c_slave_read_packet_job(
        struct i2c_slave_module *const module,
        struct i2c_slave_packet *const packet)
```

Reads a data packet from the master. A write request must be initiated by the master before the packet can be read.

The I2C_SLAVE_CALLBACK_WRITE_REQUEST callback can be used to call this function.

Table 7-32 Parameters

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

Returns
Status of starting asynchronously reading I²C packet.

Table 7-33 Return Values

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

7.3.8.2. Function i2c_slave_write_packet_job()

Initiates a write packet operation.

```
enum status_code i2c_slave_write_packet_job(
        struct i2c_slave_module *const module,
        struct i2c_slave_packet *const packet)
```

Writes a data packet to the master. A read request must be initiated by the master before the packet can be written.

The I2C_SLAVE_CALLBACK_READ_REQUEST callback can be used to call this function.

Table 7-34 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 I²C packet.

Table 7-35 Return Values

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

7.3.8.3. Function i2c_slave_cancel_job()

Cancels any currently ongoing operation.

```
void i2c_slave_cancel_job(
        struct i2c_slave_module *const module)
```

Terminates the running transfer operation.

Table 7-36 Parameters

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

7.3.8.4. Function i2c_slave_get_job_status()

Gets status of ongoing job.

```
enum status_code i2c_slave_get_job_status(
        struct i2c_slave_module *const module)
```

Will return the status of the ongoing job, or the error that occurred in the last transfer operation. The status will be cleared when starting a new job.

Table 7-37 Parameters

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

Returns
Status of job.

Table 7-38 Return Values

Return value	Description
STATUS_OK	No error has occurred
STATUS_BUSY	Transfer is in progress
STATUS_ERR_IO	A collision, timeout or bus error happened in the last transfer
STATUS_ERR_TIMEOUT	A timeout occurred
STATUS_ERR_OVERFLOW	Data from master overflows receive buffer

7.4. Enumeration Definitions

7.4.1. Enum i2c_slave_address_mode

Enum for the possible address modes.

Table 7-39 Members

Enum value	Description
I2C_SLAVE_ADDRESS_MODE_MASK	Address match on address_mask used as a mask to address
I2C_SLAVE_ADDRESS_MODE_TWO_ADDRESSES	Address math on both address and address_mask
I2C_SLAVE_ADDRESS_MODE_RANGE	Address match on range of addresses between and including address and address_mask

7.4.2. Enum i2c_slave_callback

The available callback types for the I^{2}C slave.

Table 7-40 Members

Enum value	Description
I2C_SLAVE_CALLBACK_WRITE_COMPLETE	Callback for packet write complete
I2C_SLAVE_CALLBACK_READ_COMPLETE	Callback for packet read complete
I2C_SLAVE_CALLBACK_READ_REQUEST	Callback for read request from master - can be used to issue a write
I2C_SLAVE_CALLBACK_WRITE_REQUEST	Callback for write request from master - can be used to issue a read
I2C_SLAVE_CALLBACK_ERROR	Callback for error
I2C_SLAVE_CALLBACK_ERROR_LAST_TRANSFER	Callback for error in last transfer. Discovered on a new address interrupt.

7.4.3. Enum i2c_slave_direction

Enum for the direction of a request.

Table 7-41 Members

Enum value	Description
I2C_SLAVE_DIRECTION_READ	Read
I2C_SLAVE_DIRECTION_WRITE	Write
I2C_SLAVE_DIRECTION_NONE	No direction

7.4.4. Enum i2c_slave_sda_hold_time

Enum for the possible SDA hold times with respect to the negative edge of SCL.

Table 7-42 Members

Enum value	Description
I2C_SLAVE_SDA_HOLD_TIME_DISABLED	SDA hold time disabled
I2C_SLAVE_SDA_HOLD_TIME_50NS_100NS	SDA hold time 50ns - 100ns
I2C_SLAVE_SDA_HOLD_TIME_300NS_600NS	SDA hold time 300ns - 600ns
I2C_SLAVE_SDA_HOLD_TIME_400NS_800NS	SDA hold time 400ns - 800ns

7.4.5. Enum i2c_slave_transfer_speed

Enum for the transfer speed.

Table 7-43 Members

Enum value	Description
I2C_SLAVE_SPEED_STANDARD_AND_FAST	Standard-mode (Sm) up to 100KHz and Fast-mode (Fm) up to 400KHz
I2C_SLAVE_SPEED_FAST_MODE_PLUS	Fast-mode Plus (Fm+) up to 1MHz
I2C_SLAVE_SPEED_HIGH_SPEED	High-speed mode (Hs-mode) up to 3.4MHz

7.4.6. Enum i2c_transfer_direction

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

Table 7-44 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 I²C Driver

8.1. Acronyms

Table 8-1 Acronyms on page 32 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 I²C 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 32 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 32.

Table 8-2 Module History

Changelog
• Added 10-bit addressing and high speed support in SAM D21 • 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 • Added fast mode plus support in SAM D21
Fixed incorrect logical mask for determining if a bus error has occurred in I²C Slave mode
Initial Release

9. Examples for SERCOM I²C Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM I2C Slave 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 Slave module - Basic Use Case
  - Quick Start Guide for the I2C Slave module - Callback Use Case
- Quick Start Guide for the I2C Slave module - DMA Use Case

9.1. Quick Start Guide for SERCOM I²C Slave - Basic

In this use case, the I²C will used and set up as follows:

- Slave mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value

9.1.1. Prerequisites

The device must be connected to an I²C master.

9.1.2. Setup

9.1.2.1. Code

The following must be added to the user application:

A sample buffer to write from, a sample buffer to read to and length of buffers:

```
#define DATA_LENGTH 10

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

Address to respond to:

```
#define SLAVE_ADDRESS 0x12
```

Globally accessible module structure:

```
struct i2c_slave_module i2c_slave_instance;
```

Function for setting up the module:

```
void configure_i2c_slave(void)
{
    /* Create and initialize config_i2c_slave structure */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);

    /* Change address and address_mode */
    config_i2c_slave.address        = SLAVE_ADDRESS;
```

```
    config_i2c_slave.address_mode   = I2C_SLAVE_ADDRESS_MODE_MASK;
    config_i2c_slave.buffer_timeout = 1000;

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

Add to user application `main()`:

```
configure_i2c_slave();

enum i2c_slave_direction dir;
struct i2c_slave_packet packet = {
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
};
```

9.1.2.2. Workflow

1. Configure and enable module.

```
configure_i2c_slave();
```

   1. Create and initialize configuration structure.

```
struct i2c_slave_config config_i2c_slave;
i2c_slave_get_config_defaults(&config_i2c_slave);
```

   2. Change address and address mode settings in the configuration.

```
config_i2c_slave.address        = SLAVE_ADDRESS;
config_i2c_slave.address_mode   = I2C_SLAVE_ADDRESS_MODE_MASK;
config_i2c_slave.buffer_timeout = 1000;
```

   3. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);
```

   4. Enable the module.

```
i2c_slave_enable(&i2c_slave_instance);
```

2. Create variable to hold transfer direction.

```
enum i2c_slave_direction dir;
```

3. Create packet variable to transfer.

```
struct i2c_slave_packet packet = {
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
};
```

9.1.3. Implementation

9.1.3.1. Code

Add to user application `main()`:

```
while (true) {
    /* Wait for direction from master */
```

```
    dir = i2c_slave_get_direction_wait(&i2c_slave_instance);

    /* Transfer packet in direction requested by master */
    if (dir == I2C_SLAVE_DIRECTION_READ) {
        packet.data = read_buffer;
        i2c_slave_read_packet_wait(&i2c_slave_instance, &packet);
    } else if (dir == I2C_SLAVE_DIRECTION_WRITE) {
        packet.data = write_buffer;
        i2c_slave_write_packet_wait(&i2c_slave_instance, &packet);
    }
}
```

9.1.3.2. Workflow

1. Wait for start condition from master and get transfer direction.

```
dir = i2c_slave_get_direction_wait(&i2c_slave_instance);
```

2. Depending on transfer direction, set up buffer to read to or write from, and write or read from master.

```
if (dir == I2C_SLAVE_DIRECTION_READ) {
    packet.data = read_buffer;
    i2c_slave_read_packet_wait(&i2c_slave_instance, &packet);
} else if (dir == I2C_SLAVE_DIRECTION_WRITE) {
    packet.data = write_buffer;
    i2c_slave_write_packet_wait(&i2c_slave_instance, &packet);
}
```

9.2. Quick Start Guide for SERCOM I²C Slave - Callback

In this use case, the I²C will used and set up as follows:

- Slave mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value

9.2.1. Prerequisites

The device must be connected to an I²C master.

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 sample buffer to read to and length of buffers:

```
#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];
```

Address to respond to:

```
#define SLAVE_ADDRESS 0x12
```

Globally accessible module structure:

```
struct i2c_slave_module i2c_slave_instance;
```

Globally accessible packet:

```
static struct i2c_slave_packet packet;
```

Function for setting up the module:

```
void configure_i2c_slave(void)
{
    /* Initialize config structure and module instance */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);
    /* Change address and address_mode */
    config_i2c_slave.address      = SLAVE_ADDRESS;
    config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
    /* Initialize and enable device with config */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

Callback function for read request from a master:

```
void i2c_read_request_callback(
        struct i2c_slave_module *const module)
{
    /* Init i2c packet */
    packet.data_length = DATA_LENGTH;
    packet.data        = write_buffer;

    /* Write buffer to master */
    i2c_slave_write_packet_job(module, &packet);
}
```

Callback function for write request from a master:

```
void i2c_write_request_callback(
        struct i2c_slave_module *const module)
{
    /* Init i2c packet */
    packet.data_length = DATA_LENGTH;
    packet.data        = read_buffer;

    /* Read buffer from master */
    if (i2c_slave_read_packet_job(module, &packet) != STATUS_OK) {
    }
}
```

Function for setting up the callback functionality of the driver:

```
void configure_i2c_slave_callbacks(void)
{
    /* Register and enable callback functions */
    i2c_slave_register_callback(&i2c_slave_instance,
i2c_read_request_callback,
            I2C_SLAVE_CALLBACK_READ_REQUEST);
    i2c_slave_enable_callback(&i2c_slave_instance,
            I2C_SLAVE_CALLBACK_READ_REQUEST);
```

```
    i2c_slave_register_callback(&i2c_slave_instance,
i2c_write_request_callback,
            I2C_SLAVE_CALLBACK_WRITE_REQUEST);
    i2c_slave_enable_callback(&i2c_slave_instance,
            I2C_SLAVE_CALLBACK_WRITE_REQUEST);
}
```

Add to user application `main()`:

```
/* Configure device and enable */
configure_i2c_slave();
configure_i2c_slave_callbacks();
```

9.2.2.2. Workflow

1. Configure and enable module.

```
configure_i2c_slave();
```

   1. Create and initialize configuration structure.

```
struct i2c_slave_config config_i2c_slave;
i2c_slave_get_config_defaults(&config_i2c_slave);
```

   2. Change address and address mode settings in the configuration.

```
config_i2c_slave.address      = SLAVE_ADDRESS;
config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
```

   3. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);
```

   4. Enable the module.

```
i2c_slave_enable(&i2c_slave_instance);
```

2. Register and enable callback functions.

```
configure_i2c_slave_callbacks();
```

   1. Register and enable callbacks for read and write requests from master.

```
i2c_slave_register_callback(&i2c_slave_instance,
i2c_read_request_callback,
        I2C_SLAVE_CALLBACK_READ_REQUEST);
i2c_slave_enable_callback(&i2c_slave_instance,
        I2C_SLAVE_CALLBACK_READ_REQUEST);

i2c_slave_register_callback(&i2c_slave_instance,
i2c_write_request_callback,
        I2C_SLAVE_CALLBACK_WRITE_REQUEST);
i2c_slave_enable_callback(&i2c_slave_instance,
        I2C_SLAVE_CALLBACK_WRITE_REQUEST);
```

9.2.3. Implementation

9.2.3.1. Code

Add to user application `main()`:

```
while (true) {
    /* Infinite loop while waiting for I2C master interaction */
}
```

9.2.3.2. Workflow

1. Infinite while loop, while waiting for interaction from master.

```
while (true) {
    /* Infinite loop while waiting for I2C master interaction */
}
```

9.2.4. Callback

When an address packet is received, one of the callback functions will be called, depending on the DIR bit in the received packet.

9.2.4.1. Workflow

- Read request callback:
  1. Length of buffer and buffer to be sent to master is initialized.

```
packet.data_length = DATA_LENGTH;
packet.data        = write_buffer;
```

  2. Write packet to master.

```
i2c_slave_write_packet_job(module, &packet);
```

- Write request callback:
  1. Length of buffer and buffer to be read from master is initialized.

```
packet.data_length = DATA_LENGTH;
packet.data        = read_buffer;
```

  2. Read packet from master.

```
if (i2c_slave_read_packet_job(module, &packet) != STATUS_OK) {
}
```

9.3. Quick Start Guide for Using DMA with SERCOM I²C Slave

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 I²C will used and set up as follows:

- Slave mode

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

9.3.1. Prerequisites

The device must be connected to an I²C slave.

9.3.2. Setup

9.3.2.1. Code

The following must be added to the user application:

- Address to respond to:

```
#define SLAVE_ADDRESS 0x12
```

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

```
#define DATA_LENGTH 10
uint8_t read_buffer[DATA_LENGTH];
```

- Globally accessible module structure:

```
struct i2c_slave_module i2c_slave_instance;
```

- Function for setting up the module:

```
void configure_i2c_slave(void)
{
    /* Create and initialize config_i2c_slave structure */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);

    /* Change address and address_mode */
    config_i2c_slave.address        = SLAVE_ADDRESS;
    config_i2c_slave.address_mode   = I2C_SLAVE_ADDRESS_MODE_MASK;
    config_i2c_slave.buffer_timeout = 1000;

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

- Globally accessible DMA module structure:

```
struct dma_resource i2c_dma_resource;
```

- Globally accessible DMA transfer descriptor:

```
COMPILER_ALIGNED(16)
DmacDescriptor i2c_dma_descriptor;
```

- Function for setting up the DMA resource:

```
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:

```
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.src_increment_enable = false;
    descriptor_config.block_transfer_count = DATA_LENGTH;
    descriptor_config.destination_address = (uint32_t)read_buffer +
DATA_LENGTH;
    descriptor_config.source_address =
            (uint32_t)(&i2c_slave_instance.hw->I2CS.DATA.reg);

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

- Add to user application `main()`:

```
configure_i2c_slave();

configure_dma_resource(&i2c_dma_resource);
setup_dma_descriptor(&i2c_dma_descriptor);
dma_add_descriptor(&i2c_dma_resource, &i2c_dma_descriptor);
```

9.3.2.2. Workflow

1. Configure and enable module:

```
void configure_i2c_slave(void)
{
    /* Create and initialize config_i2c_slave structure */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);

    /* Change address and address_mode */
    config_i2c_slave.address        = SLAVE_ADDRESS;
    config_i2c_slave.address_mode   = I2C_SLAVE_ADDRESS_MODE_MASK;
    config_i2c_slave.buffer_timeout = 1000;

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

   1. Create and initialize configuration structure.

```
struct i2c_slave_config config_i2c_slave;
i2c_slave_get_config_defaults(&config_i2c_slave);
```

2. Change settings in the configuration.

```
config_i2c_slave.address        = SLAVE_ADDRESS;
config_i2c_slave.address_mode   = I2C_SLAVE_ADDRESS_MODE_MASK;
config_i2c_slave.buffer_timeout = 1000;
```

3. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);
```

4. Enable the module.

```
i2c_slave_enable(&i2c_slave_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 RX trigger causes a beat 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.src_increment_enable = false;
descriptor_config.block_transfer_count = DATA_LENGTH;
descriptor_config.destination_address = (uint32_t)read_buffer +
DATA_LENGTH;
descriptor_config.source_address =
        (uint32_t)(&i2c_slave_instance.hw->I2CS.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(&i2c_dma_resource);

while (true) {
    if (i2c_slave_dma_read_interrupt_status(&i2c_slave_instance) &
                SERCOM_I2CS_INTFLAG_AMATCH) {
        i2c_slave_dma_write_interrupt_status(&i2c_slave_instance,
                SERCOM_I2CS_INTFLAG_AMATCH);
    }
}
```

9.3.3.2. Workflow

1. Start to wait a packet from master.

```
dma_start_transfer_job(&i2c_dma_resource);
```

2. Once data ready, clear the address match status.

```
while (true) {
    if (i2c_slave_dma_read_interrupt_status(&i2c_slave_instance) &
                SERCOM_I2CS_INTFLAG_AMATCH) {
        i2c_slave_dma_write_interrupt_status(&i2c_slave_instance,
                SERCOM_I2CS_INTFLAG_AMATCH);
    }
}
```

10. Document Revision History

Doc. Rev.	Date	Comments
42116E	12/2015	Added support for SAM L21/L22, SAM DA1, SAM D09, and SAM C21
42116D	12/2014	Added support for 10-bit addressing and high speed in SAM D21. Added support for SAM R21 and SAM D10/D11.
42116C	01/2014	Added support for SAM D21
42116B	06/2013	Corrected documentation typos. Updated I^2C Bus State Diagram.
42116A	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**