APPLICATION NOTE

AT07335: SAM4 TWI Slave Mode Driver

ASF PROGRAMMERS MANUAL

SAM4 TWI Slave Mode Driver

This driver provides access to the main features of the TWIS The TWIS interconnects components on a unique two-wire bus in a manner similar to I^2C.

The TWIS is programmable as a slave with sequential or single-byte access. High speed mode capability is also supported. This document does not provide any information regarding SMBus usage.

Table of Contents

Software License

1. Prerequisites

There are no prerequisites for this module.

2. Module Overview

The outline of this section is as follows:

- Functional Description
- TWI Bus Topology

2.1 Functional Description

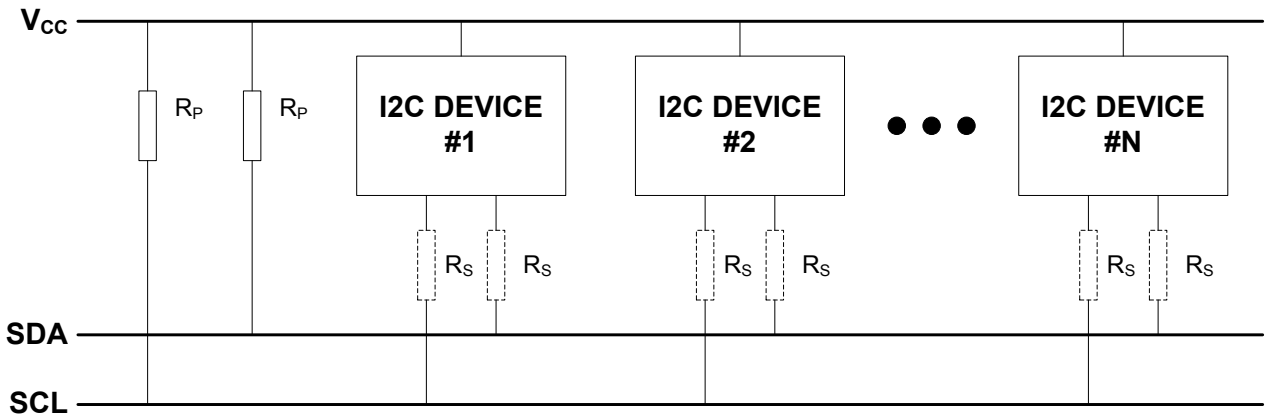
Driver for the TWIS (Two-Wire aka I^2C Slave Interface). This driver provides access to the main features of the TWIS controller. The two wire interface connects components on a two-wire bus, each having a unique ID. The TWI modules is programmable as a slave with sequential or single-byte access. High speed mode capability is also supported.

The TWI 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.

2.2 TWI Bus Topology

Figure 2-1: I2C Bus Topology on page 6. The pull-up resistors (R_s) will provide a high level on the bus lines when none of the TWI devices are driving the bus. These are optional, and can be replaced with a constant current source.

Figure 2-1. I2C Bus Topology



Note: R_S is optional

3. Special Considerations

NONE.

4. Extra Information

4.1 Terms and Definitions

Term	Definition
Standard speed	Original 100kHz clock
Fast Mode plus	1MHz clock
High spead mode	3.4MHz clock
Ultra Fast mode	5MHz clock

5. Examples

- Quickstart guide for SAM TWIS driver
- TWI Slave Example

6. API Overview

6.1 Variable and Type Definitions

6.1.1 Type twis_callback_t

```
typedef struct twis_callback twis_callback_t
```

6.1.2 Type twis_error_callback_t

```
typedef void(* twis_error_callback_t )(void)
```

6.1.3 Type twis_interrupt_source_t

```
typedef enum twis_interrupt_source twis_interrupt_source_t
```

TWIS interrupt source type.

6.1.4 Type twis_rx_callback_t

```
typedef void(* twis_rx_callback_t )(uint8_t)
```

6.1.5 Type twis_stop_callback_t

```
typedef void(* twis_stop_callback_t )(void)
```

6.1.6 Type twis_tx_callback_t

```
typedef uint8_t(* twis_tx_callback_t )(void)
```

6.2 Structure Definitions

6.2.1 Struct twis_callback

Table 6-1. Members

Type	Name	Description
twis_error_callback_t	error	Routine to handle bus error
twis_rx_callback_t	rx	Routine to receive data from TWI master

Type	Name	Description
twis_stop_callback_t	stop	Routine to signal a TWI STOP
twis_tx_callback_t	tx	Routine to transmit data to TWI master

6.2.2 Struct twis_config

Table 6-2. Members

Type	Name	Description
bool	ack_general_call	Acknowledge the general call address
bool	ack_slave_addr	Acknowledge the specified slave address
bool	ack_smbus_default_addr	Acknowledge the SMBus default address
bool	ack_smbus_host_header	Acknowledge the SMBus host header
uint32_t	chip	The desired address.
bool	enable_pec	Enable packet error checking
uint8_t	exp	Clock Prescaler
uint8_t	fs_dadrivel	Data Drive Strength Low in F/S mode
uint8_t	fs_daslew	Data Slew Limit in F/S mode
uint8_t	fs_filter	Input Spike Filter Control in F/S mode
uint8_t	hddat	Data Hold Cycles
uint8_t	hs_dadrivel	Data Drive Strength Low in HS mode
uint8_t	hs_daslew	Data Slew Limit in HS mode
uint8_t	hs_filter	Input Spike Filter Control in HS mode
bool	smbus	SMBUS mode
bool	stretch_clk_addr	Stretch clock on address match
bool	stretch_clk_data	Stretch clock on data byte reception
bool	stretch_clk_hr	Stretch clock if RHR is full or THR is empty
uint8_t	sudat	Data Setup Cycles
bool	ten_bit	Ten-bit addressing
uint8_t	tlows	SMBus Low:Sext Cycles
uint8_t	ttouts	SMBus TIMEOUT Cycles

6.2.3 Struct twis_dev_inst

6.2.3.1 asfdoc_sam_twis_structure

Device instance structure for a TWI Slave driver instance. This structure should be initialized by the twis_init() function to associate the instance with a particular hardware module of the device.

Table 6-3. Members

Type	Name	Description
Twis *	hw_dev	Base address of the TWIS module.
struct twis_config *	twis_cfg	Pointer to TWIS configuration structure.

6.3 Macro Definitions

6.3.1 Macro TWIS_WAIT_TIMEOUT

```
#define TWIS_WAIT_TIMEOUT
```

6.4 Function Definitions

6.4.1 Function twis_clear_status()

Clear the current status of the TWIS.

```
void twis_clear_status(
  struct twis_dev_inst *const dev_inst,
  uint32_t clear_status)
```

Table 6-4. Parameters

Data direction	Parameter name	Description
[out]	dev_inst	Device structure pointer
[in]	clear_status	The TWIS status to be clear

6.4.2 Function twis_disable()

Disable TWIS Module.

```
void twis_disable(
  struct twis_dev_inst *const dev_inst)
```

Table 6-5. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

6.4.3 Function twis_disable_interrupt()

Disable the TWIS interrupts.

```
void twis_disable_interrupt(
  struct twis_dev_inst *const dev_inst,
  twis_interrupt_source_t interrupt_source)
```

Table 6-6. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[in]	interrupt_source	The TWIS interrupt to be disabled

6.4.4 Function twis_enable()

Enable TWIS Module.

```
void twis_enable(
  struct twis_dev_inst *const dev_inst)
```

Table 6-7. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

6.4.5 Function twis_enable_interrupt()

Enable the TWIS interrupts.

```
void twis_enable_interrupt(
  struct twis_dev_inst *const dev_inst,
  twis_interrupt_source_t interrupt_source)
```

Table 6-8. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[in]	interrupt_source	The TWIS interrupt source to be enabled

6.4.6 Function twis_get_config_defaults()

Get the TWIS master default configurations.

```
void twis_get_config_defaults(
  struct twis_config *const cfg)
```

Use to initialize the configuration structure to known default values. This function should be called at the start of any TWIS initiation.

The default configuration is as follows:

- 7-bit addressing
- Self address is 0x50

- Normal mode
- Do not stretch clock on data byte reception
- Do not stretch clock on address match
- Stretch clock if RHR is full or THR is empty
- Do not acknowledge the general call address
- Acknowledge the specified slave address
- Disable packet error checking
- Do not acknowledge the SMBus host header
- Do not acknowledge the SMBus default address
- 0 data setup cycles in F/S mode and high speed mode
- Zero-initialization for slew rate setting in F/S mode and high speed mode
- Clock Prescaler is 0
- 0 SMBus TIMEOUT cycle
- 0 SMBus Low:Sext cycle

Table 6-9. Parameters

Data direction	Parameter name	Description
[out]	cfg	Pointer to configuration structure to be initiated

6.4.7 Function twis_get_interrupt_mask()

Get the TWIS interrupt mask.

```
uint32_t twis_get_interrupt_mask(
  struct twis_dev_inst *const dev_inst)
```

Table 6-10. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

Returns TWIS interrupt mask.

6.4.8 Function twis_get_smbus_pec()

Get the calculated PEC value. Only for SMBus mode.

```
uint8_t twis_get_smbus_pec(
  struct twis_dev_inst *const dev_inst)
```

Note Only for SMBus mode

Table 6-11. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

Returns Calculated PEC value

6.4.9 Function twis_get_smbus_transfer_nb()

Get the progress of the transfer in SMBus mode.

```
uint8_t twis_get_smbus_transfer_nb(
  struct twis_dev_inst *const dev_inst)
```

Note Only for SMBus mode

Table 6-12. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

Returns The left number of data bytes in the transmission.

6.4.10 Function twis_get_status()

Information about the current status of the TWIS.

```
uint32_t twis_get_status(
  struct twis_dev_inst *const dev_inst)
```

Table 6-13. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

Returns TWI Status.

6.4.11 Function twis_init()

Initialize the TWI Slave Module.

```
enum status_code twis_init(
  struct twis_dev_inst *const dev_inst,
  Twis *const twis,
  struct twis_config * config)
```

Table 6-14. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[in]	twis	Base address of the TWIS
[in]	config	Configuration for the TWIS

Returns Status of module initialization.

Table 6-15. Return Values

Return value	Description
STATUS_OK	The data is written correctly
STATUS_ERR_DENIED	Initialization failed due to the module was enable before.
STATUS_ERR_BUSY	Initialization failed due to the module is busy with transfer.

6.4.12 Function twis_read()

Get the last byte data received from TWI bus.

```
enum status_code twis_read(
  struct twis_dev_inst *const dev_inst,
  uint8_t * data)
```

Table 6-16. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[out]	data	The read data

Returns Status of write operation.

Table 6-17. Return Values

Return value	Description
STATUS_OK	The data is read correctly
STATUS_ERR_TIMEOUT	The read operation aborted due to timeout

6.4.13 Function twis_send_data_ack()

Enable ACK transfer in Slave Receiver Mode.

```
void twis_send_data_ack(
  struct twis_dev_inst *const dev_inst)
```

Table 6-18. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

6.4.14 Function twis_send_data_nack()

Enable NACK transfer in Slave Receiver Mode.

```
void twis_send_data_nack(
  struct twis_dev_inst *const dev_inst)
```

Table 6-19. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer

6.4.15 Function twis_set_callback()

Set callback for TWIS.

```
void twis_set_callback(
  struct twis_dev_inst *const dev_inst,
  twis_interrupt_source_t source,
  twis_callback_t callback,
  uint8_t irq_level)
```

Note Slave address match interrupt is enabled by default so that TWIS ISR can work appropriately.

Table 6-20. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[in]	source	Interrupt source
[in]	callback	Callback function pointer
[in]	irq_level	Interrupt level

6.4.16 Function twis_set_smbus_transfer_nb()

Set the total number of data bytes in the transmission. Only for SMBus mode.

```
void twis_set_smbus_transfer_nb(
  struct twis_dev_inst *const dev_inst,
  uint8_t nb,
  bool increment)
```

Note Only for SMBus mode

Table 6-21. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[in]	nb	Total number of data bytes in the transmission
[in]	increment	Count up per byte transferred if true, otherwise count down.

6.4.17 Function twis_write()

Write one byte data to TWI bus.

```
enum status_code twis_write(
  struct twis_dev_inst *const dev_inst,
  uint8_t byte)
```

Table 6-22. Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Device structure pointer
[in]	byte	The byte data to write

Returns Status of write operation.

Table 6-23. Return Values

Return value	Description
STATUS_OK	The data is written correctly
STATUS_ERR_TIMEOUT	The write operation aborted due to timeout

6.5 Enumeration Definitions

6.5.1 Enum twis_interrupt_source

TWIS interrupt source type.

Table 6-24. Members

Enum value	Description
TWIS_INTERRUPT_RX_BUFFER_READY	
TWIS_INTERRUPT_TX_BUFFER_READY	
TWIS_INTERRUPT_TRANS_COMP	
TWIS_INTERRUPT_UNDER_RUN	
TWIS_INTERRUPT_OVER_RUN	
TWIS_INTERRUPT_NAK_RECEIVED	

Enum value	Description
TWIS_INTERRUPT_SMBUS_TIMEOUT	
TWIS_INTERRUPT_SMBUS_PEC_ERROR	
TWIS_INTERRUPT_BUS_ERROR	
TWIS_INTERRUPT_SLAVEADR_MATCH	
TWIS_INTERRUPT_GENCALL_MATCH	
TWIS_INTERRUPT_SMBUS_HHADR_MATCH	
TWIS_INTERRUPT_SMBUS_DEFADR_MATCH	
TWIS_INTERRUPT_STOP_RECEIVED	
TWIS_INTERRUPT_RESTART_RECEIVED	
TWIS_INTERRUPT_BYTE_TRANS_FINISHED	
TWIS_INTERRUPT_ERRORS	
TWIS_INTERRUPT_ALL	

7. TWI Slave Example

7.1 Introduction

The application demonstrates how to use use the SAM TWIS peripheral.

7.2 Requirements

This package can be used with SAM4L boards.

In addition, another device will be needed to act as the TWI master. The twim_example can be used for that, in which case a second kit supported by that project is needed.

1. Connect TWD (SDA) for the two boards.
2. Connect TWCK (SCL) for the two boards.
3. Connect GND for the two boards.
4. Add a pull up resistor on TWD and TWCK.

7.3 Main Files

- twis.c SAM Two-Wire Slave Interface driver implementation
- twis.h SAM Two-Wire Slave Interface driver definitions
- twi_slave_example.c Example application

7.4 Description of the Example

After launching the program, the device will act as a simple TWI-enabled serial memory containing 50 bytes. This enables this project to be used with the twim_example project as the master.

To write in the memory, the TWI master must address the device first, then send two bytes containing the memory address to access. Additional bytes are treated as the data to write.

Reading is done in the same fashion, except that after receiving the memory address, the device will start outputting data until a STOP condition is sent by the master. The default address for the TWI slave is fixed to 0x50. If the board has a TWI component with this address, you can change the define TARGET_ADDRESS in conf_example.h of twim_example project, and then define SLAVE_ADDRESS in twi_slave_example.c of twi_slave_example project.

7.5 Compilation Info

This software was written for the GNU GCC and IAR(™) EWARM. Other compilers may or may not work.

8. Quickstart guide for SAM TWIS driver

This is the quickstart guide for the SAM TWIS driver with step-by-step instructions on how to configure and use the driver in a selection of use cases.

The use cases contain several code fragments. The code fragments in the steps for setup can be copied into a custom initialization function, while the steps for usage can be copied into, e.g., the main application function.

8.1 Basic Use Case

The basic use case demonstrate how to initialize the TWIS module and configure it.

8.1.1 Prerequisites

- System Clock Management (Sysclock).

8.2 Setup Steps

8.2.1 Example Code

Place the following at the start of your main function.

```
twis_callback_t slave_callbacks;

/* Initialize the SAM system */
sysclk_init();

/* Initialize the board */
board_init();

/* Initialize the console UART */
configure_console();
```

Then set the device as a TWI slave.

```
struct twis_config config;
twis_get_config_defaults(&config);
config.chip = SLAVE_ADDRESS;
```

Now initialise the TWI subsystem:

```
if (twis_init(&twis_device, BOARD_BASE_TWI_SLAVE, &config) == STATUS_OK) {
```

Set up the callback functions to handle reception, transmission, STOP condition and errors. In the end, enable the slave address match interrupt:

```
/* Set pointer to user specific application routines */
slave_callbacks.rx = &twis_slave_rx;
slave_callbacks.tx = &twis_slave_tx;
slave_callbacks.stop = &twis_slave_stop;
slave_callbacks.error = &twis_slave_error;
twis_set_callback(&twis_device, TWIS_INTERRUPT_SLAVEADR_MATCH,
    slave_callbacks, 1);
/* Enable the TWIS module */
twis_enable(&twis_device);
```

Index

Document Revision History

Doc. Rev.	Date	Comments
42273A	05/2014	Initial document release

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