AVR311: Using the TWI module as I²C slave

Features

- **C-code driver for TWI slave**
- **Compatible with Philips' I²C protocol**
- **Uses the hardware TWI module**
- **Interrupt driven transmission**
- **Supports both Standard mode and Fast mode**
- **Wake up from all sleep modes on own address recognition**

Introduction

The Two Wire serial Interface (TWI) is compatible with Philips' I²C protocol. The bus was developed to allow simple, robust and cost effective communication between integrated circuits in electronics. The strengths of the TWI bus includes the capability of addressing up to 128 devices on the same bus, arbitration, and the possibility to have multiple masters on the bus.

A hardware TWI module is included in most of the AVR devices available.

This application note describes a TWI slave implementation, in form of a full-featured driver and an example of usage for this driver. The driver handles transmission according to both Standard mode (<100kbps) and Fast mode (<400kbps).

8-bit AVR® Microcontrollers

Application Note

Theory

This section gives a short description of the TWI interface in general and the TWI module on the megaAVR's. For more detailed information refer to the datasheets.

Two-wire serial Interface
The Two-wire Serial Interface (TWI) is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 individually addressable devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

Figure 1. TWI Bus Interconnection.
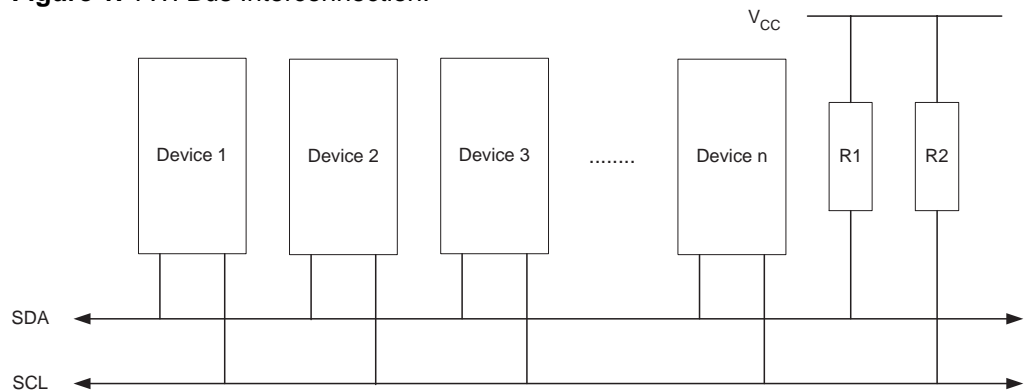
The TWI bus is a multi-master bus where one or more devices, capable of taking control of the bus, can be connected. Only Master devices can drive both the SCL and SDA lines while a Slave device is only allowed to issue data on the SDA line.

Data transfer is always initiated by a Bus Master device. A high to low transition on the SDA line while SCL is high is defined to be a START condition or a repeated start condition.

Figure 2. TWI Address and Data Packet Format
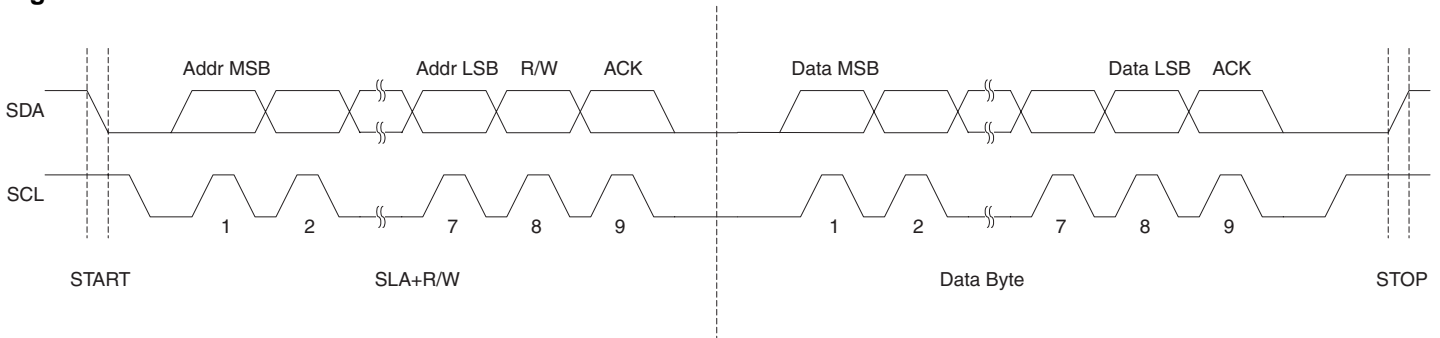
A START condition is always followed by the (unique) 7-bit slave address and then by a Data Direction bit. The Slave device addressed now acknowledges to the Master by holding SDA low for one clock cycle. If the Master does not receive any acknowledge the transfer is terminated. Depending of the Data Direction bit, the Master or Slave now transmits 8-bit of data on the SDA line. The receiving device then acknowledges the data. Multiple bytes can be transferred in one direction before a repeated START or a STOP condition is issued by the Master. The transfer is terminated when the Master issues a STOP condition. A STOP condition is defined by a low to high transition on the SDA line while the SCL is high.

If a Slave device cannot handle incoming data until it has performed some other function, it can hold SCL low to force the Master into a wait-state.

All data packets transmitted on the TWI bus are 9 bits long, consisting of one data byte and an acknowledge bit. During a data transfer, the master generates the clock and the START and STOP conditions, while the receiver is responsible for acknowledging the reception. An Acknowledge (ACK) is signaled by the receiver pulling the SDA line low during the ninth SCL cycle. If the receiver leaves the SDA line high, a NACK is signaled.

The AVR TWI Module

The TWI module is comprised of several sub modules, as shown in Figure 3. All registers drawn in a thick line are accessible through the AVR data bus.

Figure 3. Overview of the TWI module in the AVR devices.
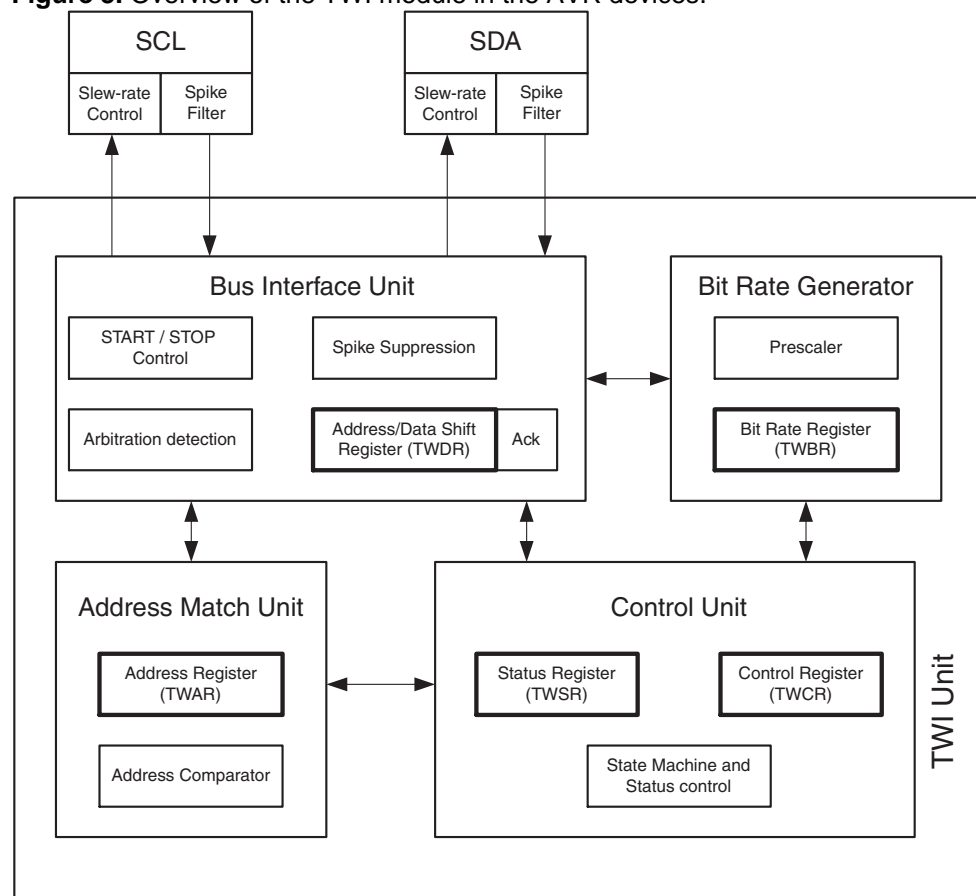
Control Unit

The AVR TWI module can operate in both Master and Slave mode. The mode of operation is distinguished by the TWI status codes in the TWI Status Register (TWSR) and by the use of certain bits in the TWI Control Register (TWCR).

A set of predefined status codes covers the different states that the TWI can be in when a TWI event occurs. The status codes are divided in Master and Slave codes and further in receive and transmit related codes. Status codes for Bus Error and Idle also exist.

The TWI module operates as a state machine and is event driven: if a START CONDITION followed by a TWI address matches the address in the Slave's TWI Address Register (TWAR) the TWINT flag is set, resulting in the execution of the corresponding interrupt (if Global Interrupt and TWI interrupts are enabled). The firmware of the Slave responds by reading the status code in TWSR and responding accordingly. All TWI events will set the TWINT flag, and the firmware must respond based on the status in TWSR.

As long as the TWINT Flag is set, the SCL line is held low. This allows the application software to complete its tasks before allowing the TWI transmission to continue.

The TWINT Flag is set in the following situations:

- After the TWI has transmitted a START/REPEATED START condition
- After the TWI has transmitted SLA+R/W
- After the TWI has transmitted an address byte
- After the TWI has lost arbitration
- After the TWI has been addressed by own Slave address or general call
- After the TWI has received a data byte
- After a STOP or REPEATED START has been received while still addressed as a Slave.
- When a bus error has occurred due to an illegal START or STOP condition

Bit Rate Generator

The Bite Rate Generator unit controls the period of SCL when operating in a Master mode. The SCL period is controlled by settings in the TWI Bit Rate Register (TWBR) and the Prescaler bits in the TWI Status Register (TWSR). Slave operation does not depend on Bit Rate or Prescaler settings, but the CPU clock frequency in the Slave must be at least 16 times higher than the SCL frequency. Table 1 shows minimum CPU clock speeds for normal and high speed TWI transmission.

Table 1. Minimum CPU clock frequencies versus SCL frequencies.

| CPU Clock frequency [MHz]	SCL frequency [kHz]
>6.4	400
>1.6	100

SCL and SDA Pins

Both TWI lines (SDA and SCL) are bi-directional, therefore outputs connected to the TWI bus must be of an open-drain or an open-collector type. Each line must be connected to the supply voltage via a pull-up resistor. A line is then logic high when none of the connected devices drives the line, and logic low if one or more drives the line low.

The output drivers contain a slew-rate limiter. The input stages contain a spike suppression unit removing spikes shorter than 50 ns. Note that the internal pull-ups in the AVR pads can be enabled by setting the PORT bits corresponding to the SCL and

SDA pins, as explained in the I/O Port section. The internal pull-ups can in some systems eliminate the need for external ones.

Figure 4 shows how to connect the TWI units to the TWI bus. The value of Rp depends on V_{CC} and the bus capacitance (typically 4.7 k).

Figure 4. TWI connection.
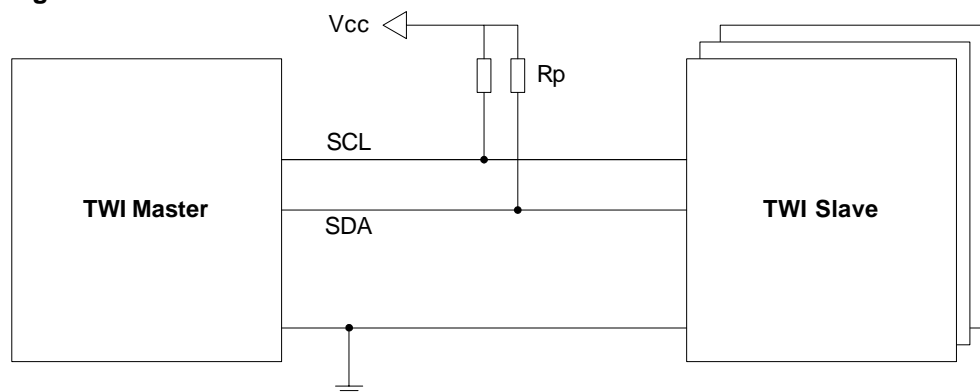
Address Match Unit

The Address Match unit is only used in slave mode, and checks if the received address bytes match the 7-bit address in the TWI Address Register (TWAR). Upon an address match, the Control Unit is informed, allowing correct action to be taken. The TWI may or may not acknowledge its address, depending on settings in the TWCR.

On devices with a TWI Address Mask Register (TWAMR) the Address Match unit can react on a masked subset of addresses.

Although the clock system to the TWI is turned off in all sleep modes, the interface can still acknowledge its own Slave address or the general call address by using the TWI Bus clock as a clock source. The part will then wake up from sleep and the TWI will hold the SCL clock low during the wake up and until the TWINT Flag is cleared.

Bus Interface Unit

This unit contains the Data and Address Shift Register (TWDR), a START/STOP Controller and Arbitration detection hardware. The TWDR contains the address or data bytes to be transmitted, or the address or data bytes received. In addition it also contains a register containing the (N)ACK bit to be transmitted or received. Note that after waking up from sleep the content of TWDR is undefined. I.e. on devices with multi slave address support it is not possible to use the TWDR content to determine what slave address that triggered the device to wake up from sleep.

The START/STOP Controller is responsible for generation and detection of START, REPEATED START, and STOP conditions. The START/STOP controller is able to detect START and STOP conditions even when the AVR MCU is in one of the sleep modes, enabling the MCU to wake up if addressed by a Master. If the TWI has initiated a transmission as Master, the Arbitration Detection hardware continuously monitors the transmission trying to determine if arbitration is in process. If the TWI has lost an arbitration, the Control Unit is informed. Correct action can then be taken and appropriate status codes generated.

Implementation

The implemented code in this application note is a pure slave driver. The TWI modules also support master operation. See "AVR315 TWI Master Implementation" for a sample of a master driver. The master and slave drivers could be merged to one combined master and slave driver, but this is not the scope of this application note.

The slave driver c-code consists of three files.

- TWI_Slave.c
- TWI_Slave.h
- Main.c

There is an example on how to use of the driver in the main.c file. The TWI_Slave.h file must be included in the main application and contains all function declarations and defines for all TWI status codes. The TWI status code defines can be used to evaluate error messages and to take appropriate actions. The file TWI_Slave.c contains all the driver functions.

Some devices have an additional TWI Address Mask Register (TWAMR) that enables a device to respond to several TWI slave addresses. A customized version of the standard implementation described here, is included in the application note attachment.

Functions

The driver consists of the TWI Interrupt Service Routine and several functions. All functions are available for use outside the driver file scope. Some of them are however also used internally by the driver it self. All functions in the driver are listed in Table 2. The actual code sizes for the functions, compiled with the IAR compiler are listed in Table 4.

Table 2. Description of functions in the TWI Slave driver.

Function name	Description

`void TWI_Slave_Initialise`
`( uchar ownSlaveAddress )`
	Call this function to set up the TWI slave to its initial standby state. Remember to enable interrupts from the main application after initializing the TWI. Pass both the slave address and the requrements for triggering on a general call in the same byte. Use e.g. this notation when calling this function:
	`TWI_Slave_Initialise((TWI_slaveAddress<<TWI_ADR_BITS)`
	`| (TRUE<<TWI_GEN_BIT));`
	The TWI module is configured to NACK on any requests. Use a TWI_Start_Transceiver function to start the TWI.

`void TWI_Start_Transceiver_With_Data` `( uchar *message, uchar messageSize)`	Call this function to send a prepared message, or start the Transceiver for reception. Include a pointer to the data to be sent if a SLA+W is received. The data will be copied to the TWI buffer. Also include how many bytes that should be sent. Note that unlike the similar Master function, the Address byte is not included in the message buffers. The function will hold execution (loop) until the TWI_ISR has completed with the previous operation, then initialize the next operation and return.
`void TWI_Start_Transceiver( )`	Call this function to start the Transceiver without specifing new transmission data. Usefull for restarting a transmission, or just starting the transceiver for reception. The driver will reuse the data previously put in the transceiver buffers. The function will hold execution (loop) until the TWI_ISR has completed with the previous operation, then initialize the next operation and return.

uchar TWI_Transceiver_Busy( )	Call this function to test if the TWI_ISR is busy transmitting.
uchar TWI_Get_State_Info( )	Call this function to fetch the state information of the previous operation. The function will hold execution (loop) until the TWI_ISR has completed with the previous operation. If there was an error, then the function will return the TWI State code.
uchar TWI_Get_Data_From_Transceiver (uchar *message, uchar messageSize)	Call this function to read out the received data from the TWI transceiver buffer. I.e. first call TWI_Start_Transceiver to get the TWI Transceiver to fetch data. Then Run this function to collect the data when they have arrived. Include a pointer to where to place the data and the number of bytes to fetch in the function call. The function will hold execution (loop) until the TWI_ISR has completed with the previous operation, before reading out the data and returning. If there was an error in the previous transmission the function will return the TWI State code.
__interrupt void TWI_ISR( )	This function is the Interrupt Service Routine (ISR), and automatically called when the TWI interrupt is triggered; that is whenever a TWI event has occurred. This function should not be called directly from the main application.

Table 3. Description of the driver register byte containing status information from the last transceiver operation. Available as bit fields within a byte.

TWI_statusReg	Description
TWI_statusReg.lastTransOK	Set to 1 when an operation has completed successfully.
TWI_statusReg.RxDataInBuf	This setting is only valid if lastTransOK is set to 1. Set to 1 when data has been received and stored in the transceiver buffer. I.e. if 0 then it was a transmission.
TWI_statusReg.genAddressCall	This setting is only valid if RxDataInBuf is set to 1. It is set to 1 when the reception was a General Call. I.e. if 0 then it was an Address Match.

Table 4. Code sizes with the IAR 3.10 compiler.

TWI Master functions	[bytes]
TWI_Slave_Initialise( )	12
TWI_Transceiver_Busy( )	6
TWI_Get_State_Info( )	12
TWI_Start_Transceiver_With_Data( )	68
TWI_Start_Transceiver ( )	8
TWI_ TWI_Get_Data_From_Transceiver ( )	62
TWI_ISR( )	162
	335

Figure 5 shows flowcharts of the process of receiving and transmitting data over the TWI interface through the drivers. Data is passed through parameters to the functions while the status of an operation is available trough a global status variable. Figure 6 contains the flowchart for the TWI driver it self. A more detailed description of the actions for each event/state in the TWI Interrupt Service Routine can be found in a flowchart in Figure 7. In Figure 7 the left column contains the different states/events

the TWI state machine can be in when entering the Interrupt. A case switch executes the different actions dependant on the cause of the interrupt call.

The transceiver uses only one transmission buffer. At any time a message from the master will be processed on this buffer. I.e. a Master Write will overwrite the content, while a Master Read will transmit the content of this buffer.

When calling the Start Transceiver function the complete message is copied into the transceiver buffer. Then it enables the TWI interrupt to initiate the transceiver. The Interrupt then takes care of the complete transmission and disables itself when the transmission is completed, or if an error state occurs. The driver can this way poll the interrupt enable bit to check if a transmission is complete. The main application is only allowed to access the global transceiver variables while the TWI transceiver is not busy. The interrupt stores eventual error states in a variable that is available for the main application through a function call.

Note that the driver puts the TWI module in passive mode after each transceiver operation. This is to enable the application to read and interpreted the message from the master before responding to any new requests. All new messages from the master coming before the TWI slave is restarted will therefore be NACKed on. It is therefore important that the master gives the slave enough time to respond before sending the next message.

In the "Combined" flowchart in Figure 5 there is suggested a place to add error handling code. Note that if this is not implemented in this example, then an error state will lead to a restart of the slave transceiver which could be a way of handling an error transmission.

Figure 5. Calling the TWI driver from the application.
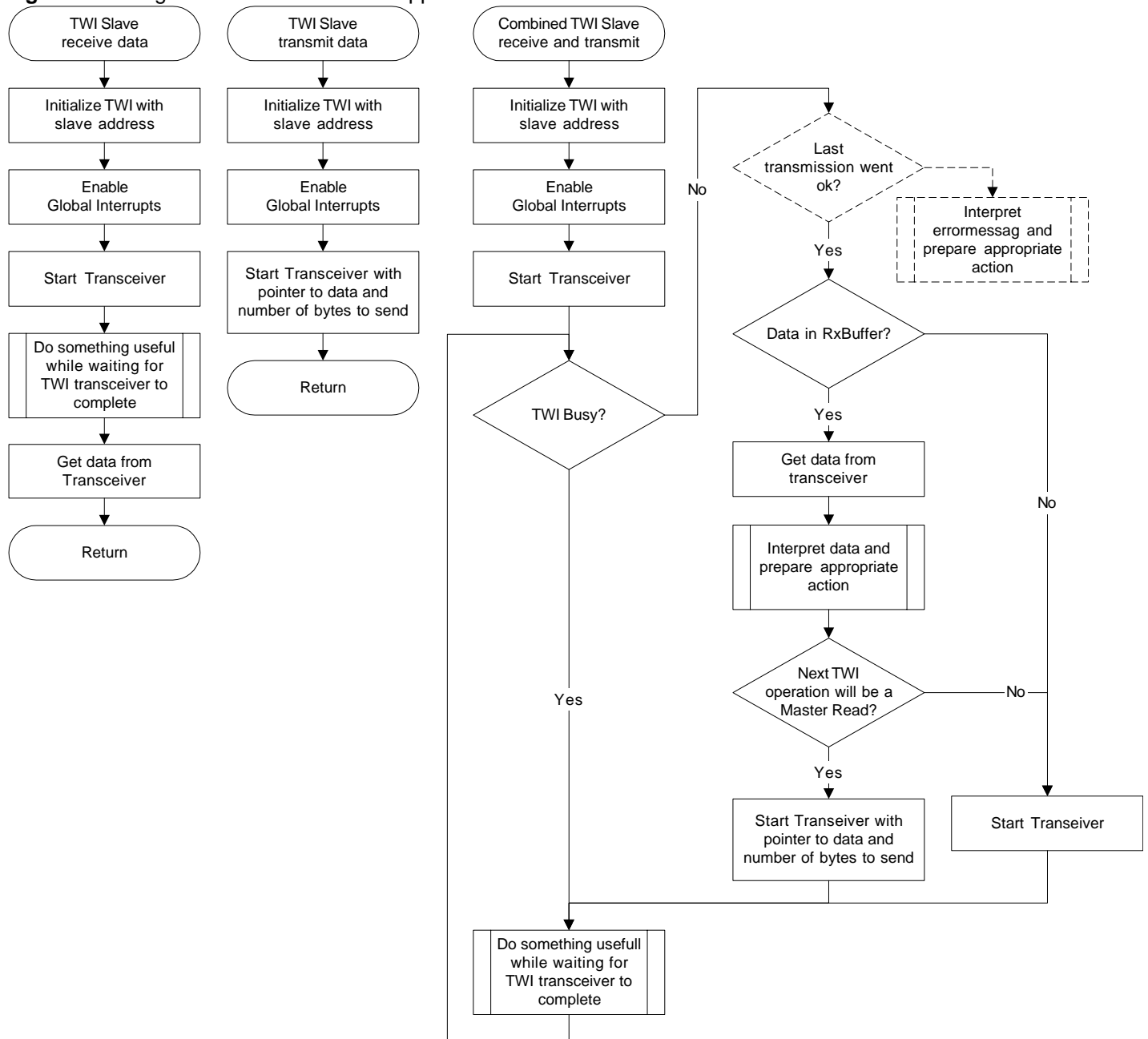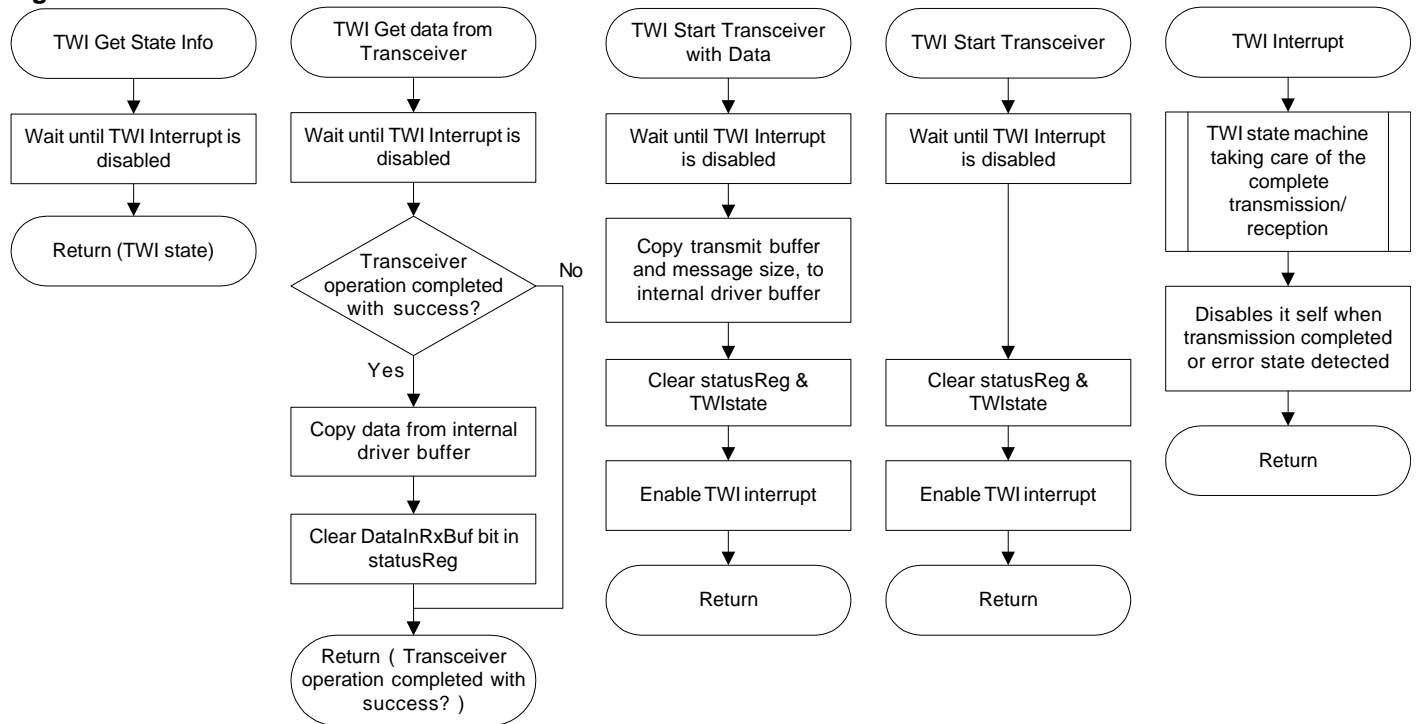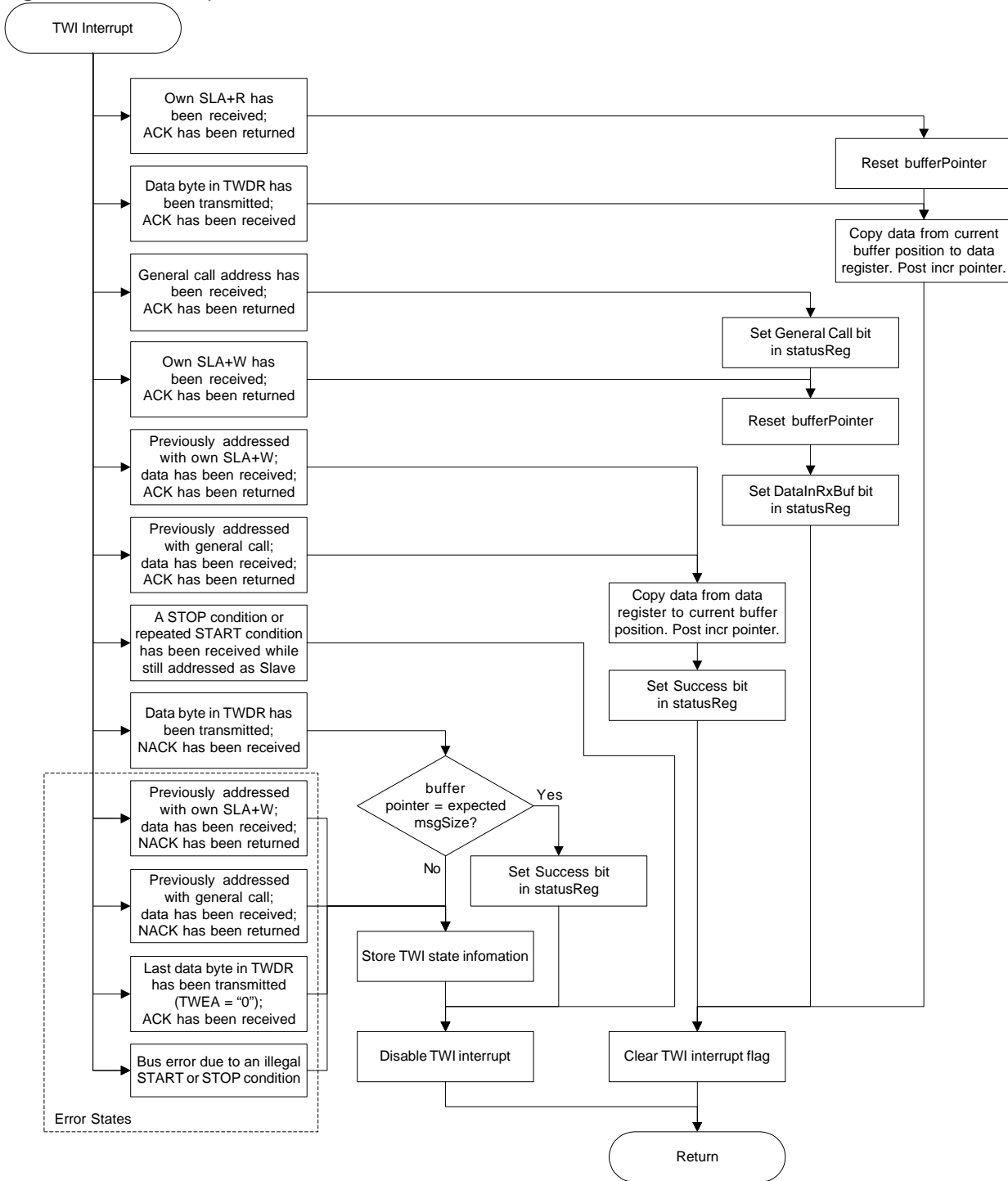
Figure 6. TWI driver functions.

Figure 7. TWI interrupt service routine

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/ High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature