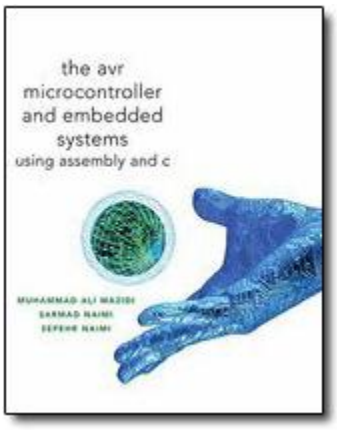
Serial Communications and I2C

Table of Contents



Chapter 18 "I2C Protocol and DS1307 RTC Interfacing"

There is a lot of material about I2C serial communications that is not covered in the following lecture material. The textbook does an excellent job of covering many additional topics related to this communications protocol.

ATMEL 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash
http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdf

- Chapter 18. SPI - Serial Peripheral Interface
- Chapter 19. USART 0
- Chapter 20. USART in SPI Mode
- Chapter 21. 2-wire Serial Interface

7-bit, 8-bit, and 10-bit I2C Slave Addressing by Total Phase

Serial Communications (I2C and SPI) Eugene Ho, Stanford University

Arduino Application Program Examples

- Texas Instruments PCA9535 and PCA9555 16-bit I/O port expanders: Arduino I2C Expansion I/O
- LIS3LV02DQ Accelerometer: Arduino and the Two-Wire Interface (TWI/I2C)

Introduction

Many embedded systems include peripheral devices connected to the microprocessor in order to expand its capabilities including

- DS1307 RTC (Real-Time Clock), Textbook section 18.4 “DS1307 RTC Interfacing and programming”
- Adafruit Motor Shield SPI
- HM6352 Digital Compass I2C
- BMA180 Triple Axis Accelerometer SPI or I2C
- ITG-3200 Triple-Axis Digital-Output Gyro SPI or I2C
- GPS module USART
- 1.44" LCD Display USART

Communication methods can be divided into the two categories parallel and serial. All of these peripherals interface with the microcontroller via a serial protocol. A protocol is the language that governs communications between systems or devices. Protocols may specify many aspects of inter-device communications including bit ordering, bit-pattern meanings, electrical connections, and even mechanical considerations in some cases.

The peripheral device may be completely passive; i.e., there is no controlling mechanism in place within the peripheral or complex enough to include an embedded controller, in which case the protocol may be more sophisticated.

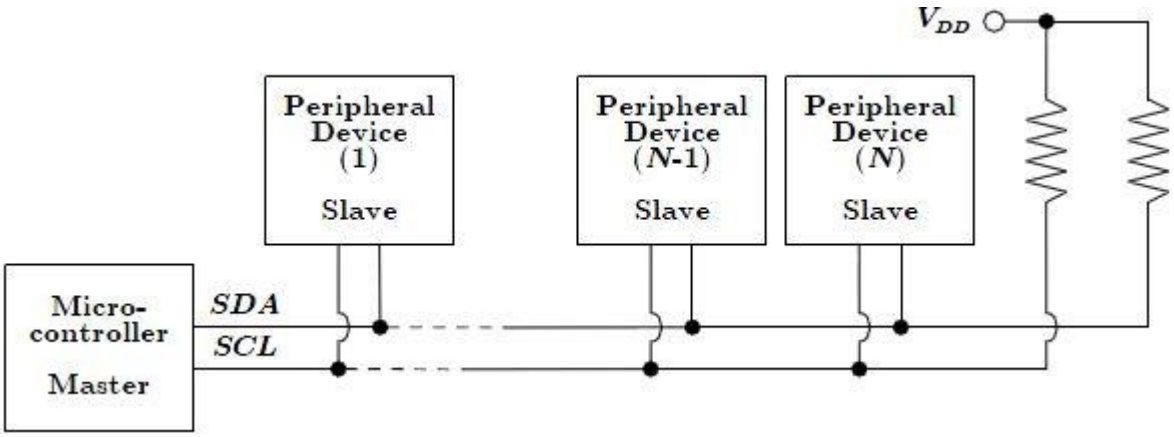
There are many questions that need to be answered when defining a serial communications protocol:

- In what **order** are the series of **bit**s shifted across the data line? Suppose the master transmits the most-significant bit (MSB) first, then the peripheral will receive the series in the order {0, 1, 0, 1, 0, 0, 1, 1}. Alternatively, the master could transmit the least-significant bit (LSB) first; in which case, the peripheral will receive the series {1, 1, 0, 0, 1, 0, 1, 0}. Either method is fine, but the peripheral and master must agree beforehand; otherwise, incorrect bytes will be received.
- What constitutes a **CLK** event? The master could use either a falling-edge or a rising-edge clock to specify a sampling signal to the peripheral device.
- How does the peripheral know when it is supposed to receive bytes and when it is supposed to transmit bytes?

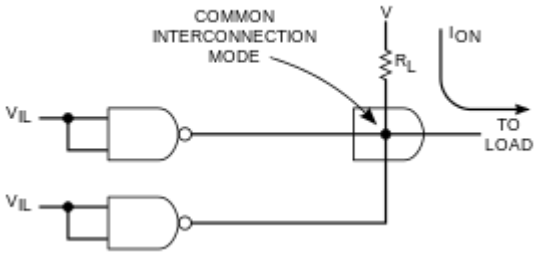
All of these questions and many more are answered by the protocol. We have already answered these questions for the SPI interface protocol, now it is time to look at the answers for the I2C interface protocol.

Inter-Integrated Circuit (I2C, IIC, TWI)

The Inter-Integrated Circuit (I2C or IIC) serial protocol was created by NXP Semiconductors, originally a Philips semiconductor division, to attach peripherals to an embedded microprocessor as shown here. I2C is a multi-point protocol in which peripheral devices are able to communicate along the serial interface which is composed of a **bidirectional serial data** line (SDA) and a bidirectional serial clock (SCL).



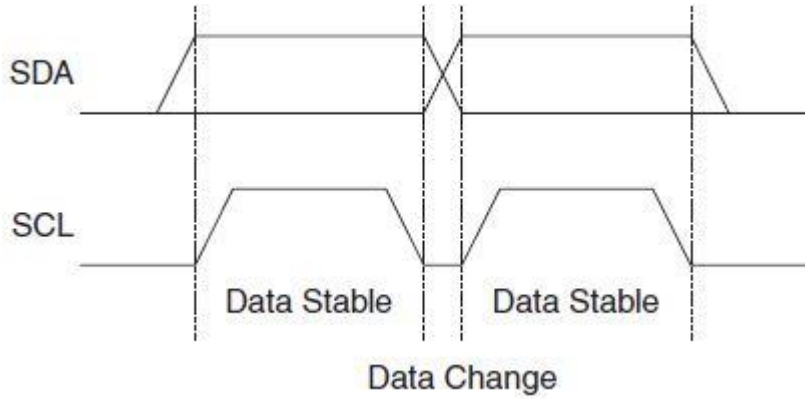
Electrical Interconnection beyond the wires is a single **4.7 kilohm pull-up resistor** for each of the I2C bus lines. Consequently, a slave device writing to SDA only needs to pull the line low for each '0'written; otherwise, it leaves the line alone to write a '1', which occurs due to the lines being pulled high externally by the pull-up resistor. This is commonly known as a **wired-AND** configuration.



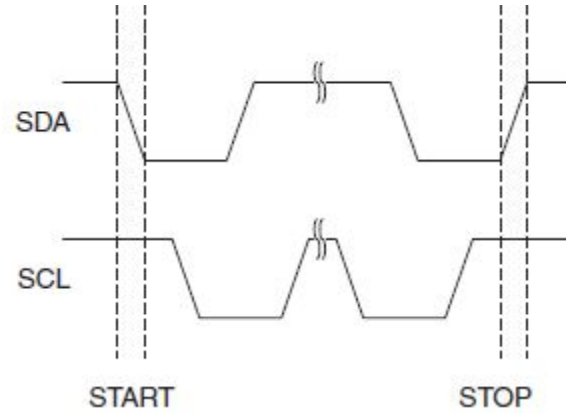
INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Source: [Wikipedia, Wired logic connection]

Transferring Bits on the I2C bus is accompanied by a **pulse on the clock** (SCL) **line**. The level of the data line must be stable when the clock line is high. The only exception to this rule is for generating start and stop conditions.



START and STOP conditions are signaled by changing the level of the SDA line when the SCL line is high.



All **packets** transmitted on the I2C bus are **9 bits long**.

- An **address packet** consists of 7 address bits (2^7 = 128 possible addresses), one READ/WRITE control bit and an acknowledge bit. An address packet consisting of a **SL**ave **A**ddress and a **R**EAD or a **W**RITE bit is called **SLA+R** or **SLA+W**, respectively. See Figure on the next page.
- A **data packet** consists of one data byte and an acknowledge bit.

How Many Slave Addresses?

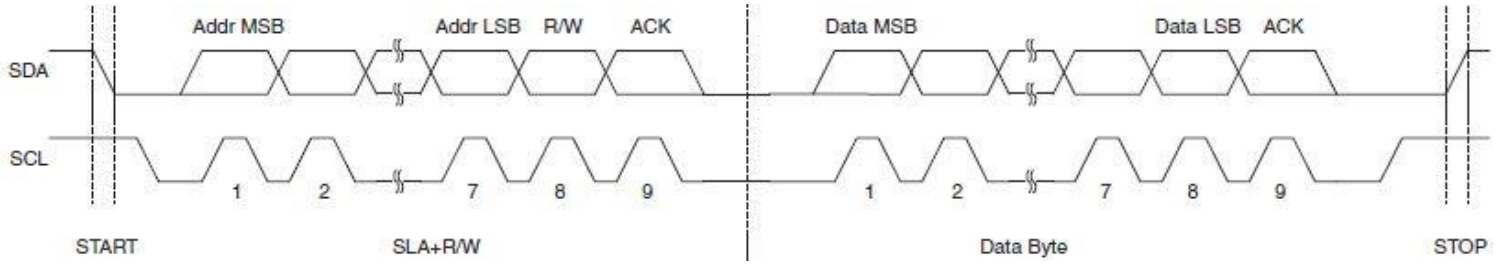
The Atmel datasheet for the ATmega328P states that the I2C supports up to 128 peripheral devices. The textbook claims up to 120 peripheral devices. To find the right answer we need to look at the NXP I2C-bus specification and user manual or this handy summary article. The I2C uses 7 bits to address peripheral devices. This means a maximum of 2^7 = 128 devices can be addressed (ATmega328 document). However, the I2C specification reserves 16 addresses (0x00 to 0x07 and 0x78 to 0x7F) for special purposes as defined in the following table. Therefore for practical purposes the I2C supports up to 128 – 16 = 112 peripheral devices. I am not sure how the textbook came up with 120.

Slave Address	R/W Bit	Description
000 0000	0	General call address
000 0000	1	START byte[1]
000 0001	x	CBUS address[2]
000 0010	x	Reserved for different bus format[3]
000 0011	x	Reserved for future purposes
000 01xx	x	Hs-mode master code
111 10xx	x	10-bit slave addressing
111 11xx	x	Reserved for future purposes

Thanks to Aaron Roudebush EE444 S'14 for answering this question.

Typical Data Transmission

The master begins communication by transmitting a single start bit followed by the unique 7-bit address of the slave device for which the master is attempting to access, followed by read/write bit. The corresponding slave device responds with an acknowledge bit if it is present on the serial bus. The master continues to issue clock events on the SCL line and either receives information from the slave or writes information to the slave depending on the read/write bit at the start of the session. The number of bits transferred during a single session is dependent upon the peripheral device and the agreed-upon higher-level protocol.

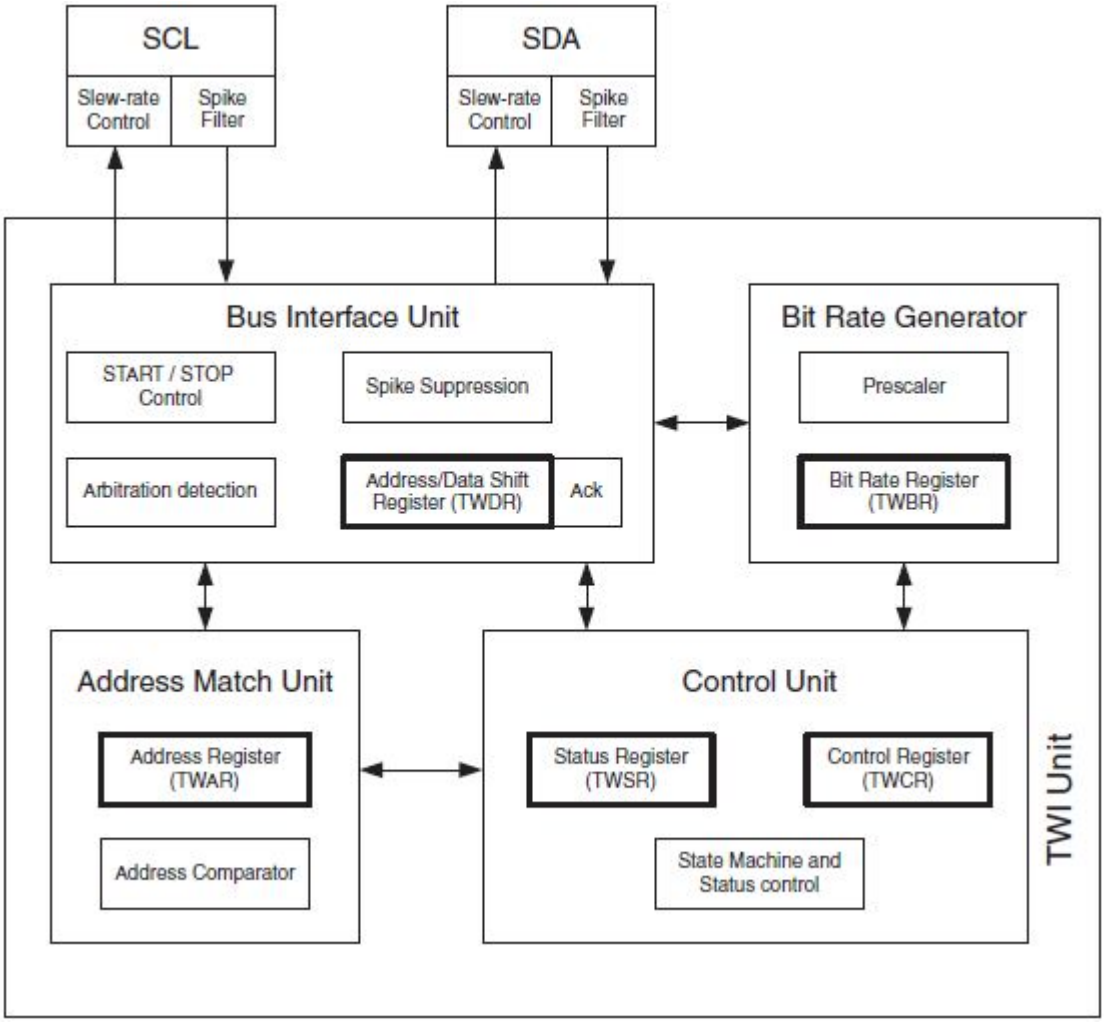


As an overview discussion of the I2C bus I am intentionally leaving out a number of interesting subjects including:

- **Clock Stretching** (textbook "Clock stretching" page 636)
- General call address 0000 000
- Receiver sending a **NACK** after the final byte. (textbook Example 18-4 page 635)
- Master and slave implementing a hand shake to adjust the clock speed to accommodate the slave.
- Multi-master Bus Systems, **Arbitration** and Synchronization (textbook "Arbitration" page 636, and "TWI Programming with Checking Status Register" page 668)

Overview of the ATmega328P TWI Module

The ATmega328P provides an I2C serial interface via the 2-wire Serial Interface (TWI) module. The bus allows for up to 128 different slave devices (textbook says 120) and up to **400 kHz data transfer speed**. The TWI provides an interrupt-based system in which an interrupt is issued after all bus events such as reception of a byte or transmission of a start condition. The TWI module is comprised of several submodules, as shown here.



All registers drawn in a thick line are accessible through the AVR data bus.

SCL and SDA Pins contain a slew-rate limiter in order to conform to the TWI specification. The input stages contain a spike suppression unit removing **spikes shorter than 50 ns**. Enabling the internal pull-ups in the GPIO PORT pins corresponding to SCL and SDA can in some systems eliminate the need for external pull-up resistors (20 – 50 kilohm versus 4.7 kilohm specification).

Bit 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).

Bit	7	6	5	4	3	2	1	0	
(0xB8)	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0	TWBR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \cdot (PrescalerValue)}$$

TWPS1	TWPS0	Prescaler Value
0	0	1
0	1	4
1	0	16
1	1	64

Bus Interface 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.

Bit	7	6	5	4	3	2	1	0	
(0xBB)	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0	TWDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	1	1	1	1	1	1	1	

Address Match Unit checks if received address bytes match the seven-bit address in the

TWI Address Register (TWAR) when the ATmega328P is acting as a slave device. The TWGCE bit enables the recognition of a General Call.

Bit	7	6	5	4	3	2	1	0	
(0xBA)	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	TWAR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	1	1	1	1	1	1	0	

Control Unit monitors the TWI bus and generates responses corresponding to settings in the

TWI Control Register (TWCR).

Bit	7	6	5	4	3	2	1	0	
(0xBC)	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

When an event requiring the attention of the application occurs on the TWI bus, the TWI Interrupt Flag (TWINT) is asserted. In the next clock cycle, the TWI Status Register (TWSR) is updated with a status code identifying the event.

Bit	7	6	5	4	3	2	1	0	
(0xB9)	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	TWSR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	1	1	1	1	1	0	0	0	

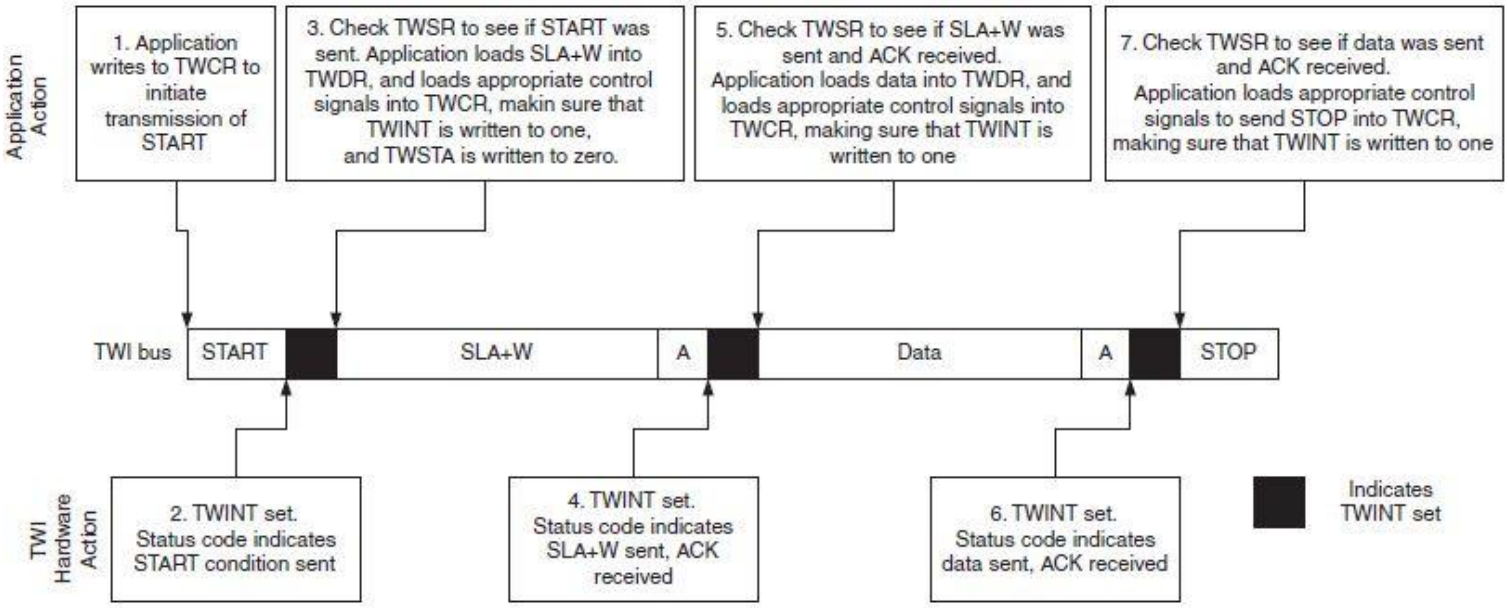
The TWSR only contains relevant status information when the TWI Interrupt Flag is asserted. At all other times, the TWSR contains a special status code indicating that no relevant status information is available.

Program Examples

AVR TWI in Master Transmitter Operating Mode

Timeline

The following timeline shows the interplay of registers comprising the TWA module. A detailed description of the registers and timeline is outside the scope of this overview article and the interested reader is encouraged to read Section 21.6 "Using the TWI" in the ATmega Datasheet.



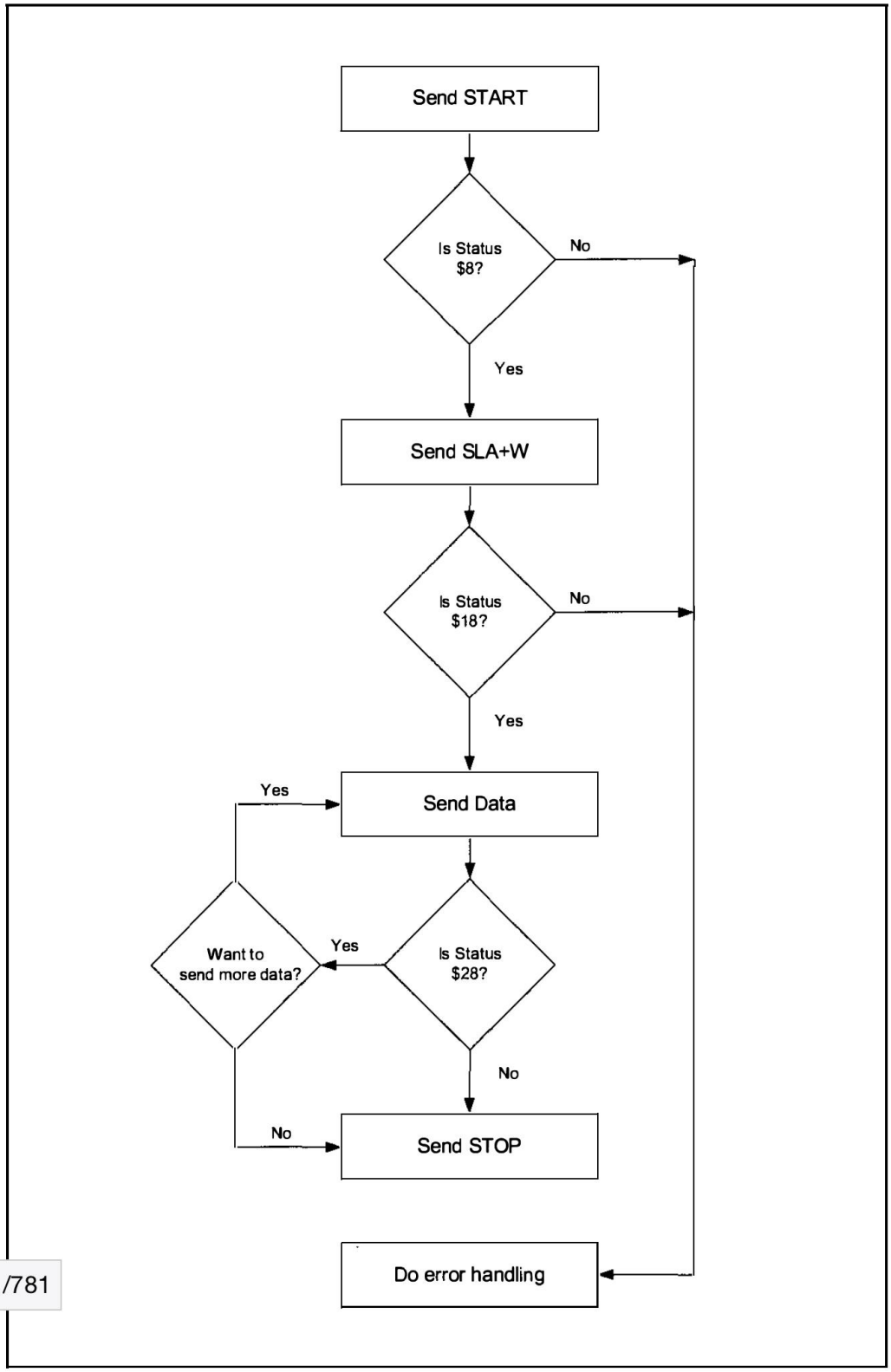
Notes:

The **TW**I **INT**errupt (TWINT) flag bit is set by hardware when the TWI has finished its current job and expects application software response. The TWINT Flag must be **cleared by software by writing a logic one** to it.

The application writes the **TW**I **STA**RT condition (TWSTA) bit to one when it desires to become a Master on the 2-wire Serial Bus. The TWI hardware checks if the bus is available, and generates a START condition on the bus if it is free. TWSTA **must be cleared by software** when the START condition has been transmitted.

Flowchart

Textbook: Section 18.5: TWI Programming with Checking Status Register (page 668) Figure 18-18.



676 /781

C Code

Code source: ATmega Datasheet Section 21.6 “Using the TWI”

```
//Send START condition
TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
// Wait for TWINT Flag set. This indicates that the START condition has been transmitted
while (!(TWCR & (1<<TWINT)))
;
// Check value of TWI Status Register. Mask prescaler bits.
// If status different from START go to ERROR
if ((TWSR & 0xF8) != START) ERROR();
// Load SLA_W into TWDR Register. Clear TWINT bit in TWCR to start transmission of address
TWDR = SLA_W;
TWCR = (1<<TWINT)|(1<<TWEN);
// Wait for TWINT Flag set. This indicates that the SLA+W has been transmitted,
// and ACK/NACK has been received.
while (!(TWCR & (1<<TWINT)))
;
// Check value of TWI Status Register. Mask prescaler bits. If status
// different from MT_SLA_ACK go to ERROR
if ((TWSR & 0xF8) != MT_SLA_ACK) ERROR();
// Load DATA into TWDR Register. Clear TWINT bit in TWCR to start transmission of data
TWDR = DATA;
TWCR = (1<<TWINT)|(1<<TWEN);
// Wait for TWINT Flag set. This indicates that the DATA has been transmitted,
// and ACK/NACK has been received.
while (!(TWCR & (1<<TWINT)))
;
// Check value of TWI Status Register. Mask prescaler bits. If status
// different from MT_DATA_ACK go to ERROR
if ((TWSR & 0xF8) != MT_DATA_ACK) ERROR();
// Transmit STOP condition
TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO);
```

HMC6352 Digital Compass Arduino Example

source: Compass Heading HMC6352 sparkfun

I2C HMC6352 compass heading (Sparkfun breakout) by BARRAGAN (http://barraganstudio.com)
Demonstrates use of the Wire library reading data from the HMC6352 compass heading

```
#include <Wire.h>

int compassAddress = 0x42 >> 1; // From datasheet compass address is 0x42
// shift the address 1 bit right, the Wire library only needs the 7
// most significant bits for the address
int reading = 0;

void setup()
{
Wire.begin(); // join i2c bus (address optional for master)
Serial.begin(9600); // start serial communication at 9600bps
pinMode(48, OUTPUT);
digitalWrite(48, HIGH);
}

void loop()
{
// step 1: instruct sensor to read echoes
Wire.beginTransmission(compassAddress); // transmit to device
// the address specified in the datasheet is 66 (0x42)
// but i2c adressing uses the high 7 bits so it's 33
Wire.send('A');         // command sensor to measure angle
Wire.endTransmission(); // stop transmitting

// step 2: wait for readings to happen
delay(10); // datasheet suggests at least 6000 microseconds

// step 3: request reading from sensor
Wire.requestFrom(compassAddress, 2); // request 2 bytes from slave device #33

// step 4: receive reading from sensor
if(2 <= Wire.available())  // if two bytes were received
{
reading = Wire.receive();  // receive high byte (overwrites previous reading)
reading = reading << 8;    // shift high byte to be high 8 bits
reading += Wire.receive(); // receive low byte as lower 8 bits
reading /= 10;
Serial.println(reading);   // print the reading
}

delay(500); // wait for half a second
}
```

More Arduino I2C Sketch Examples

1. Texas Instruments PCA9535 and PCA9555 16-bit I/O port expanders: Arduino I2C Expansion I/O
2. LIS3LV02DQ Accelerometer: Arduino and the Two-Wire Interface (TWI/I2C)
3. ITG 3200 Triple-Axis Digital-Output Gyro: http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1278367409/all