CHAPTER 18

I2C PROTOCOL AND DS1307 RTC INTERFACING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Understand the Inter-Integrated Circuit (I2C) protocol**
- **>> Explain how the I2C read and write operations work**
- **>> Examine the I2C pins SCK and SCL**
- **>> Explain the function of I2C (TWI) registers in AVR**
- **>> Code programs in Assembly and C for I2C (TWI)**
- **>> Explain how the real-time clock (RTC) chip works**
- **>> Explain the function of the DS1307 RTC pins**
- **>> Explain the function of the DS1307 RTC registers**
- **>> Understand the interfacing of the DS1307 RTC to the AVR**
- **>> Code programs to display time and date in Assembly and C**

This chapter discusses the I2C bus and shows the interfacing of the DS1307 real-time clock (RTC), an I2C chip. In Section 18.1, we describe the I2C bus and focus on I2C terminology and protocols. In Section 18.2, we describe the registers of AVR associated with I2C. In Section 18.3, we show how to write a simple program in Assembly and C to use the I2C features of AVR. In Section 18.4, we describe the DS1307 RTC's pin functions and show its interfacing and programming with the AVR. Advanced programming of the I2C (TWI) is discussed in Section 18.5.

SECTION 18.1: I2C BUS PROTOCOL

The IIC (Inter-Integrated Circuit) is a bus interface connection incorporated into many devices such as sensors, RTC, and EEPROM. The IIC is also referred to as I2C (I^2C) or I square C in many technical literatures. In this section we examine the pins of the I2C bus and focus on I2C terminology and protocols.

I2C bus

The I2C bus was originally started by Philips, but in recent years has become a widely used standard adapted by many semiconductor chip companies. I2C is ideal for attaching low-speed peripherals to a motherboard or embedded system or anywhere that a reliable communication over a short distance is required. As we will see in this chapter, I2C provides a connection-oriented communication with acknowledge. I2C devices use only 2 pins for data transfer, instead of the 8 or more pins used in traditional buses. They are called SCL (Serial Clock), which synchronize the data transfer between two chips, and SDA (Serial Data). This reduction of communication pins reduces the package size and power consumption drastically, making them ideal for many applications in which space is a major concern. These two pins, SDA and SCK, make the I2C a 2-wire interface. In many application notes, including AVR datasheets, I2C is referred to as *Two-Wire Serial Interface (TWI)*. In this chapter we use I2C and TWI interchangeably.

I2C line electrical characteristics

I2C devices use only 2 bidirectional open-drain pins for data communication. To implement I2C, only a 4.7 kilohm pull-up resistor for each of bus lines is needed (see Figure 18-1). This implements a wired-AND, which is needed to implement I2C protocols. This means that if one or more devices pull the line to

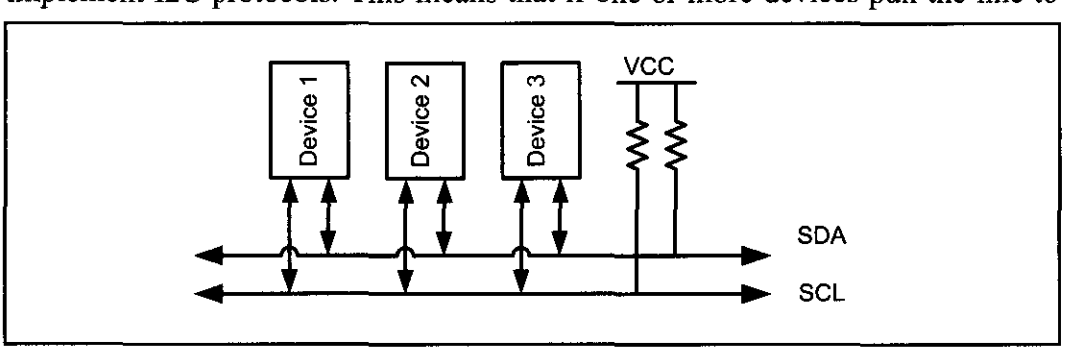


Figure 18-1. I2C Bus

low (zero) level, the line state is zero and the level of line will be 1 only if none of devices pull the line to low level.

I2C nodes

In the AVR up to 120 different devices can share an I2C bus. Each of these devices is called a *node*. In I2C terminology, each node can operate as either master or slave. Master is a device that generates the clock for the system; it also initiates and terminates a transmission. Slave is the node that receives the clock and is addressed by the master. In I2C, both master and slave can receive or transmit data, so there are four modes of operation. They are master transmitter, master receiver, slave transmitter, and slave receiver. Notice that each node can have more than one mode of operation at different times, but it has only one mode of operation at a given time. See Example 18-1.

Example 18-1

Give an example to show how a device (node) can have more than one mode of operation.
Solution:

If you connect the AVR to an EEPROM with I2C, the AVR does a master transmit operation to write to EEPROM. The AVR also does master receive operations to read from EEPROM. In the following sections, you will see that a node can do the operations of master and slave at different times.

Bit format

I2C is a synchronous serial protocol; each data bit transferred on the SDA line is synchronized by a high-to-low pulse of clock on the SCL line. According to I2C protocols the data line cannot change when the clock line is high; it can change only when the clock line is low. See Figure 18-2. The STOP and START conditions are the only exceptions to this rule.

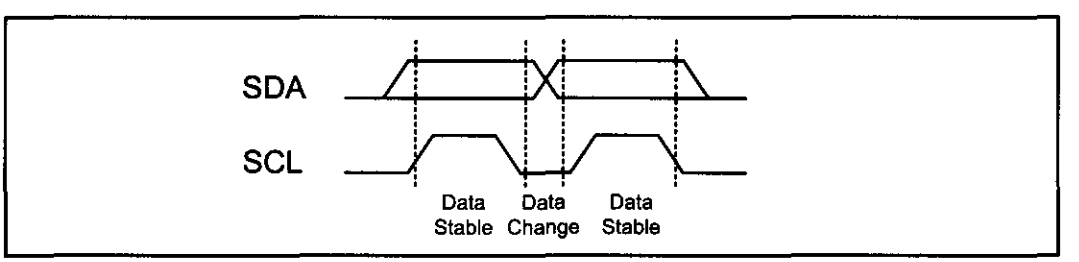


Figure 18-2. I2C Bit Format

START and STOP conditions

As we mentioned before, I2C is a connection-oriented communication protocol. This means that each transmission is initiated by a START condition and is terminated by a STOP condition. Remember that the START and STOP conditions are generated by the master.

STOP and START conditions must be distinguished from bits of address or data. That is why they do not obey the bit format rule that we mentioned before.

START and STOP conditions are generated by keeping the level of the SCL line high and then changing the level of the SDA line. The START condition is generated by a high-to-low change in the SDA line when SCL is high. The STOP condition is generated by a low-to-high change in the SDA line when SCL is low. See Figure 18-3.

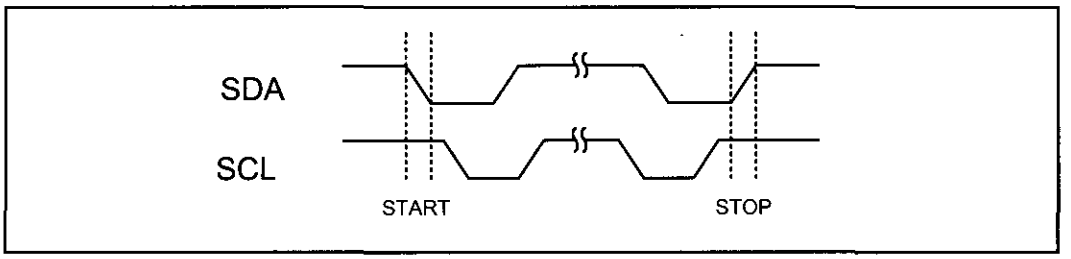


Figure 18-3. START and STOP Conditions

The bus is considered busy between each pair of START and STOP conditions, and no other master tries to take control of the bus when it is busy. If a master, which has the control of the bus, wishes to initiate a new transfer and does not want to release the bus before starting the new transfer, it issues a new START condition between a pair of START and STOP conditions. It is called the REPEATED START condition. See Figure 18-4.

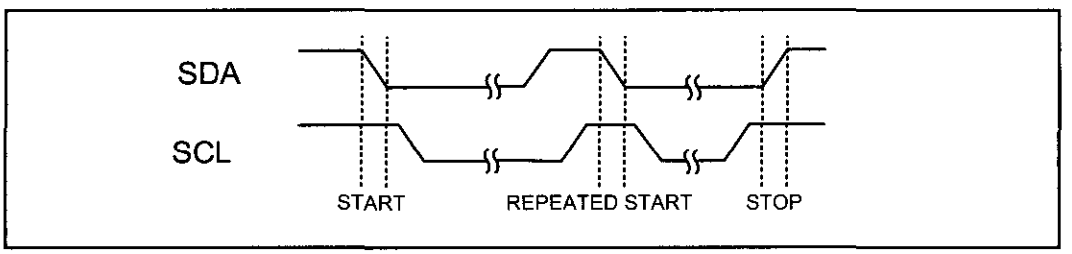


Figure 18-4. REPEATED START Condition

Example 18-2 shows why the REPEATED START condition is necessary.

Example 18-2

Give an example to show when a master must use the REPEATED START condition. What will happen if the master does not use it?

Solution:

If you connect two AVRs (AVR A and AVR B) and an EEPROM with I2C, and AVR A wants to display the addition of the contents of addresses 0x34 and 0x35 of EEPROM, it has to use the REPEATED START condition. Let's see what may happen if AVR A does not use the REPEATED START condition. AVR A transmits a START condition, reads the content of address 0x34 of EEPROM into R16, and transmits a STOP condition to release the bus. Before AVR A reads the contents of address 0x35 into R17, AVR B seizes the bus and changes the contents of addresses 0x34 and 0x35 of EEPROM. Then AVR A reads the content of address 0x35 into R17, adds it to R16, and displays the result on the LCD. The result on the LCD is neither the sum of the old values of addresses 0x34 and 0x35 nor the sum of the new values of addresses 0x34 and 0x35 of EEPROM!

Packet format in I2C

In I2C, each address or data to be transmitted must be framed in a packet. Each packet is 9 bits long. The first 8 bits are put on the SDA line by the transmitter, and the 9th bit is an acknowledge by the receiver or it may be NACK (not acknowledge). The clock is generated by the master, regardless of whether it is the transmitter or receiver. To get an acknowledge, the transmitter releases the SDA line during the ninth clock so that the receiver can pull the SDA line low to indicate an ACK. If the receiver doesn't pull the SDA line low, it is considered as NACK. See Figure 18-5.

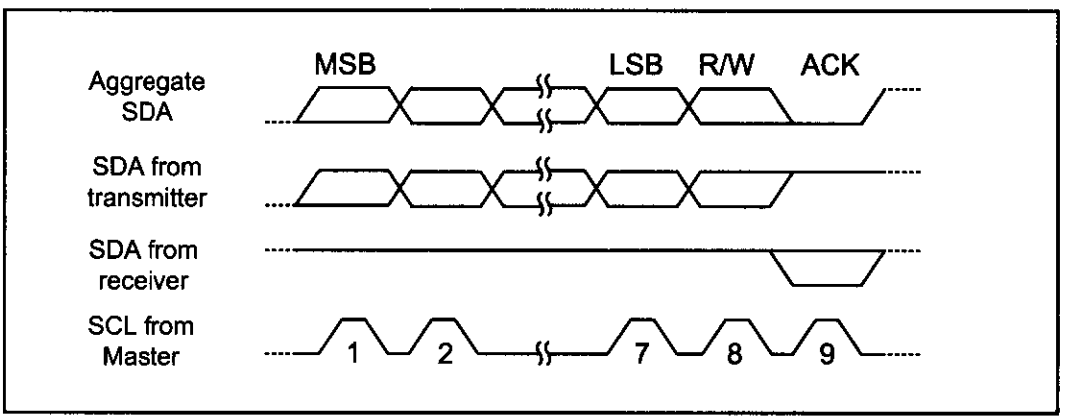


Figure 18-5. Packet Format in I2C

In I2C, each packet may contain either address or data. Also notice that START condition + address packet + one or more data packet + STOP condition together form a complete data transfer. Next we will study address and data packet formats and how to combine them to make a complete transmission.

Address packet format

Like any other packets, all address packets transmitted on the I2C bus are nine bits long. An address packet consists of seven address bits, one READ/WRITE control bit, and an acknowledge bit (see Figure 18-6).

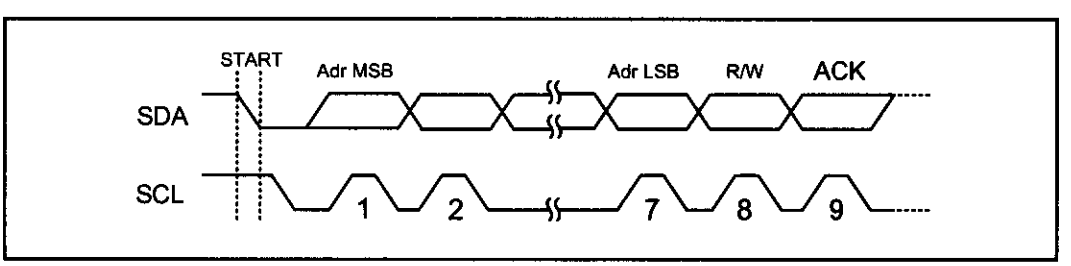


Figure 18-6. Address Packet Format in I2C

Address bits are used to address a specific slave device on the bus. The 7-bit address lets the master address a maximum of 128 slaves on the bus, although the address 0000 000 is reserved for general call and all addresses of the format 1111 xxx are reserved. That means 119 (128 – 1 – 8) devices can share an I2C bus. In the I2C bus the MSB of the address is transmitted first.

The eighth bit in the packet is the READ/WRITE control bit. If this bit is set, the master will read the next frame (Data) from the slave, otherwise, the mas-

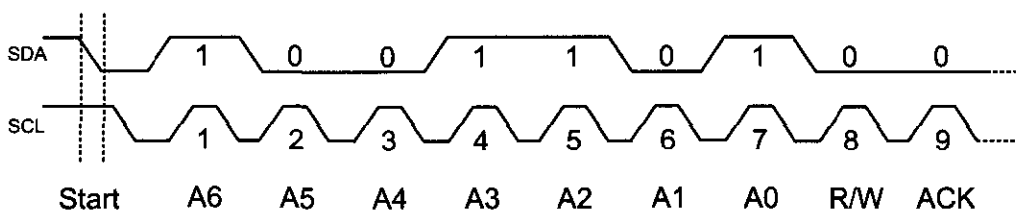
ter will write the next frame (Data) on the bus to the slave. When a slave detects its address on the bus, it knows that it is being addressed and it should acknowledge in the ninth SCL (ACK) cycle by changing SDA to zero. If the addressed slave is not ready or for any reason does not want to service the master, it should leave the SDA line high in the ninth clock cycle. This is considered to be NACK. In case of NACK, the master can transmit a STOP condition to terminate the transmission, or a REPEATED START condition to initiate a new transmission.

Example 18-3 shows how a master says that it wants to write to a slave.

Example 18-3

Show how a master says that it wants to write to a slave with address 1001101.
Solution:

The following actions are performed by the master:
(1) The master puts a high-to-low pulse on SDA, while SCL is high to generate a start bit condition to start the transmission.
(2) The master transmits 10011010 into the bus. The first seven bits (1001101) indicates the slave address, and the eighth bit (0) indicates a Write operation and says that the master will write the next byte (data) into the slave.



An address packet consisting of a slave address and a READ is called SLA+R, while an address packet consisting of a slave address and a WRITE is called SLA+W.

As we mentioned before, address 0000 000 is reserved for general call. This means that when a master transmits address 0000 000, all slaves respond by changing the SDA line to zero and wait to receive the data byte. This is useful when a master wants to transmit the same data byte to all slaves in the system. Notice that the general call address cannot be used to read data from slaves because no more than one slave is able to write to the bus at a given time.

Data packet format

Like other packets, data packets are 9 bits long too. The first 8 bits are a byte of data to be transmitted, and the 9th bit is ACK. If the receiver has received the last byte of data and there is no more data to be received, or the receiver cannot receive or process more data, it will signal a NACK by leaving the SDA line high. In data packets, like address packets, MSB is transmitted first.

Combining address and data packets into a transmission

In I2C, normally, a transmission is started by a START condition, followed by an address packet (SLA + R/W), one or more data packets, and finished by a

STOP condition. Figure 18-7 shows a typical data transmission. Try to understand each element in the figure (see Example 18-4).

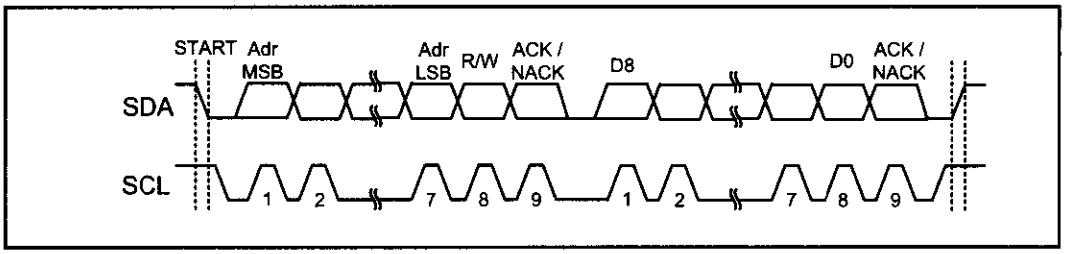


Figure 18-7. Typical Data Transmission

Example 18-4

Show how a master writes the value 11110000 to a slave with address 1001101.
Solution:

The following actions are performed by the master:

(1) The master puts a high-to-low pulse on SDA while, SCL is high to generate a START condition to start the transmission.

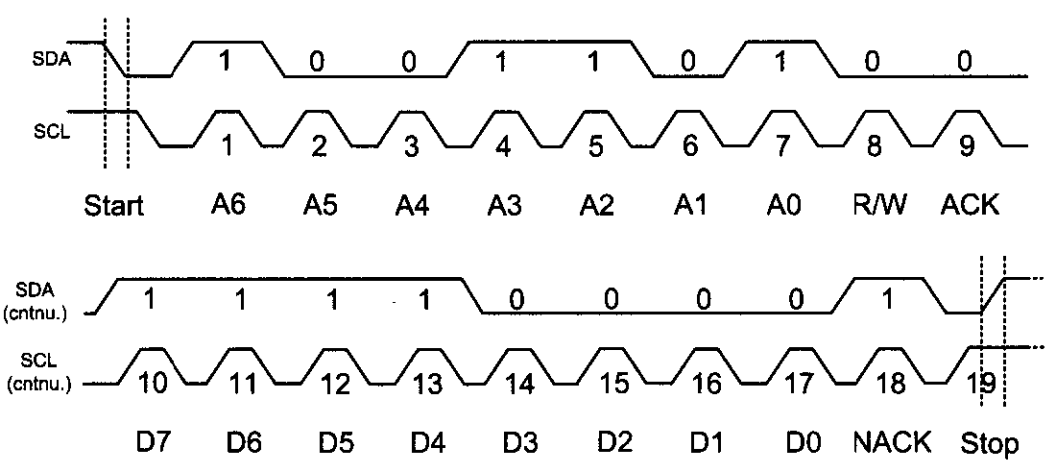
(2) The master transmits 10011010 into the bus. The first seven bits (1001101) indicate the slave address, and the eighth bit (0) indicates the Write operation stating that the master will write the next byte (data) into the slave.

(3) The slave pulls the SDA line low to signal an ACK to say that it is ready to receive the data byte.

(4) After receiving the ACK, the master will transmit the data byte (1111000) on the SDA line (MSB first).

(5) When the slave device receives the data it leaves the SDA line high to signal NACK. This informs the master that the slave received the last data byte and does not need any more data.

(6) After receiving the NACK, the master will know that no more data should be transmitted. The master changes the SDA line when the SCL line is high to transmit a STOP condition and then releases the bus.



Clock stretching

One of the features of the I²C protocol is clock stretching. It is a kind of flow control. If an addressed slave device is not ready to process more data it will stretch the clock by holding the clock line (SCL) low after receiving (or sending) a bit of data. Thus the master will not be able to raise the clock line (because devices are wire-ANDed) and will wait until the slave releases the SCL line to show it is ready to transfer the next bit. See Figure 18-8.

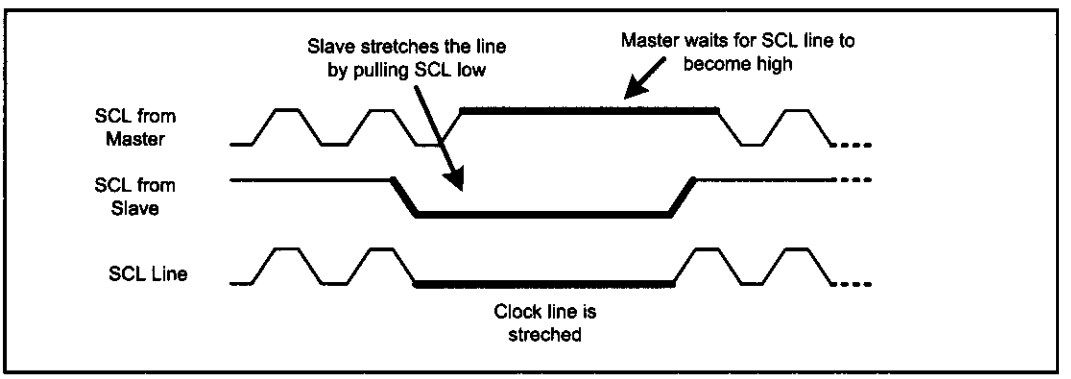


Figure 18-8. Clock Stretching

Arbitration

I2C protocol supports a multimaster bus system. This doesn't mean that more than one master can use the bus at the same time. Rather, each master waits for the current transmission to finish and then starts to use the bus. But it is possible that two or more masters initiate a transmission at about the same time. In this case the arbitration happens.

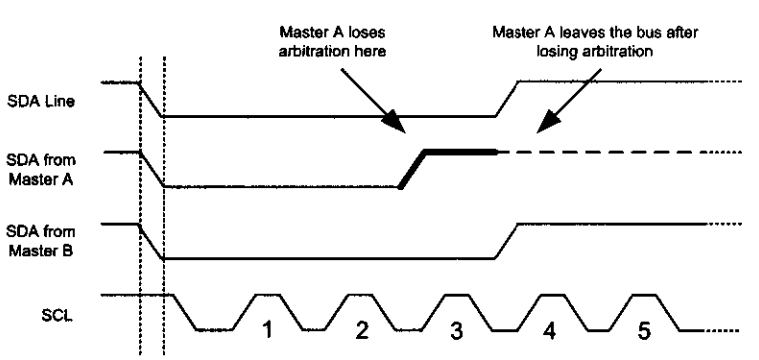
Each transmitter has to check the level of the bus and compare it with the level it expects; if it doesn't match, that transmitter has lost the arbitration, and will switch to slave mode. In the case of arbitration, the winning master will continue its job. Notice that neither the bus is corrupted nor the data is lost. See Example 18-5.

Example 18-5

Two masters, A and B, start at about the same time. What happens if master A wants to write to slave 0010 000 and master B wants to write to slave 0001 111?

Solution:

Master A will lose the arbitration in the third clock because the SDA line is different from the output of master A at the third clock. Master A switches to slave mode and leaves the bus after losing the arbitration.



Multibyte burst write

Burst mode writing is an effective means of loading consecutive locations. It is supported in I2C, SPI, and many other serial protocols. In burst mode, we provide the address of the first location, followed by the data for that location. From then on, consecutive bytes are written to consecutive memory locations. In this mode, the I2C device internally increments the address location as long as the STOP condition is not detected. The following steps are used to send (write) multiple bytes of data in burst mode for I2C devices.

1. Generate a START condition.
2. Transmit the slave address followed by zero (for write).
3. Transmit the address of the first location.
4. Transmit the data for the first location and from then on, simply provide consecutive bytes of data to be placed in consecutive memory locations.
5. Generate a STOP condition.

Figure 18-9 shows how to write 0x01, 0x02, and 0x03 to three consecutive locations starting from location 00001111 of slave 1111000.

Start	Slave address	Write	ACK	First location address	ACK	Data byte #1	ACK	Data byte #2	ACK	Data byte #3	ACK	Stop
S	1111000	0	A	00001111	A	00000001	A	00000010	A	00000011	A	P

Figure 18-9. Multibyte Burst Write

Multibyte burst read

Burst mode reading is an effective way of bringing out the contents of consecutive locations. In burst mode, we provide the address of the first location only. From then on, contents are brought out from consecutive memory locations. In this mode, the I2C device internally increments the address location as long as the STOP condition is not detected. The following steps are used to get (read) multiple bytes of data using burst mode for I2C devices.

1. Generate a START condition.
2. Transmit the slave address followed by zero (for address write).
3. Transmit the address of the first location.
4. Generate a START (REPEATED START) condition.
5. Transmit the slave address followed by one (for read).
6. Read the data from the first location and from then on, bring contents out from consecutive memory locations.
7. Generate a STOP condition.

Figure 18-10 shows how to read three consecutive locations starting from location 00001111 of slave number 1111000.

Start	Slave address	Write	ACK	First location address	ACK	Start	Slave address	Read	ACK	Data byte #1	ACK	Data byte #2	ACK	Data byte #3	ACK	Stop
S	1111000	0	A	00001111	A	S	1111000	1	A	xxxxxxxx	A	xxxxxxxx	A	xxxxxxxx	A	P

Figure 18-10. Multibyte Burst Read

Review Questions

1. True or false. I2C protocol is ideal for short distances.
2. How many bits are there in a frame? Which bit is for acknowledge?
3. True or false. START and STOP conditions are generated when the SDA is high.
4. What is the name of the flow control method in the I2C protocol?
5. What is the recommended value for the pull-up resistors in the I2C protocol?
6. True or false. After the arbitration of two masters, both of them must start transmission from the beginning.

SECTION 18.2: TWI (I2C) IN THE AVR

In many applications, including AVR datasheet, I2C is referred to as *Two-wire Serial Interface (TWI)*. From now on, in this book we use TWI to conform with the AVR data sheets. In this section we discuss the TWI module and registers of the AVR. Then we show how to program the AVR to address a slave device and send or receive data using TWI. The TWI module in the AVR is composed of four submodules: bit rate generation unit, bus interface unit, address match unit, and control unit. Figure 18-11 shows the TWI module. All registers drawn with a thick line are accessible through the AVR data bus.

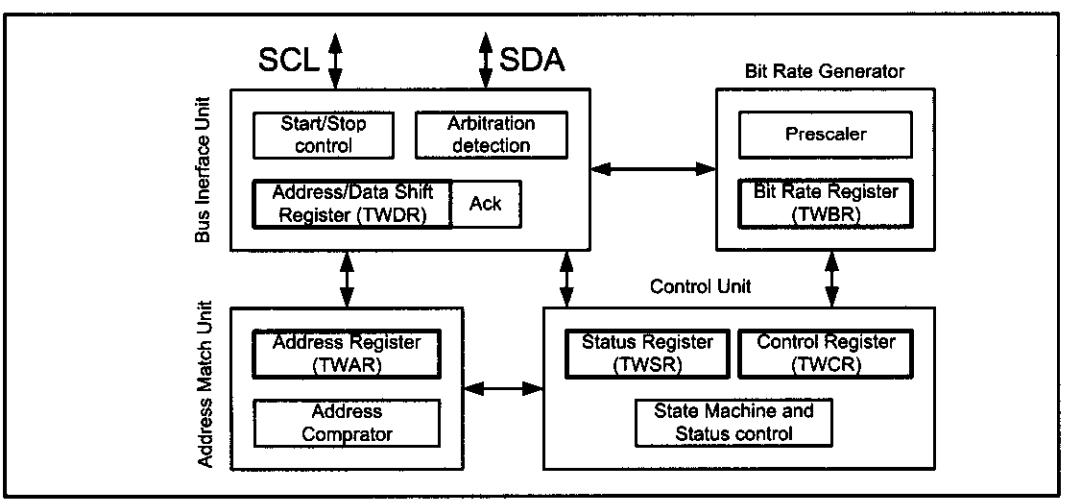


Figure 18-11. TWI (I2C) in AVR

The bit rate generation unit controls the frequency of the system clock (SCL) when operating in a master mode. The bus interface unit detects and generates START, REPEATED START and STOP conditions. It also detects arbitration, controls sending or receiving ACK, and also transfers packets of data or address. The address match unit compares the received address byte with the 7-bit address in TWI address register and informs the control unit upon an address match. The control unit controls the TWI module and generates responses according to settings in the TWI control register. It also sets the contents of the status register according to current state.

In the AVR microcontroller, five major registers are associated with the TWI. They are TWBR (TWI Bit rate Register), TWCR (TWI Control Register), TWSR (TWI Status Register), TWAR (TWI Address Register), and TWDR (TWI

Data Register). Next, we will focus on registers related to TWI and study each bit of them in detail.

TWI Bit Rate Register (TWBR)

The following figure shows the TWBR register and its bits.

TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0

TWBR selects the division factor to control the SCL clock frequency in master mode. The SCL frequency is controlled by settings in the TWBR and the prescaler bits in the TWSR (TWI status register is discussed next). The following equation demonstrates the relation between SCL frequency, TWBR, and TWPS bits in TWI status register:

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2\,(\text{TWBR}) \times 4^{\text{TWPS}}}$$

Notice that the value of TWBR should be 10 or higher if the TWI operates in master mode. Example 18-6 shows how the frequency of SCL is calculated.

Example 18-6

Calculate the SCL frequency if the value of TWPS bits in TWSR is 01 (1 Dec) and the value of TWBR is 00100110 (38 Dec). Assume that CPU clock frequency is 8 MHz.

Solution:

The SCL frequency will be: 8 MHz / ((16 + 2 (38) × 4) = 25 kHz

TWI Status Register (TWSR)

As you see in Figure 18-12, five bits of TWSR are dedicated to show the status of the TWI logic and bus. Notice that if you read TWSR, you will read both the status bits and the prescaler value. To check the status bits, you should mask the two LSB bits (prescaler values) to zero. In this book we do not list all of the status codes and their meanings, but we will cover some of more common ones. To see the complete list of status register codes, you should refer to the data sheet of the chip. Next we will see how to use these bits when we want to program the AVR to use the TWI module.

TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0

Bits 7..3 – TWS: TWI Status
These five bits show the status of the TWI control and bus.

Bits 1..0 – TWPS: TWI Prescaler Bits
These bits control the bit rate prescaler.

Figure 18-12. TWSR: TWI Status Register

TWI Control Register (TWCR)

TWCR controls the operation of the TWI. In Figure 18-13 you see each bit of TWCR and a short description of it. Here we will describe some of these bits in more detail.

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE

Bit 7 – TWINT: TWI Interrupt
This bit is set by hardware when the TWI module has finished its current job. If the TWI and general interrupt are enabled, changing TWINT to one will cause the MCU to jump to the TWI interrupt vector. Clearing this flag starts the operation of the TWI. TWINT must be cleared by software.

Bit 6 – TWEA: TWI Enable Acknowledge
Making this bit HIGH will enable the generation of ACK when needed in slave or receiver mode.

Bit 5 – TWSTA: TWI START condition Bit
Making this bit HIGH will generate a START condition if the bus is free; otherwise, the TWI module waits for the bus to become free and then generates a START condition

Bit 4 – TWSTO: TWI STOP condition bit
In master mode, making this bit HIGH causes the TWI to generate a STOP condition. This bit is cleared by hardware when the STOP condition is transmitted.

Bit 3 – TWWC: TWI Write Collision Flag
This bit is set HIGH when we attempt to access the TWI Data Register when TWINT is low. This flag is cleared by writing to the TWDR register when TWINT is high.

Bit 2 – TWEN: TWI Enable
Making this bit HIGH enables the TWI module.

Bit 0 – TWIE: TWI Interrupt Enable
Making this bit HIGH enables the TWI interrupt if the general interrupt is enabled.

Figure 18-13. TWCR: TWI Control Register

TWI Interrupt (TWINT) flag

When the TWI hardware finishes its job, it sets the TWINT bit to one. If the TWI and general interrupts are enabled, changing TWINT to HIGH will cause the MCU to jump to the TWI interrupt vector. When the TWINT bit is set, the TWI module "stretches" the SCL line to provide enough time for software to do specified jobs. When the software finishes its job, it must clear the TWINT bit to resume the operation of the TWI module. Notice that all accesses to the TWI address, status, and data registers must be complete before clearing this flag. If you try to write to the TWI Data Register when TWINT is low, a collision will happen and the TWI collision flag (TWWC) will be set to HIGH by hardware. Software can monitor (poll) the TWI bit to know when the TWI module finishes its job and is ready for a new command.

TWI Enable Acknowledge (TWEA) bit

Making this bit HIGH will enable the generation of the ACK bit if any of the following conditions are met:

1. The TWI Address Match module detects that the TWI module is addressed by receiving its own slave address from the bus.
2. A general call has been received while the TWGCE bit in the TWAR is set to one (to enable accepting of global calls).
3. A data byte has been received in each of the receiving modes, master receiver or slave receiver mode.

If you clear the TWEA bit to zero, the device will not generate ACK and will be virtually disconnected from the TWI bus.

TWI Start bit and TWI Stop bit (TWSTA and TWSTO)

To generate START or STOP conditions, you have to set the TWSTA or TWSTO bit to one respectively and then clear the TWINT flag to zero by writing a one to it.

TWI Data Register (TWDR)

In Receive mode, the last received byte will be in the TWDR, and in Transmit mode, you should write the next byte into TWDR to be transmitted. As we mentioned before, you can access the TWDR only when the TWIE is set to one otherwise collision happens. This means the Data Register cannot be initialized by the user before the first interrupt occurs.

TWI Address Register (TWAR)

TWAR contains the 7-bit slave address to which the TWI will respond when working as slave. The eighth bit (LSB) of TWAR is TWGCE (TWI General Call Recognition Enable). It controls recognition of general call address (00). If this bit is set to one, receiving of a general call address will cause an interrupt request.

Review Questions

1. True or false. The AVR has an internal TWI module.
2. What are the TWI registers in AVR?
3. How do we generate START or STOP conditions in the AVR?
4. True or false. After reading status register we should mask the 2 MSB bits.
5. Which bit is polled to know if the TWI is ready now?
6. True or false. We can write to TWDR when the TWI module is busy.
7. Which bit controls the generation of ACK?

SECTION 18.3: AVR TWI PROGRAMMING IN ASSEMBLY AND C

In this section we discuss TWI programming in Assembly and C. Here we will focus on the simplest form of TWI programming without checking the status register. In most applications, if you are not dealing with critical systems and there is not more than one master on a single bus, you can use this method. If you want to deal with multimaster or critical designs you must check the value of the status flag. TWI programming with checking the value of the status flag is discussed in a later section.

In I2C protocol, a device can be either master or slave. In this section we will discuss the steps of programming in each mode.

Programming of the AVR TWI in master operating mode

To work in master operating mode, we must be able to initialize the TWI, transmit a START condition, send or receive data, and transmit a STOP condition. Next we will discuss each one in more detail.

Initialization

To initialize the TWI module to operate in master operating mode, we should do the following steps:

1. Set the TWI module clock frequency by setting the values of the TWBR register and the TWPS bits in the TWSR register.
2. Enable the TWI module by setting the TWEN bit in the TWCR register to one.

Transmit START condition

To start data transfer in master operating mode, we must transmit a START condition. This is done by setting the TWEN, TWSTA, and TWINT bits of TWCR to one. Setting the TWEN bit to one enables the TWI module. Setting the TWSTA bit to one tells the TWI to initiate a START condition when the bus is free, and setting the TWINT bit to one clears the interrupt flag to initiate operation of the TWI module to transmit the START condition. Then we should poll the TWINT flag in the TWCR register to see whether the START condition transmitted completely.

Send data

To send a byte of data, after transmitting the START condition, we should do the following steps:

1. Copy the data byte to the TWDR.
2. Set the TWEN and TWINT bits of the TWCR register to one to start sending the byte.
3. Poll the TWINT flag in the TWCR register to see whether the byte transmitted completely.

Notice that right after the START condition, we should transmit SLA + W (Slave Address + Write) or SLA + R (Slave Address + Read). As we mentioned in the first section, right after sending SLA+W we should write to the slave, and right after sending SLA+R we should read from it. To transmit SLA + R, SLA + W, and to write a byte of data to a slave we use a function called I2C_WIRITE.

Receive data

To receive a byte of data, after transmitting of SLA + R, we should do the following steps:

1. Set the TWEN and TWINT bits of the TWCR register to one to start receiving a byte. Notice that if you want to return ACK after receiving data you should also set the TWEA bit of the TWCR register to one.
2. Poll the TWINT flag in the TWCR register to see whether a byte has been received completely.
3. Copy the received byte from the TWDR to another register to save it.

Transmit STOP condition

To stop data transfer, we must transmit a STOP condition. This is done by setting the TWEN, TWSTO, and TWINT bits of the TWCR register to one. Notice that we cannot poll the TWINT flag after transmitting the STOP condition.

Program 18-1 shows how a master writes 11110000 to a slave with address 1101000.

```
;Tested OK-ok
.INCLUDE "M32DEF.INC"

      LDI   R21,HIGH(RAMEND)      ;set up stack
      OUT   SPH,R21
      LDI   R21,LOW(RAMEND)
      OUT   SPL,R21

      CALL  I2C_INIT              ;initialize TWI module
      CALL  I2C_START             ;transmit START condition
      LDI   R27, 0b11010000       ;SLA(1001100) + W (0)
      CALL  I2C_WRITE             ;write R27 to I2C bus
      LDI   R27, 0b11110000       ;data to be transmitted
      CALL  I2C_WRITE             ;write R27 to I2C bus
      CALL I2C_STOP               ;transmit STOP condition

HERE: RJMP  HERE                  ;wait here forever

;*********************************************************
I2C_INIT:
      LDI   R21, 0
      OUT   TWSR,R21              ;set prescaler bits to zero
      LDI   R21, 0x47             ;move 0x47 into R21
      OUT   TWBR,R21        ;set clock freq. to 50k (8 MHz XTAL)
      LDI   R21, (1<<TWEN)        ;move 0x04 into R21
      OUT   TWCR,R21              ;enable the TWI
      RET

;*********************************************************
I2C_START:
      LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
      OUT   TWCR,R21              ;transmit a START condition
```

Program 18-1: Writing a Byte in Master Mode *(continued on next page)*

```
WAIT1:
      IN    R21, TWCR              ;read control register into R21
      SBRS  R21, TWINT             ;skip next line if TWINT is 1
      RJMP  WAIT1                  ;jump to WAIT1 if TWINT is 1
      RET

;*********************************************************
I2C_WRITE:
      OUT   TWDR, R27              ;move the byte into TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21              ;configure TWCR to send TWDR
WAIT3:
      IN    R21, TWCR              ;read control register into R21
      SBRS  R21, TWINT             ;skip next line if TWINT is 1
      RJMP  WAIT3                  ;jump to WAIT3 if TWINT is 1
      RET

;*********************************************************
I2C_STOP:
      LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
      OUT   TWCR, R21              ;transmit STOP condition
      RET
```

Program 18-1: Writing a Byte in Master Mode *(continued from previous page)*

Program 18-2 shows how to read a byte from a slave with address 1001100 and displays the result on Port A.

```
;Tested OK- ok
.INCLUDE "M32DEF.INC"

      LDI   R21,HIGH(RAMEND)       ;set up stack
      OUT   SPH,R21
      LDI   R21,LOW(RAMEND)
      OUT   SPL,R21

      LDI   R21,$FF                ;move $FF to R21
      OUT   DDRA, R21              ;Port A is output

      CALL  I2C_INIT               ;initialize TWI module
      CALL  I2C_START              ;transmit START condition
      LDI   R27, 0b11010001        ;SLA(1001100) + R (1)
      CALL  I2C_WRITE              ;write R27 to I2C bus
      CALL  I2C_READ               ;write R27 to I2C bus
      OUT   PORTA,R27              ;show received data on Port A
      CALL I2C_STOP                ;transmit STOP condition

HERE:
      RJMP  HERE                   ;wait here forever
```

Program 18-2: Reading a Byte in Master Mode *(continued on next page)*

```
;*********************************************************
I2C_INIT:
      LDI   R21, 0
      OUT   TWSR,R21              ;set prescaler bits to zero
      LDI   R21, $47              ;move $47 into R21
      OUT   TWBR,R21              ;SCL freq. is 50k for 8MHz XTAL
      LDI   R21, (1<<TWEN)        ;move 0x04 into R21
      OUT   TWCR,R21              ;enable the TWI
      RET

;*********************************************************
I2C_START:
      LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
      OUT   TWCR,R21              ;transmit a START condition
WAIT1:
      IN    R21, TWCR             ;read control register into R21
      SBRS  R21, TWINT            ;skip next line if TWINT is 1
      RJMP  WAIT1                 ;jump to WAIT1 if TWINT is 1
      RET

;*********************************************************
I2C_READ:
      LDI   R21,(1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21
WAIT2:
      IN    R21, TWCR             ;read control register into R21
      SBRS  R21, TWINT            ;skip next line if TWINT is 1
      RJMP  WAIT2                 ;jump to WAIT2 if TWINT is 0
      IN    R27, TWDR             ;read received data into R21
      RET

;*********************************************************
I2C_WRITE:
      OUT   TWDR, R27             ;move the byte into TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21             ;configure TWCR to send TWDR
WAIT3:
      IN    R21, TWCR             ;read control register into R21
      SBRS  R21, TWINT            ;skip next line if TWINT is 1
      RJMP  WAIT3                 ;jump to WAIT3 if TWINT is 1
      RET

;*********************************************************
I2C_STOP:
      LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
      OUT   TWCR, R21             ;transmit STOP condition
      RET
```

Program 18-2: Reading a Byte in Master Mode *(continued from previous page)*

C programming of the AVR TWI in master operating mode

Program 18-3 shows how a master writes 11110000 to a slave with address 1101000. This program is the C version of Program 18-1.

Program 18-4 shows how a master reads from a slave with address 1101000 and displays the result on Port A. This program is the C version of Program 18-2.

```
#include <avr/io.h>

void i2c_write(unsigned char data)
{
   TWDR = data ;
   TWCR = (1<< TWINT)|(1<<TWEN);
   while ((TWCR & (1 <<TWINT)) == 0);
}

//************************************************************

void i2c_start(void)
{
   TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
   while ((TWCR & (1 << TWINT)) == 0);
}

//************************************************************

void i2c_stop()
{
   TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
}

//************************************************************

void i2c_init(void)
{
   TWSR=0x00;                  //set prescaler bits  to zero
   TWBR=0x47;                  //SCL frequency is 50K for XTAL = 8M
   TWCR=0x04;                  //enable the TWI module
}

//************************************************************

int main (void)
{
   i2c_init();
   i2c_start();                //transmit START condition
   i2c_write(0b11010000);      //transmit SLA + W(0)
   i2c_write(0b11110000);      //transmit data
   i2c_stop();                 //transmit STOP condition
   while(1);                   //stay here forever
   return 0 ;
}
```

Program 18-3: Writing a Byte in Master Mode in C

```
#include <avr/io.h>

void i2c_init(void)
{
  TWSR=0x00;                  //set prescaler bits  to zero
  TWBR=0x47;                  //SCL Frequency is 50K for XTAL=8M
  TWCR=0x04;                  //enable the TWI module
}
//************************************************************
void i2c_start(void)
{
  TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
  while ((TWCR & (1 << TWINT)) == 0);
}
//************************************************************
void i2c_write(unsigned char data)
{
  TWDR = data ;
  TWCR = (1<< TWINT)|(1<<TWEN);
  while ((TWCR & (1 <<TWINT)) == 0);
}
//************************************************************
unsigned char i2c_read(unsigned char isLast)
{
  if (isLast == 0)            //if want to read more than 1 byte
    TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWEA);
  else                        //if want to read only one byte
    TWCR = (1<< TWINT)|(1<<TWEN);
  while ((TWCR & (1 <<TWINT)) == 0);
  return TWDR  ;
}
//************************************************************
void i2c_stop()
{
  TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
}
//************************************************************
int main (void)
{
  unsigned char i = 0 ;
  DDRA = 0xFF;                //Port A is output
  i2c_init();                 //initialize TWI for master mode
  i2c_start();                //transmit START condition
  i2c_write(0b11010001);      //transmit SLA + R(1)
  i=i2c_read(1);              //read only one byte of data
  PORTA= i;                   //show the byte on Port A
  i2c_stop();                 //transmit STOP condition
  while(1);                   //stay here forever
  return 0 ;
}
```

Program 18-4: Reading a Byte in Master Mode in C

Programming of the AVR TWI in slave operating mode

To work in slave operating mode, we must be able to initialize the TWI and we must also be able to send or receive data. In slave mode we cannot transmit START or STOP conditions. A slave device should listen to the bus and wait to be addressed by a master device or general call.

Initialization

To initialize the TWI module to operate in slave operating mode, we should do the following steps:

1. Set the slave address by setting the values for the TWAR registers. As we mentioned before, the upper seven bits of TWAR are the slave address, and the eighth bit is TWGCE. If you set this bit to one, the TWI will respond to the general call address ($00); otherwise, it will ignore the general call address.
2. Enable the TWI module by setting the TWEN bit in the TWCR register to one.
3. Set the TWEN, TWINT, and TWEA bits of TWCR to one to enable the TWI and acknowledge generation.

Notice that we cannot combine steps 2 and 3 into a single step. We have to enable the TWI module before doing the third step.

Listen to the bus

After initializing the TWI module, a slave device should listen to the bus to detect when it is addressed by a master device. When the TWI module detects its own address on the bus, it returns ACK and then sets the TWINT flag in the TWCR register to one. We should poll the TWINT flag to see when the slave is addressed by a master device.

Send data

After being addressed by a master device for read, we should do the following steps to send a byte of data:

1. Copy the data byte to the TWDR.
2. Set the TWEN, TWEA, and TWINT bits of the TWCR register to one to start sending the byte. Notice that if you expect not to receive ACK after receiving data you can leave the TWEA cleared. It will have no effect on generation of ACK by master and will only change the internal state of the TWI module. We recommend that you set the TWEA bit of the TWCR register to one anyway.
3. Poll the TWINT flag in the TWCR register to see when the byte is completely transmitted.

Receive data

After being addressed by a master device, we should do the following steps to receive a byte of data:

1. Set the TWEN and TWINT bits of the TWCR register to one to start receiving a byte. Notice that if you want to return ACK after receiving data you should also set the TWEA bit of the TWCR register to one.
2. Poll the TWINT flag in the TWCR register to see whether a byte has been received completely.
3. Copy the received byte from the TWDR to another register to save it.

Programs 18-5 and 18-6 show how to initialize the TWI module to operate

in slave mode. In Program 18-5 the TWI module listens to the bus and waits to be addressed by a master device. Then it transmits the letter 'G' to the master device.

```
.INCLUDE "M32DEF.INC"

     LDI   R21,HIGH(RAMEND);set up stack
     OUT   SPH,R21
     LDI   R21,LOW(RAMEND)
     OUT   SPL,R21

     CALL  I2C_INIT         ;initialize the TWI module as slave
     CALL  I2C_LISTEN       ;listen to the bus to be addressed
     LDI   R21, 'G'         ;load 'G' into R21
     CALL  I2C_WRITE        ;write the byte to the bus
HERE:
     RJMP  HERE             ;wait here forever

;*********************************************************

I2C_INIT:
     LDI   R21, 0x10        ;load slave address 00010000 into R21
     OUT   TWAR,R21         ;load TWI Address Register
     LDI   R21, (1<<TWEN)   ;move 0x04 into R21
     OUT   TWCR,R21         ;enable the TWI
     LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
     OUT   TWCR,R21         ;enable TWI and ACK(can't be ignored)
     RET

;*********************************************************

I2C_LISTEN:
W1:
     IN    R21, TWCR        ;read control register into R21
     SBRS  R21, TWINT       ;skip next intruction if TWINT is 1
     RJMP  W1               ;jump to W1 if TWINT is 0
     RET

;*********************************************************

I2C_WRITE:

     OUT   TWDR, R21        ;move R21 to TWDR
     LDI   R21, (1<<TWINT)|(1<<TWEN)
     OUT   TWCR, R21        ;configure TWCR to send TWDR
W2:
     IN    R21, TWCR        ;read control register into R21
     SBRS  R21, TWINT       ;skip next line if TWINT is 1
     RJMP  W2               ;jump to W2 if TWINT is 0
     RET
```

Program 18-5: Writing a Byte in Slave Mode

In Program 18-6 the TWI module listens to the bus and waits to be addressed by a master device. Then it reads a byte of data from the master device and displays it on Port A.

```
.INCLUDE "M32DEF.INC"

      LDI   R21,HIGH(RAMEND);set up stack
      OUT   SPH,R21
      LDI   R21,LOW(RAMEND)
      OUT   SPL,R21

      LDI   R21, 0xFF        ;move 0xFF into R21
      OUT   DDRA,R21         ;set Port A as output

      CALL  I2C_INIT         ;initialize the TWI module as slave
      CALL  I2C_LISTEN       ;listen to the bus to be addressed
      CALL  I2C_READ         ;read a byte and copy it to R27
      OUT   PORTA,R27        ;copy R27 to PORTA
HERE:
      RJMP  HERE             ;wait here forever
;*********************************************************

I2C_INIT:
      LDI   R21, 0x10        ;load 00010000 into R21
      OUT   TWAR,R21         ;set address register
      LDI   R21, (1<<TWEN)   ;move 0x04 into R21
      OUT   TWCR,R21         ;enable the TWI
      LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
      OUT   TWCR,R21         ;enable TWI and ACK(can't be ignored)
      RET
;*********************************************************

I2C_LISTEN:
W1:
      IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;skip next intruction if TWINT is 1
      RJMP  W1               ;jump to W1 if TWINT is 0
      RET
;*********************************************************

I2C_READ:
      LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
      OUT   TWCR, R21        ;configure TWCR to receive TWDR
W2:   IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;skip next line if TWINT is 1
      RJMP  W2               ;jump to W2 if TWINT is 0
      IN    R27,TWDR         ;move received data into R27
      RET
```

Program 18-6: Reading a Byte in Slave Mode

C programming of the AVR TWI in slave operating mode

Program 18-7 is the C version of Program 18-5. Program 18-7 shows how to initialize the TWI module to operate in slave mode. In Program 18-7 the TWI module listens to the bus and waits to be addressed by a master device. Then it transmits the letter 'G' to the master device.

```
#include <avr/io.h>                          //standard AVR header

void i2c_initSlave(unsigned char slaveAddress)
{
   TWCR = 0x04;                              //enable TWI module
   TWAR = slaveAddress;                      //set the slave address
   TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWEA);//init. TWI module
}

//***********************************************************

void i2c_send(unsigned char data)
{
   TWDR = data;                              //copy data to TWDR
   TWCR = (1<< TWINT)|(1<<TWEN);             //start transmission
   while ((TWCR & (1 <<TWINT))==0);          //wait to complete
}

//***********************************************************

void i2c_listen()
{
   while ((TWCR & (1 <<TWINT))==0);          //wait to be addressed
}

//***********************************************************

int main (void)
{
   i2c_initSlave(0x10);                      //init. TWI module as
                                             //slave with address
                                             //0b0001000 and do not
                                             //accept general call
   i2c_listen();                             //listen to be addressed
   i2c_send('G');                            //transmit letter 'G'
   while(1);                                 //stay here forever
   return 0;
}
```

Program 18-7: Writing a Byte in Slave Mode in C

Program 18-8 is the C version of Program 18-6. In Program 18-8 the TWI module listens to the bus and waits to be addressed by a master device. Then it reads a byte of data from the master device and displays it on Port A.

```
#include <avr/io.h>                          //standard AVR header

void i2c_initSlave(unsigned char slaveAddress)
{
   TWCR = 0x04;                              //enable TWI module
   TWAR = slaveAddress;                      //set the slave address
   TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWEA);//init. TWI module
}

//*************************************************************

unsigned char i2c_receive(unsigned char isLast)
{
   if (isLast == 0)          //if want to read more than 1 byte
     TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWEA);
   else                      //if want to read only one byte
     TWCR = (1<< TWINT)|(1<<TWEN);

   while ((TWCR & (1 <<TWINT))==0);    //wait to complete
   return (TWDR);
}

//*************************************************************

void i2c_listen()
{
   while ((TWCR & (1 <<TWINT))==0);    //wait to be addressed
}

//*************************************************************

int main (void)
{
   DDRA = 0xFF;
   i2c_initSlave(0x10);                      //init. TWI module as
                                             //slave with address
                                             //0b0001000 and do not
                                             //accept general call
   i2c_listen();                             //listen to be addressed
   PORTA = i2c_receive(1);                   //
   while(1);                                 //stay here forever
   return 0;
}
```

Program 18-8: Reading a Byte in Slave Mode in C

Review Questions

1. True or false. We can ignore the status flag in multimaster systems.
2. Which of the following is not needed to initialize the TWI module to operate in master operating mode? (More than one choice can be true.)
   (a) Enable the TWI module.
   (b) Set the value of the prescaler bits.
   (c) Set the value of the TWBR register.
   (d) Set the value of the TWAR register.
3. Which of the following is not needed to initialize the TWI module to operate in slave operating mode? (More than one choice can be true.)
   (a) Enable the TWI module.
   (b) Set the value of the prescaler bits.
   (c) Set the value of the TWBR register.
   (d) Set the value of the TWAR register.
4. Which of the following instructions is used to transmit a STOP condition in master operating mode?
   (a)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
   OUT   TWCR,R21
   ```
   (b)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWEN)
   OUT   TWCR, R21
   ```
   (c)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
   OUT   TWCR, R21
   ```
   (d)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
   OUT   TWCR,R21
   ```
5. Which of the following instructions is used to transmit a STOP condition in master operating mode?
   (a)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
   OUT   TWCR,R21
   ```
   (b)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWEN)
   OUT   TWCR, R21
   ```
   (c)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
   OUT   TWCR, R21
   ```
   (d)
   ```
   LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
   OUT   TWCR,R21
   ```

SECTION 18.4: DS1307 RTC INTERFACING AND PROGRAMMING

The real-time clock (RTC) is a widely used device that provides accurate time and date information for many applications. Many systems such as the x86 PC come with such a chip on the motherboard. The RTC chip in the x86 PC provides the time components of hour, minute, and second, in addition to the date/calendar components of year, month, and day. Many RTC chips use an internal battery, which keeps the time and date even when the power is off. Although some microcontrollers, such as the DS5000T and some of AVRs, come with the RTC already embedded into the chip, we have to interface the vast majority of them to an external RTC chip. One of the most widely used RTC chips is the DS12887 from Dallas Semiconductor/Maxim Corp. This chip is found in the vast majority of x86 PCs. The original IBM PC/AT used the MC14618B RTC from Motorola (now Freescale). The DS12887 is the replacement for that chip. It uses an internal lithium battery to keep operating for over 10 years in the absence of external power. The DS12887 is a parallel RTC with 8 pins for the data bus. The DS1307 is a serial RTC with an I2C bus. In this section, we interface and program the DS1307 RTC. According to the DS1307 data sheet from Maxim, "The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with AM/PM indicator. The DS1307 has a built-in power-sense circuit that detects power failures and automatically switches to the battery supply." The DS1307 does not support the Daylight Savings Time option. Next, we describe the pins of the DS1307. See Figure 18-14.

X1–X2

These are input pins that allow the DS1307 connection to an external crystal oscillator to provide the clock source to the chip. We must use the standard 32.768 kHz quartz crystal. The accuracy of the clock depends on the quality of this crystal oscillator. Heat can cause a drift on the oscillator. To avoid this, we can use the DS32KHZ chip, which automatically adjusts for temperature variations. Notice that when using the DS32KHZ or similar clock generators, we only need to connect X1 because the X2 loopback is not required.

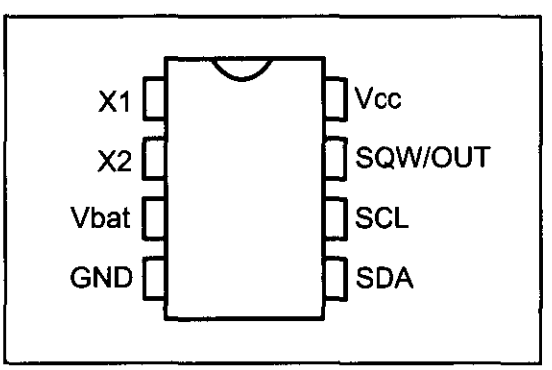


Figure 18-14. DS1307 Pin Out

V_{bat}

Pin 3 can be connected to an external +3 V lithium battery, thereby providing the power source to the chip when the external supply voltage is not available. We must connect this pin to ground if it is not used. A 48mAhr lithium battery can

provide the power needed for more than 10 years to back up the chip.

GND

Pin 4 is the ground.

SDA (Serial Data)

Pin 5 is the SDA pin and must be connected to the SDA line of the I2C bus.

SCL (Serial Clock)

Pin 6 is the SCL pin and must be connected to the SCL line of the I2C bus.

SWQ/OUT

Pin 7 is an output pin providing 1 kHz, 4 kHz, 8 kHz, or 32 kHz frequency if enabled. This pin needs an external pull-up resistor to generate the frequency because it is open drain. If you do not want to use this pin you can omit the external pull-up resistor. We will see shortly how to control this pin.

V_{CC}

Pin 8 is used as the primary voltage supply to the chip. This primary voltage source is generally set to +5 V. When V_{cc} falls below the Vbat level, the DS1307 switches to Vbat and the external lithium battery provides power to the RTC. According to the DS1307 data sheet, "upon power-up, the device switches from Vbat to Vcc1 when Vcc1 is greater than V_{bat}+0.2 Volts." Also notice that the device is accessible only when V_{cc} is more than 1.25 × V_{bat}. Because we can connect the standard 3 V lithium battery to the V_{bat} pin, the Vcc voltage level must remain above 3.2 V in order for the Vcc to remain as the primary voltage source to the chip, and it must be more than 3.75 V if you want to access the chip.

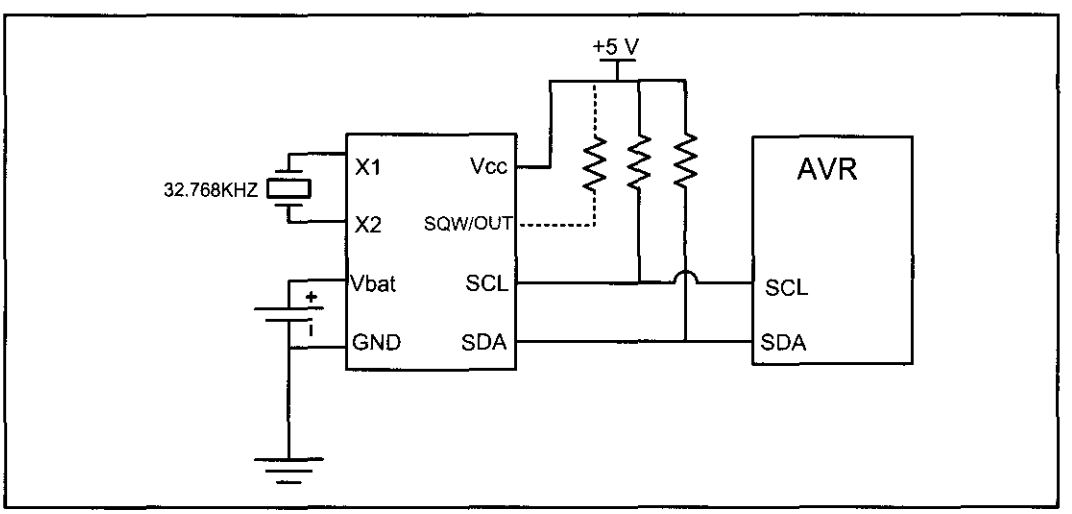


Figure 18-15. DS1307 Power Connection Options (Maxim/Dallas Semiconductor)

Address map of the DS1307

The DS1307 has a total of 64 bytes of RAM space with addresses 00–3FH. The first seven locations, 00–06, are set aside for RTC values of time and date. The next byte is used for the control register. It is located at address 07 in hex. That leaves 56 bytes, from addresses 07H to 3FH, available for general-purpose data storage. That means the entire 64 bytes of RAM are accessible directly for read or

write. Figure 18-16 shows the address map of the DS1307. Next, we study the control register, and time and date access in DS1307.

ADDRESS	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00–59
01H	0	10 Minutes			Minutes				Minutes	00–59
02H	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM
		24	PM/AM							00–23
03H	0	0	0	0	0	DAY			Day	01–07
04H	0	0	10 Date		Date				Date	01–31
05H	0	0	0	10 Month	Month				Month	01–12
06H	10 Year				Year				Year	00–99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08H-3FH									RAM 56 x 8	00H–FFH

Figure 18-16. Simplified Block Diagram of DS1307 (Maxim/Dallas Semiconductor)

The DS1307 control register

As shown in Figure 18-16, the control register has an address of 07H. In the DS1307 control register, the bits control the function of the SQW/OUT pin. In Figure 18-17 you see the function of each bit.

OUT	0	0	SQWE	0	0	RS1	RS0

OUT (output control) If the square wave output is disabled, setting the OUT bit to one will make the SQW/OUT pin low, and clearing the OUT bit to zero will make the SQW/OUT pin high.

SQWE (square wave enable) If this bit is set HIGH, the oscillator output is enabled; otherwise, it is disabled.

RS1-RS0 (rate select) These bits select the output frequency of the oscillator output according to the following table.

RS1	RS0	Output Frequency
0	0	1 Hz
0	1	4.096 kHz
1	0	8.192 kHz
1	1	32.768 kHz

Figure 18-17. DS1307 Control Register (Write location address is 8FH)

CH bit in address 00

One of the most important bits in the Seconds address location in the DS1307 is the CH (Clock Halt) bit. It is the seventh bit of address location 00. Setting the CH bit to one disables the oscillator, while setting CH to zero enables the oscillator. The CH bit is undefined upon reset. In order to enable the oscillator, we must clear the CH during initial configuration.

Time and date address locations and modes

The byte addresses 0–6 are set aside for the time and date, as shown in Figure 18-16. The DS1307 provides data in BCD format only. Notice the data range for the hour mode. We can select 12-hour or 24-hour mode with bit 6 of hour location 02. When D6 = 1, the 12-hour mode is selected, and D6 = 0 provides us the 24-hour mode. In the 12-hour mode, we decide the AM and PM with the bit 5. If D5 = 0, the AM is selected and D5 = 1 is for the PM. See Example 18-7.

Example 18-7

Find the values for address location $02 to set the hour to: (a) 21, (b) 11 AM, (c) 12 PM.

Solution:

(a) For 24-hour mode, we have D6 = 0. Therefore, we place 0010 0001 at location $02, which is 21 in BCD.
(b) For 12-hour mode, we have D6 = 1. Also, we have D5 = 0 for AM. Therefore, we place 0101 0001 at location $02, which is 51 in BCD.
(c) For 12-hour mode, we have D6 = 1. Also, we have D5 = 1 for PM. Therefore, we place 0111 0010 at location $02, which is 72 in BCD.

Register pointer

In DS1307 there is a register pointer that specifies the byte that will be accessed in the next read or write command. After each read or write operation, the content of the register pointer is automatically incremented. It is useful in multibyte read or write.

Writing to DS1307

To set the value of the register pointer and write one or more bytes of data to DS1307, you can use the following steps:

1. To access the DS1307 for a write operation, after sending a START condition, you should transmit the address of DS1307 (1001101) followed by 0 to indicate a write operation.
2. The first byte of data in the write operation will set the register pointer. For example, if you want to access the control register you should send 0x07.
3. If you want only to set the register pointer you should skip this step. If you want to write one or more bytes of data, you should transmit them one byte at a time. Remember that the register pointer is automatically incremented and you can simply transmit bytes of data to consecutive locations in a multibyte burst write.
4. Transmit a STOP bit condition.

Reading from DS1307

Notice that before reading a byte you should load the address of the byte to the register pointer by doing a write operation as mentioned before.

To read one or more bytes of data from the DS1307 you should do the fol-

lowing steps:

1. To access the DS1307 for a read operation, after sending a START condition, you should transmit the address of DS1307 (1001101) followed by 1 to indicate a read operation.
2. Now you can read one or more bytes of data. Remember that the register pointer indicates which address will be read. Also notice that the register pointer is automatically incremented and you can simply receive consecutive bytes of data in a multibyte burst read.
3. Transmit a STOP bit condition.

Setting the time in Assembly

Program 18-9 initializes the clock at 16:58:55 using the 24-hour clock mode. It uses the single-byte operation for writing into the control register of the DS1307 and multibyte burst mode for writing seconds, minutes, and hours. Notice that in this program we assume that there is only one master on the bus and we do not deal with checking the status register.

```
.INCLUDE "M32DEF.INC"

     LDI   R21,HIGH(RAMEND)      ;set up stack
     OUT   SPH,R21
     LDI   R21,LOW(RAMEND)
     OUT   SPL,R21

     CALL I2C_INIT               ;initialize the I2C module

     CALL  I2C_START             ;transmit a START condition
     LDI   R21, 0b11010000       ;SLA (1001101) + W(0)
     CALL I2C_SEND               ;transmit R21 to I2C bus
     LDI   R21, 0x07             ;set register pointer to 07
     CALL I2C_SEND               ;to access the control register
     LDI   R21, 0x00             ;set control register = 0
     CALL I2C_SEND               ;transmit R21 to I2C bus
     CALL I2C_STOP               ;transmit a STOP condition

     CALL DELAY

     CALL  I2C_START             ;transmit a START condition
     LDI   R21, 0b11010000       ;SLA (1001101) + W(0)
     CALL I2C_SEND               ;transmit R21 to I2C bus
     LDI   R21, 0x00             ;set register pointer to 0
     CALL I2C_SEND               ;transmit R21 to I2C bus
     LDI   R21, 0x55             ;set seconds to 0x55 = 55 BCD
     CALL I2C_SEND               ;transmit R21 to I2C bus
     LDI   R21, 0x58             ;set minutes to 0x58 = 58 BCD
     CALL I2C_SEND               ;transmit R21 to I2C bus
     LDI   R21, 0b00010110       ;hour = 16 in 24 hours mode
     CALL I2C_SEND               ;transmit R21 to I2C bus
```

Program 18-9: Setting the Time in Assembly

```
      CALL I2C_STOP          ;transmit a STOP condition
HERE:
      RJMP HERE              ;wait here forever

;*****************************************************
I2C_INIT:
      LDI   R21, 0
      OUT   TWSR,R21         ;set prescaler bits to zero
      LDI   R21, 0x47        ;move 0x47 into r21
      OUT   TWBR,R21         ;SCL freq. is 50k for 8 MHz XTAL
      LDI   R21, (1<<TWEN)   ;move 0x04 into r21
      OUT   TWCR,R21         ;enable the TWI
      RET

;*****************************************************
I2C_START:
      LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
      OUT   TWCR,R21         ;transmit a START condition
W1:   IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;mask the interrupt flag
      RJMP  W1               ;jump to W1 if TWINT is 1
      RET

;*****************************************************
I2C_SEND:
      OUT   TWDR, R21        ;move SLA+W into TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21        ;configure TWCR to send TWDR
W2:   IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;mask the interrupt flag
      RJMP  W2               ;jump to W2 if TWINT is 1
      RET

;*****************************************************
I2C_STOP:
      LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
      OUT   TWCR, R21        ;transmit STOP condition
W3:   IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWSTO       ;mask the interrupt flag
      RJMP  W3               ;jump to W3 if TWINT is 1
      RET

;*****************************************************
DELAY:
      LDI   R22, 0xFF
A1:   DEC   R22              ;transmit STOP condition
      NOP
      BRNE  A1
      RET
```

Program 18-9: Setting the Time in Assembly *(continued from previous page)*

Setting the date in Assembly

Program 18-10 shows how to set the date to October 19th, 2009. It uses the single-byte operation for writing into the control register of the DS1307 and multibyte burst mode for writing day, month, and year. As you can see in the program, to access the location of the date, you should write 0x04 into the register pointer and then you can use multibyte burst write to write the values of month and year in the consecutive locations. Also, notice that in this code we assume that there is only one master on the bus and we do not deal with checking the status register.

```
.INCLUDE "M32DEF.INC"

     LDI   R21,HIGH(RAMEND)       ;set up stack
     OUT   SPH,R21
     LDI   R21,LOW(RAMEND)
     OUT   SPL,R21

     CALL I2C_INIT                ;initialize the I2C module

     CALL  I2C_START              ;transmit a START condition
     LDI   R21, 0b11010000        ;SLA (1001101) + W(0)
     CALL I2C_SEND                ;transmit R21 to I2C bus
     LDI   R21, 0x07              ;set register pointer to 07
     CALL I2C_SEND                ;to access the control register
     LDI   R21, 0x00              ;set control register = 0
     CALL I2C_SEND                ;transmit R21 to I2C bus
     CALL I2C_STOP                ;transmit a STOP condition

     CALL DELAY


     CALL  I2C_START              ;transmit a START condition
     LDI   R21, 0b11010000        ;SLA (1001101) + W(0)
     CALL I2C_SEND                ;transmit R21 to I2C bus
     LDI   R21, 0x04              ;set register pointer to 4
     CALL I2C_SEND                ;transmit R21 to I2C bus
     LDI   R21, 0x19              ;set day to 0x19 = 19 BCD
     CALL I2C_SEND                ;transmit R21 to I2C bus
     LDI   R21, 0x10              ;set month to 0x10 = 10 BCD
     CALL I2C_SEND                ;transmit R21 to I2C bus
     LDI   R21, 0x09              ;set year to 0x09 = 09 BCD
     CALL I2C_SEND                ;transmit R21 to I2C bus
     CALL I2C_STOP                ;transmit a STOP condition

HERE: RJMP  HERE                  ;wait here forever
```

Program 18-10: Setting the Date in Assembly

```
;*****************************************************
I2C_INIT:
      LDI   R21, 0
      OUT   TWSR,R21          ;set prescaler bits to zero
      LDI   R21, 0x47         ;move 0x47 into R21
      OUT   TWBR,R21          ;SCL freq. is 50k for 8 MHz XTAL
      LDI   R21, (1<<TWEN)    ;move 0x04 into R21
      OUT   TWCR,R21          ;enable the TWI
      RET

;*****************************************************
I2C_START:
      LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
      OUT   TWCR,R21          ;transmit a START condition
W1:   IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWINT        ;mask the interrupt flag
      RJMP  W1                ;jump to W1 if TWINT is 1
      RET
;*****************************************************
I2C_SEND:
      OUT   TWDR, R21         ;move SLA+W into TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21         ;configure TWCR to send TWDR
W2:   IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWINT        ;mask the interrupt flag
      RJMP  W2                ;jump to W2 if TWINT is 1
      RET
;*****************************************************
I2C_STOP:
      LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
      OUT   TWCR, R21         ;transmit STOP condition
W3:   IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWSTO        ;mask the interrupt flag
      RJMP  W3                ;jump to W3 if TWINT is 1
      RET
;*****************************************************
DELAY:
      LDI   R22, 0xFF
A1:   DEC   R22               ;transmit STOP condition
      NOP
      BRNE  A1
      RET
```

Program 18-10: Setting the Date in Assembly *(continued from previous page)*

Setting the time in C

Programs 18-11 and 18-12 are the C versions of the last two programs. Notice that you have to make the optimization level o0 (optimization 0); other-

```
#include <avr/io.h>                    //standard AVR header
void i2c_stop()
{     TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
}
//*************************************************************
void i2c_write(unsigned char data)
{
      TWDR = data ;
      TWCR = (1<< TWINT)|(1<<TWEN);
      while (!(TWCR & (1 <<TWINT)));
}
//*************************************************************
void i2c_start(void)
{
     TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
     while (!(TWCR & (1 << TWINT)));
}
//*************************************************************
void i2c_init(void)
{
      TWSR=0x00;                       //set prescaler bits  to zero
      TWBR=0x47;                       //SCL freq. is 50k for XTAL=8M
      TWCR=0x04;                       //enable TWI module
}
//*************************************************************
int main (void)
{     i2c_init();                      //initialize I2C module
      i2c_start();                     //transmit START condition
      i2c_write(0b11010000);           //address DS1307 for write
      i2c_write(0x07);                 //set register pointer to 7
      i2c_write(0x00);                 //set value of location 7 to 0
      i2c_stop();                      //transmit STOP condition

      for ( int k = 0 ; k<100 ; k++); //wait for a short time

      i2c_start();                     //transmit START condition
      i2c_write(0b11010000);           //address DS1307 for write
      i2c_write(0);                    //set register pointer to 7
      i2c_write(0x55);                 //set seconds to 0x55 = 55 BCD
      i2c_write(0x58);                 //set minutes to 0x58 = 58 BCD
      i2c_write(0b00010110);           //set hour=16 in 24 hours mode
      i2c_stop();                      //transmit STOP condition

      while(1);                        //stop here
      return 0;
}
```

Program 18-11: Setting the Time in C

wise, the compiler would omit the line "`for (int k = 0 ; k<100 ; k++)`" and the program would not work correctly.

```
#include <avr/io.h>                    //standard AVR header

void i2c_stop()
{     TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
}
//*************************************************************
void i2c_write(unsigned char data)
{     TWDR = data ;
      TWCR = (1<< TWINT)|(1<<TWEN);
      while (!(TWCR & (1 <<TWINT)));
}
//*************************************************************
void i2c_start(void)
{
     TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
     while (!(TWCR & (1 << TWINT)));
}
//*************************************************************
void i2c_init(void)
{
      TWSR=0x00;                       //set prescaler bits  to zero
      TWBR=0x47;                       //SCL freq. is 50K for XTAL=8M
      TWCR=0x04;                       //enable TWI module
}
//*************************************************************
int main (void)
{
      i2c_init();                      //initialize I2C module
      i2c_start();                     //transmit START condition
      i2c_write(0b11010000);           //address DS1307 for write
      i2c_write(0x07);                 //set register pointer to 7
      i2c_write(0x00);                 //set value of location 7 to 0
      i2c_stop();                      //transmit STOP condition

      for ( int k = 0 ; k<100 ; k++) ; //wait for a short time

      i2c_start();                     //transmit START condition
      i2c_write(0b11010000);           //address DS1307 for write
      i2c_write(0x04);                 //set register pointer to 4
      i2c_write(0x19);                 //set day to 0x19 = 19 BCD
      i2c_write(0x10);                 //set month to 0x10 = 10 BCD
      i2c_write(0x09);                 //set year to 0x09 = 09 BCD
      i2c_stop();                      //transmit STOP condition

      while(1);                        //stop here
      return 0;
}
```

Program 18-12: Setting the Date in C

Setting, reading, and displaying time and date in C

Program 18-13 is the complete C code for setting, reading, and displaying the time and date. The times and dates are sent to the IBM PC screen via the serial port after they are converted from packed BCD to ASCII.

```
#include <avr/io.h>                    //standard AVR header

void usart_init(void)
{
      //initialize USART transmitter for 8-bit data no parity
      //and one stop bit
      UCSRB = (1<<TXEN) ;

      UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
      UBRRL = 0x33 ;
}
//************************************************************

void usart_send( unsigned char data )
{
      while (! (UCSRA & (1<<UDRE))); //wait until udr is empty
      UDR = data ;
}
//************************************************************

void usart_send_packedBCD( unsigned char data )
{
      usart_send('0'+ (data>>4));
      usart_send('0'+ (data & 0x0F));
}

//************************************************************

void i2c_init(void)
{
      TWSR=0x00;              //set prescaler bits  to zero
      TWBR=0x47;              //SCL frequency is 50K for XTAL = 8M
      TWCR=0x04;              //enable TWI module
}

//************************************************************
void i2c_start(void)
{
     TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
     while (!(TWCR & (1<<TWINT)));
}
//************************************************************
```

Program 18-13: A Complete DS1307 Code Example in C

```
void i2c_write(unsigned char data)
{
      TWDR = data ;
      TWCR = (1<< TWINT)|(1<<TWEN);
      while (!(TWCR & (1 <<TWINT)));
}

//************************************************************

unsigned char i2c_read(unsigned char ackVal)
{
      TWCR = (1<< TWINT)|(1<<TWEN)|(ackVal<<TWEA);
      while (!(TWCR & (1 <<TWINT)));
      return TWDR  ;
}

//************************************************************

void i2c_stop()
{
      TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
      for ( int k = 0 ; k<100 ; k++) ; //wait for a short time
}

//************************************************************

void rtc_init(void)
{
      i2c_init();           //initialize I2C module
      i2c_start();          //transmit START condition
      i2c_write(0xD0);      //address DS1307 for write
      i2c_write(0x07);      //set register pointer to 7
      i2c_write(0x00);      //set value of location 7 to 0
      i2c_stop();           //transmit STOP condition
}

//************************************************************

void rtc_setTime(unsigned char h,unsigned char m,unsigned char
s)
{
      i2c_start();          //transmit START condition
      i2c_write(0xD0);      //address DS1307 for write
      i2c_write(0);         //set register pointer to 0
      i2c_write(s);         //set seconds
      i2c_write(m);         //set minutes
      i2c_write(h);         //set hour
      i2c_stop();           //transmit STOP condition
}
```

Program 18-13: A Complete DS1307 Code Example in C *(continued from previous page)*

```
//*************************************************************
void rtc_setDate(unsigned char y,unsigned char m,unsigned char
d)
{
      i2c_start();          //transmit START condition
      i2c_write(0xD0);      //address DS1307 for write
      i2c_write(0x04);      //set register pointer to 4
      i2c_write(d);         //set day
      i2c_write(m);         //set month
      i2c_write(y);         //set year
      i2c_stop();           //transmit STOP condition
}

//*************************************************************

void rtc_getTime(unsigned char *h,unsigned char *m,unsigned char
*s)
{
      i2c_start();          //transmit START condition
      i2c_write(0xD0);      //address DS1307 for write
      i2c_write(0);         //set register pointer to 0
      i2c_stop();           //transmit STOP condition

      i2c_start();          //transmit START condition
      i2c_write(0xD1);      //address DS1307 for read
      *s = i2c_read(1);     //read second, return ACK
      *m = i2c_read(1);     //read minute, return ACK
      *h = i2c_read(0);     //read hour, return NACK
      i2c_stop();           //transmit STOP condition

}

//*************************************************************

void rtc_getDate(unsigned char *y,unsigned char *m,unsigned char
*d)
{
      i2c_start();          //transmit START condition
      i2c_write(0xD0);      //address DS1307 for write
      i2c_write(0x04);      //set register pointer to 4
      i2c_stop();           //transmit STOP condition

      i2c_start();          //transmit START condition
      i2c_write(0xD1);      //address DS1307 for read
      *d = i2c_read(1);     //read day, return ACK
      *m = i2c_read(1);     //read month, return ACK
      *y = i2c_read(0);     //read year, return NACK
      i2c_stop();           //transmit STOP condition

}
```

Program 18-13: A Complete DS1307 Code Example in C *(continued from previous page)*

```
//************************************************************
int main (void)
{
      unsigned char i,j,k;
      rtc_init();
      rtc_setTime(0x19,0x45,0x30);      //19:45:30 (hh:mm:ss)
      rtc_setDate(0x09,0x01,0x10);      //09:01:10 (yy:mm:dd)
      usart_init();
      rtc_getTime(&i,&j,&k);
      usart_send_packedBCD(i);
      usart_send_packedBCD(j);
      usart_send_packedBCD(k);
      rtc_getDate(&i,&j,&k);
      usart_send_packedBCD(i);
      usart_send_packedBCD(j);
      usart_send_packedBCD(k);


      while(1);                         //stop here
      return 0;
}
```

Program 18-13: A Complete DS1307 Code Example in C *(continued from previous page)*

Review Questions

1. True or false. All of the RAM contents of the DS1307 are nonvolatile.
2. How many bytes of RAM in the DS1307 are set aside for the clock and date?
   (a) 7 bytes
   (b) 8 bytes
   (c) 56 bytes
   (d) 64 bytes
3. How many bytes of RAM in the DS1307 are set aside for general-purpose applications?
   (a) 7 bytes
   (b) 8 bytes
   (c) 56 bytes
   (d) 64 bytes
4. True or false. The DS1307 has a single pin for data.
5. Which pin of the DS1307 is used for clock in I2C connection?
6. What is the common voltage for Vbat in the DS1307?
7. True or false. The value of the CH bit is zero at power-up time.
8. What is the address location for the control register?
   (a) 07H
   (b) 08H
   (c) 56H
   (d) 64H

SECTION 18.5: TWI PROGRAMMING WITH CHECKING STATUS REGISTER

In this section we discuss TWI programming with checking the value of status register. By checking the value of the status register you can monitor the TWI module current state and operation. This helps you to detect an error when it happens and resolve it at the same time. This is an advanced topic and used only if you are connecting I2C to multiple masters.

As we mentioned before, there are four modes of operation: master transmitter, master receiver, slave transmitter, and slave receiver. We will discuss each mode separately because each mode has its own special status codes. For each mode of operation there is a flowchart that shows the sequence of steps in each mode and also a figure that summarizes most of the status values for each mode in a single table.

Programming of the AVR TWI in master transmitter operating mode

Figure 18-18 shows the steps of programming the AVR TWI in master transmitter mode. Here we focus on each step in more detail:

Initialization

To initialize the TWI module to operate in master operating mode, we should do the following steps:

1. Set the TWI module clock frequency by setting the values of the TWBR register and the TWPS bits in the TWSR register.
2. Enable the TWI module by setting the TWEN bit in the TWCR register to one.

Transmit START condition

To start data transfer in master operating mode, we must transmit a START condition. To transmit a START condition we should do the following steps:

1. Set the TWEN, TWSTA, and TWINT bits of TWCR to one. Setting the TWEN bit to one enables the TWI module. Setting the TWSTA bit to one tells the TWI to initiate a START condition when the bus is free, and setting the TWINT bit to one clears the interrupt flag to initiate operation of the TWI module to transmit a START condition.
2. Poll the TWINT flag in the TWCR register to see when the START condition is completely transmitted.
3. When the TWINT flag is set to one, check the value of the status register to see if the START condition transmitted successfully. Notice that you have to mask the two LSB bits of the status register to get ride of prescalers. If the status value is 0x08 it indicates that the START condition has been transmitted successfully.

Send SLA + W

To send SLA + W, after transmitting the START condition, we should do the following steps:

1. Copy SLA + W to the TWDR.

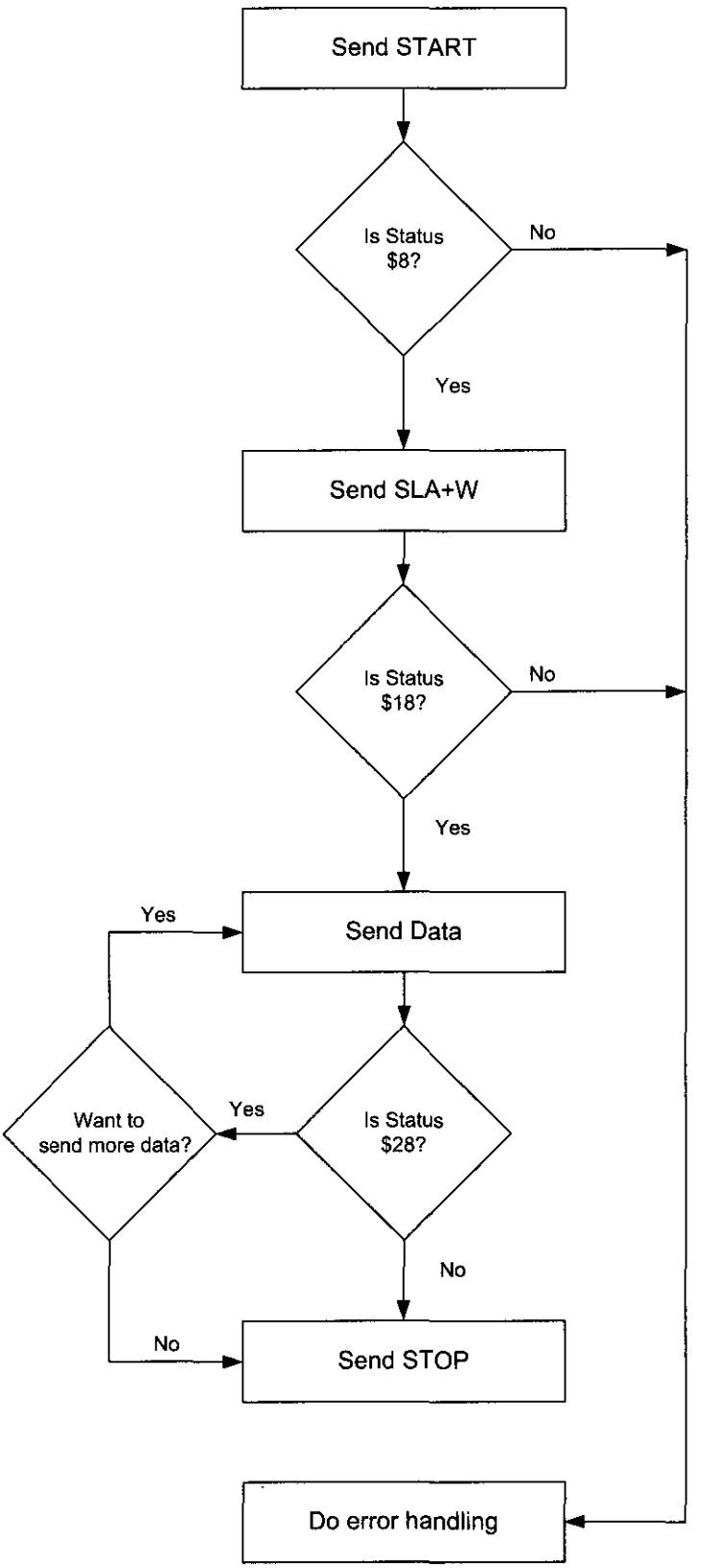


Figure 18-18. Programming Steps of Master Transmitter Mode with Checking of Flags

2. Set the TWEN and TWINT bits of the TWCR register to one to start sending the byte.
3. Poll the TWINT flag in the TWCR register to see when the byte is completely transmitted.
4. When the TWINT flag is set to one, you should check the value of the status register to see if the SLA + W is transmitted successfully. If the status value is 0x18, it indicates that the SLA + W has been transmitted and ACK received successfully.

Send data

To send data, after transmitting of SLA + W, we should do the following steps:

1. Copy the byte of data to the TWDR.
2. Set the TWEN and TWINT bits of the TWCR register to one to start sending the byte.
3. Poll the TWINT flag in the TWCR register to see whether the byte is completely transmitted.
4. When the TWINT flag is set to one, you should check the value of the status register to see if the data has been transmitted successfully and the value of ACK was as expected. Notice that NACK does not necessarily indicate an error; it may indicate that no more data needs to be transmitted. If the status value indicates that ACK is received (0x28) you can either transmit a STOP condition or repeat this function (Send Data) to transmit more data; otherwise, you should transmit a STOP condition.

Transmit STOP condition

To stop data transfer, we must transmit a STOP condition. This is done by setting the TWEN, TWSTO, and TWINT bits of the TWCR register to one. Notice that we cannot poll the TWINT flag after transmitting a STOP condition.

Figure 18-19 shows the meanings of the different values of the status register and possible responses to each of them.

Initialization:	Set values of TWBR register and prescaler bits TWCR = 0x04 TWCR = (1<<TWEN)|(1<<TWINT)|(1<<TWSTA)	Enable TWI Transmit START condition

Status	Meaning	Your Response	Next Action By TWI module
$8	START condition has been transmitted	TWDR = SLA+W TWCR =(1<<TWEN)|(TWINT)	SLA + W will be transmitted ACK or NACK will be returned
$18	SLA + W transmitted. ACK has been received	TWDR = DATA TWCR =(1<<TWEN)|(TWINT)	DATA byte will be Transmitted ACK or NACK will be returned
$20	SLA + W transmitted. NACK has been received	TWCR =(1<<TWEN)|(TWINT)|(TWSTO)	STOP condition will be transmitted
$28	Data byte has been transmitted. ACK has been received.	TWDR = DATA TWCR =(1<<TWEN)|(TWINT) OR TWCR =(1<<TWEN)|(TWINT)|(TWSTO)	DATA byte will be Transmitted ACK or NACK will be returned STOP condition will be transmitted
$30	Data transmitted. NACK received	TWCR =(1<<TWEN)|(TWINT)|(TWSTO)	STOP condition will be transmitted

Figure 18-19. TWSR Register Values for Master Transmitter

Program 18-14 shows how a master writes 11110000 on a slave with address 1101000. The program checks the value of the status register in each step of the operation.

```
.INCLUDE "M32DEF.INC"

      LDI   R21,HIGH(RAMEND);set up stack
      OUT   SPH,R21
      LDI   R21,LOW(RAMEND)
      OUT   SPL,R21

      CALL  I2C_INIT         ;initialize TWI module
      CALL  I2C_START        ;transmit START condition
      CALL  I2C_READ_STATUS  ;read status register
      CPI   R26, 0x08        ;was START transmitted correctly?
      BRNE  ERROR            ;else jump to error function
      LDI   R27, 0b11010000  ;SLA (11010000) + W(0)
      CALL  I2C_WRITE        ;write R27 to I2C bus
      CALL  I2C_READ_STATUS  ;read status register
      CPI   R26, 0x18        ;was SLA+W transmitted, ACK received?
      BRNE  ERROR            ;else jump to error function
      LDI   R27, 0b11110000  ;data to be transmitted
      CALL  I2C_WRITE        ;write R27 to I2C bus
      CALL  I2C_READ_STATUS  ;read status register
      CPI   R26, 0x28        ;was data transmitted, ACK received?
      BRNE  ERROR            ;else jump to error function
      CALL I2C_STOP          ;transmit STOP condition
HERE: RJMP  HERE             ;wait here forever
ERROR:                       ;you can type error handler here
      LDI   R21,0xFF
      OUT   DDRA,R21         ;Port A is output
      OUT   PORTA,R26        ;send error code to Port A
      RJMP  HERE             ;some error code
;*********************************************************
I2C_INIT:
      LDI   R21, 0
      OUT   TWSR,R21         ;set prescaler bits to zero
      LDI   R21, 0x47        ;move 0x47 into R21
      OUT   TWBR,R21         ;clock frequency is 50k (XTAL=50MHZ)
      LDI   R21, (1<<TWEN)   ;move 0x04 into R21
      OUT   TWCR,R21         ;enable the TWI
      RET

;*********************************************************
I2C_START:
      LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
      OUT   TWCR,R21         ;transmit a START condition
WAIT1:
      IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;skip next line if TWINT is 1
```

Program 18-14: Writing a Byte in Master Mode with Status Checking

```
      RJMP  WAIT1             ;jump to WAIT1 if TWINT is 1
      RET
;*********************************************************
I2C_WRITE:
      OUT   TWDR, R27         ;move the byte into TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21         ;configure TWCR to send TWDR
WAIT3:
      IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWINT        ;skip next line if TWINT is 1
      RJMP  WAIT3             ;jump to WAIT3 if TWINT is 1
      RET
;*********************************************************
I2C_STOP:
      LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
      OUT   TWCR, R21         ;transmit STOP condition
      RET
;*********************************************************
I2C_READ_STATUS:
      IN    R26, TWSR         ;read status register into R21
      ANDI  R26, 0xF8         ;mask the prescaler bits
      RET
```

Program 18-14: Writing a Byte in Master Mode with Status Checking *(cont. from prev. page)*

Program 18-15 is the C version of Program 18-10 and shows how a master writes 11110000 to a slave with address 1101000. The program checks the value of the status register in each step of the operation.

```
#include <avr/io.h>

void i2c_write(unsigned char data)
{
   TWDR = data ;
   TWCR = (1<< TWINT)|(1<<TWEN);
   while ((TWCR & (1 <<TWINT)) == 0);
}
//*********************************************************
void i2c_start(void)
{
   TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
   while ((TWCR & (1 << TWINT)) == 0);
}
//*********************************************************
void i2c_showError(unsigned char er)
{
   DDRA = 0xFF;
   PORTA = er;
}
```

Program 18-15: Writing a Byte in Master Mode with Status Checking in C

```
//************************************************************
unsigned char i2c_readStatus(void)
{
  unsigned char i = 0;
  i = TWSR & 0xF8;
  return i;
}
//************************************************************
void i2c_stop()
{
  TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
}
//************************************************************
void i2c_init(void)
{
  TWSR=0x00;                  //set prescaler bits  to zero
  TWBR=0x47;                  //SCL frequency is 50K for XTAL = 8M
  TWCR=0x04;                  //enable the TWI module
}
//************************************************************

int main (void)
{
  unsigned char s = 0;
  i2c_init();
  i2c_start();                //transmit START condition
  s = i2c_readStatus();
  if (s != 0x08)
  {
     i2c_showError(s);
     return 0;
  }
  i2c_write(0b11010000);      //transmit SLA + W(0)
  s = i2c_readStatus();
  if (s != 0x18)
  {
     i2c_showError(s);
     return 0;
  }
  i2c_write(0b11110000);      //transmit data
  s = i2c_readStatus();
  if (s != 0x28)
  {
     i2c_showError(s);
     return 0;
  }
  i2c_stop();                 //transmit STOP condition
  while(1);                   //stay here forever
  return 0;
}
```

Program 18-15: Writing a Byte in Master Mode with Status Checking in C *(continued)*

Programming of the AVR TWI in master receiver operating mode

The steps to program the AVR TWI to operate in master receiver mode are somewhat similar to the steps for programming for master transmitter mode. Figure 18-20 shows the steps for programming of the AVR TWI in master receiver mode. Here we focus on each step in more detail:

Initialization

To initialize the TWI module to operate in master operating mode, we should do the following steps:

1. Set the TWI module clock frequency by setting the values of the TWBR register and the TWPS bits in the TWSR register.
2. Enable the TWI module by setting the TWEN bit in the TWCR register to one.

Transmit START condition

To start data transfer in master operating mode, we must transmit a START condition. To transmit a START condition we should do the following steps:

1. Set the TWEN, TWSTA, and TWINT bits of TWCR to one. Setting the TWEN bit to one enables the TWI module. Setting the TWSTA bit to one tells the TWI module to initiate a START condition when the bus is free, and setting the TWINT bit to one clears the interrupt flag to initiate operation of the TWI module to transmit a START condition.
2. Poll the TWINT flag in the TWCR register to see when the START condition is completely transmitted.
3. When the TWINT flag is set to one, check the value of the status register to see if the START condition was successfully transmitted. Notice that you have to mask the two LSB bits of the status register to get rid of prescalers. If the status value is 0x08 it indicates that the START condition was successfully transmitted.

Send SLA + R

To send SLA + R, after transmitting a START condition, we should do the following steps:

1. Copy SLA + R to the TWDR.
2. Set the TWEN and TWINT bits of the TWCR register to one to start sending the byte.
3. Poll the TWINT flag in the TWCR register to see whether the byte has completely transmitted.
4. When the TWINT flag is set to one, you should check the value of status register to see if the SLA + R transmitted successfully. 0x40 means that the SLA + R transmitted and ACK was successfully received.

Receive data return NACK

If we want to receive only one byte of data, we should receive data and return NACK by doing the following steps:

1. Set the TWEN and TWINT bits of the TWCR register to one to start receiving a byte.
2. Poll the TWINT flag in the TWCR register to see whether a byte was com-

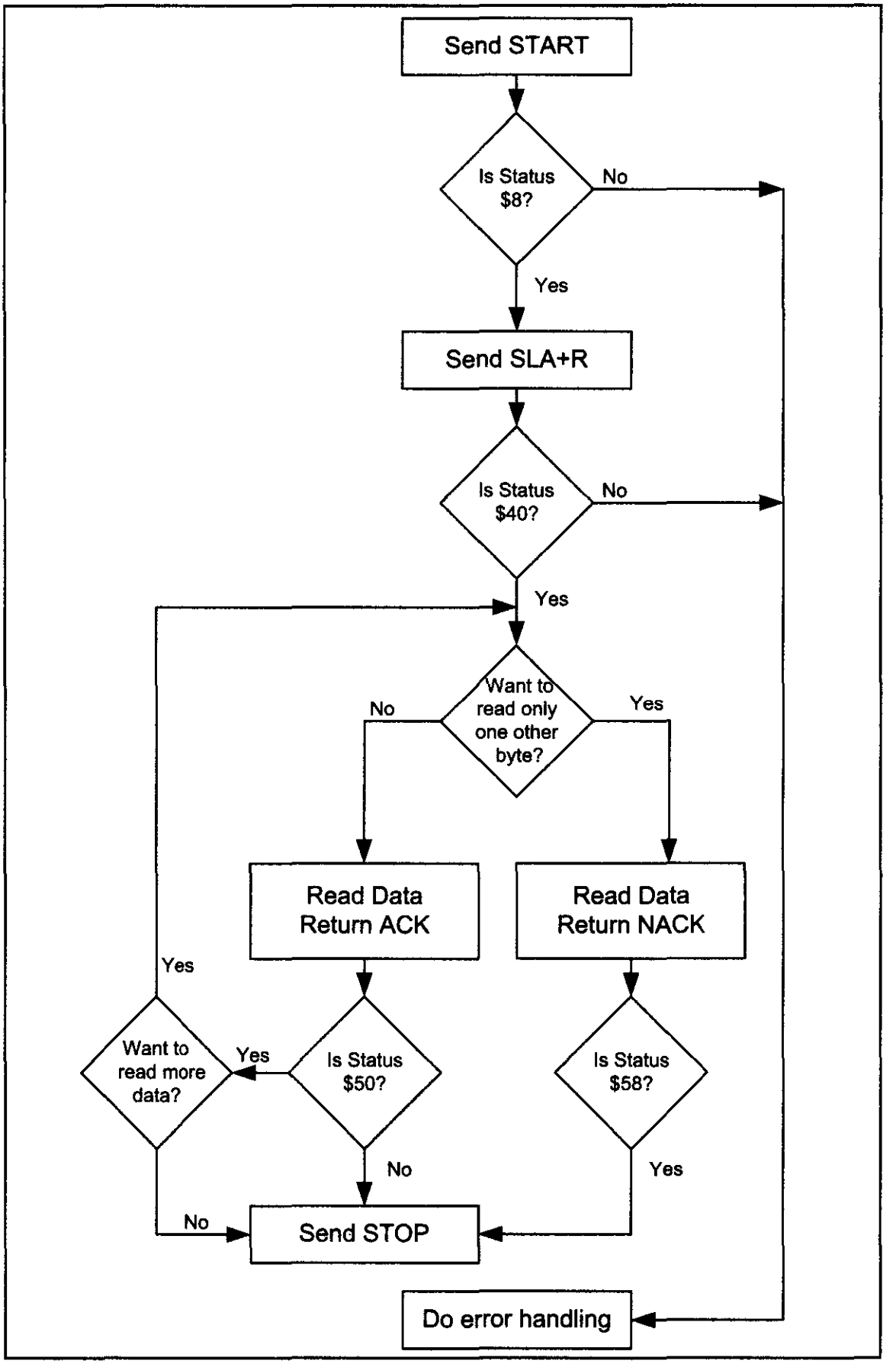


Figure 18-20. TWI Programming Steps of Master Receiver Mode with Checking of Flags

pletely received.
3. Copy the received byte from the TWDR.
4. When the TWINT flag is set to one, you should check the value of the status register to see if the byte was received successfully. 0x58 means that a byte of data was received and NACK returned successfully. After this step we should transmit a STOP condition.

Receive data and return ACK

If we want to receive more than one byte of data, we should receive data and return ACK by doing the following steps:

1. Set the TWEN, TWINT, and TWEA bits of the TWCR register to one to receive a byte of data and return ACK.
2. Poll the TWINT flag in the TWCR register to see when a byte has been received completely.
3. Copy the received byte from the TWDR.
4. When the TWINT flag is set to one, you should check the value of the status register to see if the byte was received successfully. 0x50 means that a byte of data was received and ACK returned successfully. Now you can repeat this step to receive one or more bytes of data, or you can run the "Receive Data Return NACK" function to receive only one other byte of data. Also, you can transmit a STOP condition to finish receiving data.

Transmit STOP condition

To stop data transfer, we must transmit a STOP condition. This is done by setting the TWEN, TWSTO, and TWINT bits of the TWCR register to one. Notice that we cannot poll the TWINT flag after transmitting a STOP condition.

Figure 18-21 shows the meanings of different values of the status register and possible responses to each of them in master receiver operating mode.

Initialization:	TWCR = 0x04 TWCR = (1<<TWEN)|(1<<TWINT)|(1<<TWSTA)	Enable TWI Transmit START condition.

Status	Meaning	Your Response	Next Action By TWI module
$8	START condition has been transmitted	TWDR = SLA + R (1) TWCR =(1<<TWEN)|(TWINT)	SLA + R will be transmitted ACK or NACK will be returned
$40	SLA + R has been transmitted. ACK has been received	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	DATA byte will be received ACK will be returned
		OR TWCR =(1<<TWEN)|(TWINT)	DATA byte will be received NACK will be returned
$48	SLA + R transmitted. NACK received	TWCR =(1<<TWEN)|(TWINT)|(TWSTO)	STOP condition will be transmitted
$50	Data byte has been received. ACK has been returned.	DATA = TWDR TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Another DATA byte will be received ACK will be returned
		OR DATA = TWDR TWCR =(1<<TWEN)|(TWINT)	Another DATA byte will be received NACK will be returned
$58	Data byte received. NACK ACK returned.	DATA = TWDR TWCR =(1<<TWEN)|(TWINT)|(TWSTO)	STOP condition will be transmitted

Figure 18-21. TWSR Register Values for Master Receiver Operating Mode

Program 18-15 shows how a master reads a byte from a slave with address `1101000` and displays the result on Port A. The program checks the value of the

status register in each step of the operation.

```
.INCLUDE "M32DEF.INC"
      LDI   R21,HIGH(RAMEND);set up stack
      OUT   SPH,R21
      LDI   R21,LOW(RAMEND)
      OUT   SPL,R21
      LDI   R21,0xFF
      OUT   DDRA,R21          ;Port A is output
      CALL  I2C_INIT          ;initialize TWI module
      CALL  I2C_START         ;transmit START condition
      CALL  I2C_READ_STATUS   ;read status register
      CPI   R26, 0x08         ;was start transmitted correctly?
      BRNE  ERROR             ;else jump to error function
      LDI   R27, 0b11010001   ;SLA (11010000) + R(1)
      CALL  I2C_WRITE         ;write R27 to I2C bus
      CALL  I2C_READ_STATUS   ;read status register
      CPI   R26, 0x40         ;was SLA+R transmitted, ACK received?
      BRNE  ERROR             ;else jump to error function
      CALL  I2C_READ
      CALL  I2C_READ_STATUS   ;read status register
      CPI   R26, 0x58         ;was data transmitted, ACK received?
      BRNE  ERROR             ;else jump to error function
      OUT   PORTA,R27
      CALL I2C_STOP           ;transmit STOP condition
HERE: RJMP  HERE              ;wait here forever
ERROR:RJMP HERE               ;you can type error handler here
;*********************************************************
I2C_INIT:
      LDI   R21, 0
      OUT   TWSR,R21          ;set prescaler bits to zero
      LDI   R21, 0x47         ;move 0x47 into R21
      OUT   TWBR,R21          ;SCL freq. is 50k for 8 MHz XTAL
      LDI   R21, (1<<TWEN)    ;move 0x04 into R21
      OUT   TWCR,R21          ;enable the TWI
      RET
;*********************************************************
I2C_START:
      LDI   R21, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
      OUT   TWCR,R21          ;transmit a START condition
WAIT1:
      IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWINT        ;skip next line if TWINT is 1
      RJMP  WAIT1             ;jump to WAIT1 if TWINT is 1
      RET
;*********************************************************
I2C_WRITE:
      OUT   TWDR, R27         ;move the byte into TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21         ;configure TWCR to send TWDR
```

Program 18-16: TWI Reading a Byte in Master Mode with Status Checking

```
W3:   IN    R21, TWCR              ;read control register into R21
      SBRS  R21, TWINT             ;skip next line if TWINT is 1
      RJMP  W3                     ;jump to W3 if TWINT is 1
      RET
;*********************************************************
I2C_READ:
      LDI   R21,(1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21
W2:   IN    R21, TWCR              ;read control register into R21
      SBRS  R21, TWINT             ;skip next line if TWINT is 1
      RJMP  W2                     ;jump to W2 if TWINT is 0
      IN    R27, TWDR              ;read received data into R21
      RET
;*********************************************************
I2C_STOP:
      LDI   R21, (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)
      OUT   TWCR, R21              ;transmit STOP condition
      RET
;*********************************************************
I2C_READ_STATUS:
      IN    R26, TWSR              ;read status register into R21
      ANDI  R26, 0xF8              ;mask the prescaler bits
      RET
```

Program 18-16: TWI Reading a Byte in Master Mode with Status Checking *(continued)*

Program 18-17 is the C version of Program 18-16.

```
#include <avr/io.h>
void i2c_showError(unsigned char er)
{
  DDRA = 0xFF;
  PORTA = er;
} //*************************************************************
unsigned char i2c_readStatus(void)
{
  unsigned char i = 0;
  i = TWSR & 0xF8;
  return i;
} //*************************************************************
void i2c_init(void)
{
  TWSR=0x00;                  //set prescaler bits to zero
  TWBR=0x47;                  //SCL frequency is 50K for XTAL=8M
  TWCR=0x04;                  //enable the TWI module
} //*************************************************************
void i2c_start(void)
{
  TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
  while ((TWCR & (1 << TWINT)) == 0);
}  //*************************************************************
```

Program 18-17: TWI Reading a Byte in Master Mode with Status Checking in C

```
void i2c_write(unsigned char data)
{
   TWDR = data;
   TWCR = (1<< TWINT)|(1<<TWEN);
   while ((TWCR & (1 <<TWINT)) == 0);
} //************************************************************
unsigned char i2c_read(unsigned char isLast)
{
   if (isLast == 0)          //if want to read more than 1 byte
     TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWEA);
   else                      //if want to read only one byte
     TWCR = (1<< TWINT)|(1<<TWEN);
   while ((TWCR & (1 <<TWINT)) == 0);
   return TWDR;
} //************************************************************
void i2c_stop()
{
   TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
} //************************************************************
int main (void)
{
   DDRA = 0xFF;              //Port A is output
   unsigned char s,i;
   i2c_init();
   i2c_start();              //transmit START condition
   s = i2c_readStatus();
   if (s != 0x08)
   {
      i2c_showError(s);
      return 0;
   }
   i2c_write(0b11010001);    //transmit SLA + R(1)
   s = i2c_readStatus();
   if (s != 0x40)
   {
      i2c_showError(s);
      return 0;
   }
   i=i2c_read(1);
   s = i2c_readStatus();
   if (s != 0x58)
   {
      i2c_showError(s);
      return 0;
   }
   PORTA= i;                 //show the byte on Port A
   i2c_stop();               //transmit STOP condition
   while(1);                 //stay here forever
   return 0;
}
```

Program 18-17: TWI Reading a Byte in Master Mode with Status Checking in C *(continued)*

Programming of the AVR TWI in slave transmitter operating mode

Before programming the AVR to operate in slave mode, there are some points that we must pay attention to. As we mentioned before, the slave device, regardless of whether it is receiver or transmitter, does not generate the clock pulse. To control the clock rate and let the software to complete its job, the slave device uses clock stretching. The slave device does not start or stop a transmission; it listens to the bus and replies when it is addressed by a master device.

In the slave transmitter mode, one or more bytes of data are transmitted from the slave to a master receiver. The following steps show the transmission of one or more bytes of data in slave transmitter mode.

Initialization

To initialize the TWI module to operate in slave operating mode, we should do the following steps:

1. Set the TWAR. As we mentioned before, the upper seven bits of TWAR are the slave address. It is the address to which the TWI will respond when addressed by a master. The eighth bit is TWGCE. If you set this bit to one, the TWI will respond to the general call address ($00); otherwise, it will ignore the general call address.
2. Enable the TWI module by setting the TWEN bit in the TWCR register to one.
3. Set the TWEN and TWEA bits of TWCR to one to enable the TWI and acknowledge generation.

Wait to be addressed for read

In slave mode, the TWI hardware waits until it is addressed by its own slave address (or the general call address, if enabled) followed by the R/W bit, and then sets the TWINT flag and updates the status register. If the R/W bit is zero (write), it means that the slave should operate in slave receiver mode; otherwise, the slave should operate in slave transmitter mode. Notice that you can not directly read the value of the R/W bit. Instead you should read the value of the status register. Next, we will show how to wait to be addressed by a master device.

1. Poll the TWINT flag in the TWCR register to see whether a byte has received completely.
2. When the TWINT flag is set to one, you should check the value of the status register to see if the SLA + R is received successfully. $A8 means that the SLA + R was received and ACK returned successfully.

Now if you want to transmit only one byte of data you should run the "Send Data and Wait for NACK" function. Otherwise, if you want to send more than one byte of data you should run the "Send Data and Wait for ACK" function. Next we will examine each function in detail.

Send data and wait for ACK

In slave transmitter mode, if you want to transmit more than one byte of data you should send a byte of data and wait for ACK by doing the following steps:

1. Copy the byte of data to the TWDR.
2. Set the TWEN, TWINT, and TWEA bits of the TWCR register to one to send

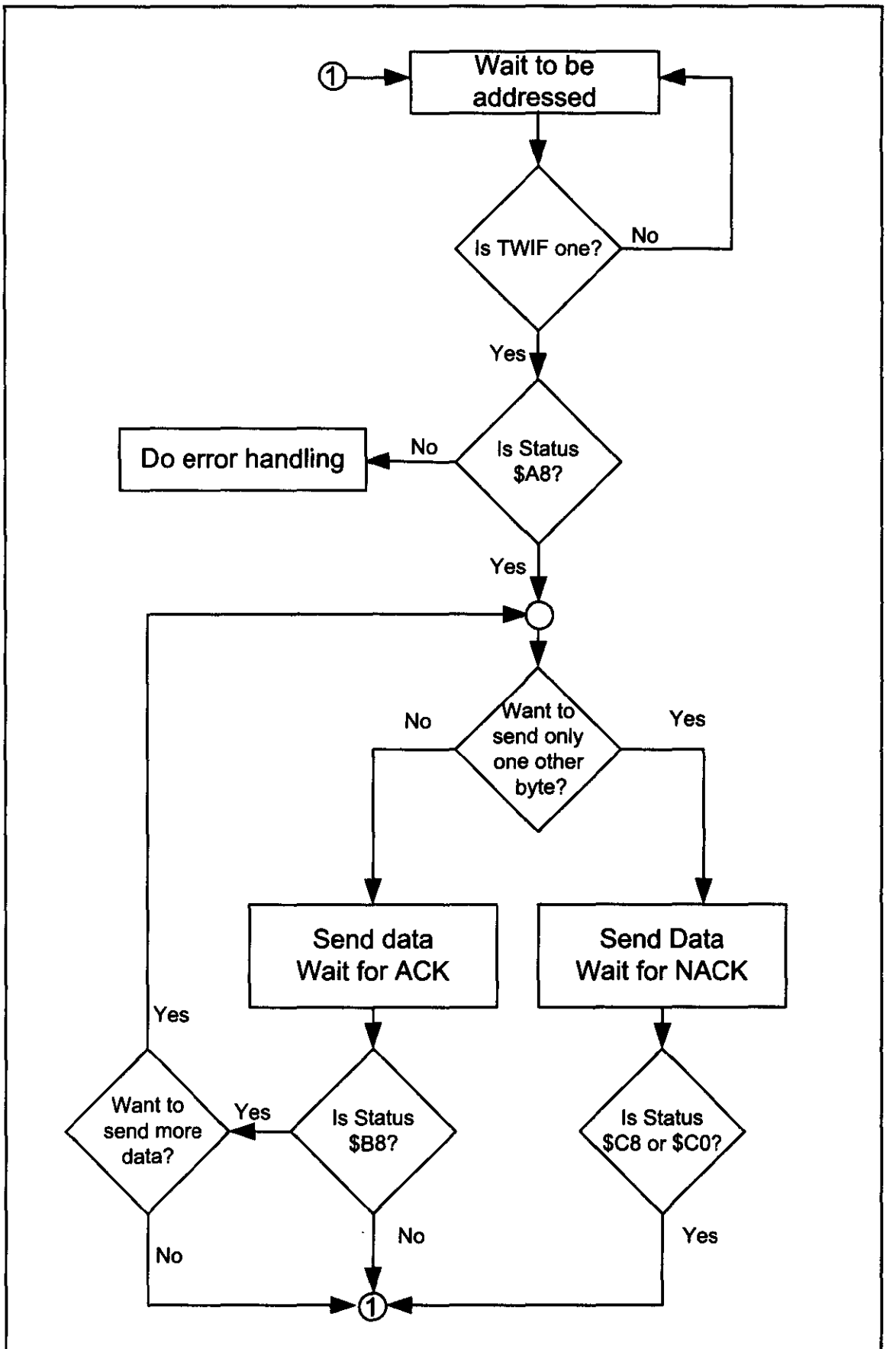


Figure 18-22. TWI Programming Steps of Slave Transmitter Mode with Checking of Flags

a byte of data and wait for ACK.

3. Poll the TWINT flag in the TWCR register to see whether the byte transmitted completely.
4. When the TWINT flag is set to one, you should check the value of the status register to see if the data transmitted successfully and the value of ACK was as expected. Notice that NACK does not necessarily indicate an error; it may indicate that no more data needs to be transmitted. If the status value indicates that NACK was received ($0C), it means that the current transmission section is finished and you should start from the beginning. If the status value indicates that ACK was received (0xC8), you can either repeat this function to transmit more than one byte of data or you can run the "Send Data and Wait for NACK" function to transmit only one byte of data.

Send data and wait for NACK

In slave transmitter mode, to transmit another byte of data you should send a byte of data and wait for NACK by doing the following steps:

1. Copy the byte of data to the TWDR.
2. Set the TWEN and TWINT bits of the TWCR register to one to send a byte and wait for NACK.
3. Poll the TWINT flag in the TWCR register to see when the byte has been transmitted completely.
4. When the TWINT flag is set to one, you should check the value of the status register. If the status value is $0C, it indicates that NACK has been received. If the value of status register is $C8, it means that ACK was received. In both cases you have to go to the "Wait to be addressed" mode because you have not set the TWEA bit in step 2 saying that you want to transmit only one other byte of data.

Notice that in most applications you can use the "Send Data and Wait for ACK" function instead of the "Send Data and Wait for NACK" function. We rec-

Initialization:	TWCR = 0x04 TWAR = the address of Slave TWCR = (1<<TWEN)|(1<<TWIF)|(1<<TWEA)	Enable TWI Set the slave address Enable Acknowledging by slave

Status	Meaning	Your Response	Next Action By TWI module
$A8	Own SLA+R received ACK returned	TWDR = DATA TWCR =(1<<TWEN)|(TWINT)|(TWEA)	DATA byte will be transmitted Wait for ACK
		OR TWDR = DATA TWCR =(1<<TWEN)|(TWINT)	DATA byte will be transmitted Wait for NACK
$B8	Data has been transmitted ACK received	TWDR = DATA TWCR =(1<<TWEN)|(TWINT)|(TWEA)	DATA byte will be transmitted Wait for ACK
		OR TWDR = DATA TWCR =(1<<TWEN)|(TWINT)	DATA byte will be transmitted Wait for NACK
$C0	Data has been transmitted NACK received	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Start from beginning and wait to be addressed
		OR TWCR =(1<<TWEN)|(TWINT)	Start from beginning but do not respond to Its address (Sleep)
$C8	Data transmitted ACK received but you wanted NACK (TWEA was 0 in last command)	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Start from beginning and wait to be addressed
		OR TWCR =(1<<TWEN)|(TWINT)	Start from beginning but do not respond to Its address (Sleep)

Figure 18-23. TWSR Register Values for Slave Transmitter Operating Mode

ommend that you use the first one.

Program 18-18 shows how to initialize the TWI module to operate in slave transmitter mode. In this program the TWI module listens to the bus and waits to be addressed by a master device. Then it transmits the letter 'G' to the master device.

```
.INCLUDE "M32DEF.INC"

      LDI   R21,HIGH(RAMEND);set up stack
      OUT   SPH,R21
      LDI   R21,LOW(RAMEND)
      OUT   SPL,R21

      CALL  I2C_INIT          ;initialize the TWI module as slave
      CALL  I2C_LISTEN        ;listen to the bus to be addressed
      CALL I2C_READ_STATUS    ;read the status value into R26
      CPI   R26, 0xA8         ;addressed as slave tranmitter ?
      BRNE  ERROR             ;else jump to error function
      LDI   R27, 'G'          ;load 'G' into R21
      CALL  I2C_WRITE
      CALL I2C_READ_STATUS    ;read the status value into R26
      CPI   R21, 0xc0         ;was data transmitted, NACK received?
      BRNE  ERROR             ;else jump to error function

HERE:
      RJMP  HERE              ;wait here forever
ERROR:                        ;you can type error handler here
      LDI   R21,0xFF
      OUT   DDRA,R21          ;Port A is output
      OUT   PORTA,R26
      RJMP  HERE
;*************************************************************

I2C_INIT:
      LDI   R21, 0x10         ;load 00010000 into R21
      OUT   TWAR,R21          ;set address register
      LDI   R21, (1<<TWEN)    ;move 0x04 into R21
      OUT   TWCR,R21          ;enable the TWI
      LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
      OUT   TWCR,R21          ;enable TWI and ACK(can't be ignored)
      RET
;*************************************************************

I2C_LISTEN:
W1:
      IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWINT        ;skip next intruction if TWINT is 1
      RJMP  W1                ;jump to W1 if TWINT is 0
      RET
```

Program 18-18: Writing a Byte in Slave Mode with Status Checking

```
;*********************************************************
I2C_WRITE:

      OUT   TWDR, R27         ;move R21 to TWDR
      LDI   R21, (1<<TWINT)|(1<<TWEN)
      OUT   TWCR, R21         ;configure TWCR to send TWDR
W2:
      IN    R21, TWCR         ;read control register into R21
      SBRS  R21, TWINT ;skip next intruction if TWINT is 1
      RJMP  W2                ;jump to W2 if TWINT is 0
      RET

;*********************************************************
I2C_READ_STATUS:
      IN    R26, TWSR         ;read status register into R21
      ANDI  R26, 0xF8         ;mask the prescaler bits
      RET
```

Program 18-18: Writing a Byte in Slave Mode with Status Checking *(cont. from prev. page)*

Program 18-19 is the C version of Program 18-18. Program 18-19 shows how to initialize the TWI module to operate in slave transmitter mode. In Program 18-19 the TWI module listens to the bus and waits to be addressed by a master device. Then it transmits the letter 'G' to the master device.

```
#include <avr/io.h>                          //standard AVR header

void i2c_showError(unsigned char er)
{
   DDRA = 0xFF;
   PORTA = er;
} //*************************************************************

unsigned char i2c_readStatus(void)
{
   unsigned char i = 0;
   i = TWSR & 0xF8;
   return i;
} //*************************************************************

void i2c_initSlave(unsigned char slaveAddress)
{
   TWCR = 0x04;                              //enable TWI module
   TWAR = slaveAddress;                      //set the slave address
   TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWEA);//init TWI module
}
```

Program 18-19: Writing a Byte in Slave Mode with Status Checking in C

```
//************************************************************

void i2c_send(unsigned char data)
{
   TWDR = data;                              //copy data to TWDR
   TWCR = (1<< TWINT)|(1<<TWEN);             //start transmission
   while ((TWCR & (1 <<TWINT))==0);          //wait to complete
}

//************************************************************

void i2c_listen()
{
   while ((TWCR & (1 <<TWINT))==0);          //wait to be addressed
}

//************************************************************

int main (void)
{
   i2c_initSlave(0x10);                      //init TWI module as
                                             //slave with address
                                             //0b0001000 and do not
                                             //accept general call
   i2c_listen();                             //listen to be addressed

   unsigned char s,i;
   s = i2c_readStatus();
   if (s != 0xA8)
   {
      i2c_showError(s);
      return 0;
   }
   i2c_send('G');
   s = i2c_readStatus();
   if (s != 0xC0)
   {
      i2c_showError(s);
      return 0;
   }

   while(1);                                 //stay here forever
   return 0;

}
```

Program 18-19: Writing a Byte in Slave Mode with Status Checking in C *(continued)*

Programming of the AVR TWI in slave receiver operating mode

In the slave receiver mode, one or more bytes of data are transmitted from a master transmitter to the slave receiver. The following steps show the functions needed to receive one or more bytes of data in slave receiver mode.

Initialization

To initialize the TWI module to operate in slave operating mode, we should do the following steps:

1. Set the TWAR. As we mentioned before, the upper seven bits of TWAR are the slave address. It is the address to which the Two-wire Serial Interface will respond when addressed by a master. The eighth bit is TWGCE. If you set this bit to one, the TWI will respond to the general call address ($00); otherwise, it will ignore the general call address.
2. Enable the TWI module by setting the TWEN bit in the TWCR register to one.
3. Set the TWEN and TWEA bits of TWCR to one to enable the TWI and acknowledge generation.

Wait to be addressed for write

In slave mode, we should do the following steps to wait to be addressed by a master for a write operation.

1. Poll the TWINT flag in the TWCR register to see when a byte has been received completely.
2. When the TWINT flag is set to one, we should check the value of the status register to see if the SLA + W was received successfully. $60 or $70 (for general call) means that the SLA + W was received and ACK returned successfully.

Now if you want to receive only one byte of data you should run the "Receive Data and Return NACK" function. Otherwise, if you want to send more than one byte of data you should run the "Receive Data and Return ACK" function. Next, we will examine each function in detail.

Receive data and Return ACK

In slave receiver mode, if you want to receive more than one byte of data you should receive a byte of data and return ACK by doing the following steps:

1. Set the TWEN, TWINT, and TWEA bits of the TWCR register to one to receive a byte and return ACK.
2. Poll the TWINT flag in the TWCR register to see when a byte has been received completely.
3. When the TWINT flag is set to one, you should check the value of the status register to see if the data was received successfully and ACK was returned. If the status value is $80 or $90 (for general call), it means that a byte of data has been received and ACK was returned. You can either repeat this function to receive more than one bytes of data or you can run the "Receive Data and Return NACK" function to receive only one byte of data.
4. Copy the received byte from the TWDR.

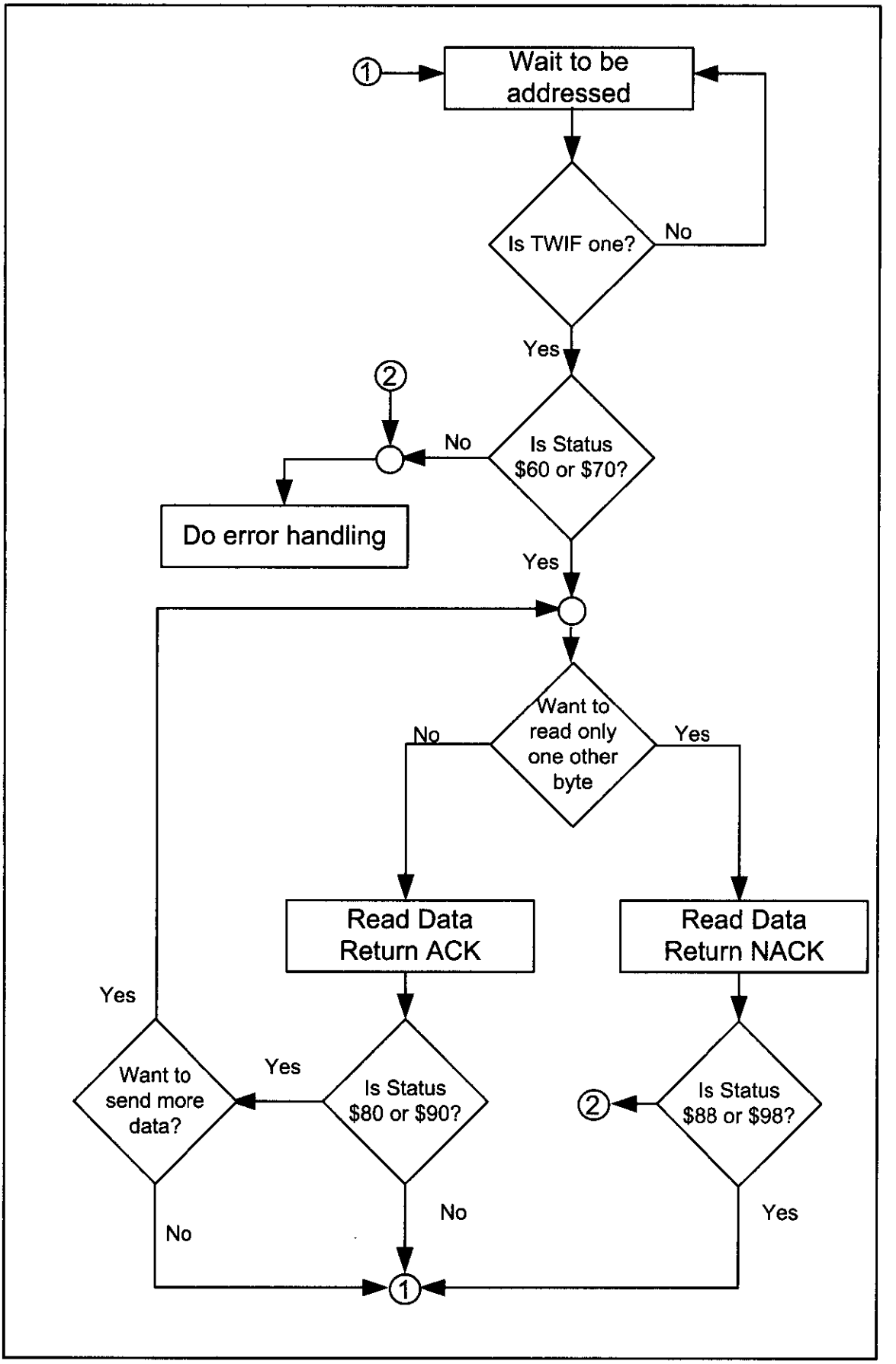


Figure 18-24. TWI Programming Steps of Slave Receiver Mode with Checking of Flags

Receive data and return NACK

In slave receiver mode, if you want to receive one byte of data you should receive the byte of data and return NACK by doing the following steps:

1. Set the TWEN and TWINT bits of the TWCR register to one to receive a byte and return NACK.
2. Poll the TWINT flag in the TWCR register to see when a byte has been received completely.
3. When the TWINT flag is set to one, you should check the value of the status register to see if the data was received successfully and NACK was returned. If the status value is $88 or $98 (for general call), it means that a byte of data was received and NACK was returned.
4. Copy the received byte from the TWDR.

Initialization:	TWAR = the address of Slave TWCR = 0x04 TWCR = (1<<TWEN)|(1<<TWIF)|(1<<TWEA)	Enable TWI Set the slave address Enable Acknowledging by slave

Status	Meaning	Your Response	Next Action By TWI module
$60 ($70 for General Call)	Own SLA+W received ACK returned	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	DATA byte will be received ACK will be returned
		OR TWCR =(1<<TWEN)|(TWINT)	DATA byte will be received NACK will be returned
$80 ($90 for General Call)	Data has been received ACK returned	DATA = TWDR TWCR =(1<<TWEN)|(TWINT)|(TWEA)	DATA byte will be received ACK will be returned
		OR DATA = TWDR TWCR =(1<<TWEN)|(TWINT)	DATA byte will be received NACK will be returned
$88 ($98 for General Call)	Data has been received NACK returned	DATA = TWDR TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Start from beginning and wait to be addressed
		OR DATA = TWDR TWCR =(1<<TWEN)|(TWINT)	Start from beginning but do not respond to its address (Sleep)
$A0	STOP or REPEATED START condition has been received	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Start from beginning and wait to be addressed
		OR TWCR =(1<<TWEN)|(TWINT)	Start from beginning but do not respond to its address (Sleep)

Figure 18-25. TWSR Register Values for Slave Receiver Operating Mode

Program 18-20 shows how to initialize the TWI module to operate in slave receiver mode. This program receives a byte of data and displays it on Port A after being addressed by a master device.

```
.INCLUDE "M32DEF.INC"

     LDI   R21,HIGH(RAMEND);set up stack
     OUT   SPH,R21
     LDI   R21,LOW(RAMEND)
     OUT   SPL,R21

     LDI   R21, 0xFF        ;move 0xFF into R21
     OUT   DDRA,R21         ;set PORTA as ouput

     CALL  I2C_INIT         ;initialize the TWI module as slave
```

Program 18-20: Reading a Byte in Slave Mode with Status Checking

```
      CALL  I2C_LISTEN       ;listen to the bus to be addressed
      CALL  I2C_READ_STATUS
      CPI   R26, 0x60        ;addressed as slave receiver?
      BRNE  ERROR            ;else jump to error function
      CALL  I2C_READ         ;read a byte and copy it to R27
      CALL  I2C_READ_STATUS
      CPI   R26, 0x80        ;addressed as slave receiver?
      BRNE  ERROR            ;else jump to error function
      OUT   PORTA,R27        ;copy R27 to PORTA

HERE:
      RJMP  HERE             ;wait here forever
ERROR:
      RJMP HERE
;*********************************************************

I2C_INIT:
      LDI   R21, 0x10        ;load 00010000 into R21
      OUT   TWAR,R21         ;set address register
      LDI   R21, (1<<TWEN)   ;move 0x04 into R21
      OUT   TWCR,R21         ;enable the TWI
      LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
      OUT   TWCR,R21         ;enable TWI and ACK(can't be ignored)
      RET

;*********************************************************

I2C_LISTEN:
W1:
      IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;skip next intruction if TWINT is 1
      RJMP  W1               ;jump to W1 if TWINT is 0
      RET

;*********************************************************

I2C_READ:
      LDI   R21, (1<<TWINT)|(1<<TWEN)|(1<<TWEA)
      OUT   TWCR, R21        ;configure TWCR to receive TWDR
W2:   IN    R21, TWCR        ;read control register into R21
      SBRS  R21, TWINT       ;skip next line if TWINT is 1
      RJMP  W2               ;jump to W2 if TWINT is 0
      IN    R27,TWDR         ;move received data into R21
      RET

;*********************************************************
I2C_READ_STATUS:
      IN    R26, TWSR        ;read status register into R21
      ANDI  R26, 0xF8        ;mask the prescaler bits
      RET
```

Program 18-20: Reading a Byte in Slave Mode with Status Checking *(cont. from prev. page)*

Program 18-21 is the C version of Program 18-20. This program receives a byte of data and displays it on Port A after being addressed by a master device.

```
#include <avr/io.h>                          //standard AVR header

void i2c_showError(unsigned char er)
{
   DDRA = 0xFF;
   PORTA = er;
}

//*************************************************************

unsigned char i2c_readStatus(void)
{
   unsigned char i = 0;
   i = TWSR & 0xF8;
   return i;
}

//*************************************************************

void i2c_initSlave(unsigned char slaveAddress)
{
   TWCR = 0x04;                              //enable TWI module
   TWAR = slaveAddress;                      //set the slave address
   TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWEA);//init. TWI module
}
//*************************************************************

unsigned char i2c_receive(unsigned char isLast)
{
   if (isLast == 0)          //if want to read more than 1 byte
     TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWEA);
   else                      //if want to read only one byte
     TWCR = (1<< TWINT)|(1<<TWEN);

   while ((TWCR & (1 <<TWINT))==0);    //wait to complete
   return (TWDR);
}

//*************************************************************

void i2c_listen()
{
   while ((TWCR & (1 <<TWINT))==0);    //wait to be addressed
}

//*************************************************************
```

Program 18-21: Reading a Byte in Slave Mode with Status Checking in C

```
int main (void)
{
  DDRA = 0xFF;
   i2c_initSlave(0x10);                  //init. TWI module as
                                         //slave with address
                                         //0b0001000 and do not
                                         //accept general call
  i2c_listen();                          //listen to be addressed

  unsigned char s,i;
  s = i2c_readStatus();
  if (s != 0x60)
  {
     i2c_showError(s);
     return 0;
  }
  i=i2c_receive(0);
  s = i2c_readStatus();
  if (s != 0x80)
  {
     i2c_showError(s);
     return 0;
  }
  PORTA = i;
  while(1);                              //stay here forever
  return 0;
}
```

Program 18-21: Reading a Byte in Slave Mode with Status Checking in C *(continued)*

Review Questions

1. True or false. We can ignore checking the status register when there is more than one master on the bus.
2. True or false. We can enable the TWI module and generate aSTART condition at the same time.
3. How can a slave device read the value of the R/W bit when it is being addressed by a master device?
4. True or false. We can check the status register to see if a STOP condition has been transmitted successfully.
5. What is the value of the status register when SLA + W is received and ACK has been returned?
6. What is the value of the status register when SLA + W is transmitted and ACK has been received?
7. What is the value of the status register when SLA + R is received and ACK has been returned?
8. What is the value of the status register when SLA + W is transmitted and ACK has been received?

SUMMARY

This chapter began by describing the TWI bus connection and protocol. Then we focused on programming of TWI in the AVR. We also discussed the function of each pin of the DS1307 RTC chip. The DS1307 can be used to provide a real-time clock and dates for many applications. Various features of the RTC were explained, and numerous programming examples were given.

PROBLEMS

SECTION 18.1: I2C BUS PROTOCOL

1. True or false. The I2C bus needs an external clock.
2. True or false. The SDA pin is internally pulled up.
3. True or false. The I2C bus needs two wires to transfer data.
4. True or false. The SDA line is output for the master device.
5. True or false. When a device is used as a slave, the SCL is an input pin.
6. True or false. In I2C, the data frame is 8 bits long.
7. True or false. In I2C devices, each bit of information (data, address, ACK/NACK) is transferred with a single clock pulse.
8. True or false. In I2C devices, the 8-bit data is followed by an ACK/NACK.
9. In terms of data pins, what is the difference between the SPI and I2C connections?
10. How does the I2C protocol distinguish between the read and write cycles?

SECTION 18.2: TWI (I2C) IN THE AVR

11. True or false. The AVR uses the term TWI instead of I2C.
12. What are the TWI submodules in AVR?
13. Which unit generates START or STOP conditions in the AVR?
14. True or false. After reading the status register we should mask the 2 LSB bits.
15. Which bits of TWSR are used to specify the clock of the TWI module?
16. Which bit of TWCR enables generation of interrupts when the TWINT flag is set?
17. How can we virtually disconnect the TWI module from the bus?

SECTION 18.3: AVR TWI PROGRAMMING IN ASSEMBLY AND C

18. Write a program to read a byte from a slave with address 0110 100 and write the byte to a slave with address 0110 101.
19. Write a program to operate in slave mode and transmit "Y" to the master when the slave device is addressed. The slave address should be 0110 100.

SECTION 18.4: DS1307 RTC INTERFACING AND PROGRAMMING

20. The DS1307 DIP package is a(n) ____-pin package.
21. Which pin is assigned as GND?

22. Which pin is assigned as V_{cc}?
23. True or false. The DS1307 needs an external battery.
24. True or false. The DS1307 needs an external crystal oscillator.
25. True or false. The DS1307's crystal oscillator and heat affect the time-keeping accuracy.
26. What is the maximum year that the DS1307 can provide?
27. Describe the functions of the SQW/OUT pin.
28 X1 is an _____ (input, output) pin.
29. The SQW/OUT pin is controlled by _____ , _____, and _____ bits.
30. DS1307 has a total of _____bytes of RAM locations.
31. When does the DS1307 switch to a battery energy source?
32. What are the addresses assigned to the real-time clock (time) registers?
33. What are the addresses assigned to the calendar?
34. Which bit is used to set the AM/PM mode?
35. Which bit is used to set the 24-hour mode?
36. At what memory location does the DS1307 store the year 2009?
37. What is the address of the last location of RAM for the DS1307?
38. True or false. The DS1307 provides data in BCD format only.
39. Write a C program to set the time to 9:15:05 PM.
40. Write a C program to set the time to 22:47:19.
41. Write a C program to set the date to May 14, 2009.
42. Write a C program to get the hour and minute data and send it to Port B and Port D.

ANSWERS TO REVIEW QUESTIONS

SECTION 18.1: I2C BUS PROTOCOL

1. True
2. 9 bits. The ninth bit
3. True
4. Clock stretching
5. 4.7 kilohms
6. False

SECTION 18.2: TWI (I2C) IN THE AVR

1. True
2. TWDR, TWAR, TWBR, TWCR, and TWSR
3. By writing 1 to the TWSTA and TWSTO bits, respectively
4. False
5. TWINT
6. False
7. TWEA

SECTION 18.3: AVR TWI PROGRAMMING IN ASSEMBLY AND C

1. False
2. d
3. b and c

4. d
5. c

SECTION 18.4: DS1307 RTC INTERFACING AND PROGRAMMING

1. True
2. a
3. c (64 – 8 = 56 bytes)
4. True
5. SCL
6. 3V
7. False
8. a

SECTION 18.5: TWI PROGRAMMING WITH CHECKING STATUS REGISTER

1. False
2. False. We have to first enable the TWI module by writing one to the TWEN bit and then we can generate a START condition.
3. It should read the value of the status register.
4. False
5. $60
6. $18
7. $A8
8. $40