THE AVR MICROCONTROLLER AND EMBEDDED SYSTEMS

Using Assembly and C

Online Part

Muhammad Ali Mazidi
Sepehr Naimi
Sarmad Naimi

Mazidi & Naimi

To contact authors, use the following email addresses:
Sepehr.Naimi@gmail.com

Visit our website at
http://www.MicroDigitalEd.com

CONTENTS

SECTION 8.2: AVR FUSE BITS

There are some features of the AVR that we can choose by programming the bits of fuse bytes. These features will reduce system cost by eliminating any need for external components.

ATmega32 has two fuse bytes. Tables 8-6 and 8-7 give a short description of the fuse bytes. Notice that the default values can be different from production to production and time to time. In this section we examine some of the basic fuse bits. The Atmel website (http://www.atmel.com) provides the complete description of fuse bits for the AVR microcontrollers. It must be noted that if a fuse bit is incorrectly programmed, it can cause the system to fail. An example of this is changing the SPIEN bit to 1, which disables SPI programming mode. In this case you will not be able to program the chip any more! Also notice that the fuse bits are '0' if they are programmed and '1' when they are not programmed.

In addition to the fuse bytes in the AVR, there are 4 lock bits to restrict access to the Flash memory. These allow you to protect your code from being copied by others. In the development process it is not recommended to program lock bits because you may decide to read or verify the contents of Flash memory. Lock bits are set when the final product is ready to be delivered to market. In this book we do not discuss lock bits. To study more about lock bits you can read the data sheets for your chip at http://www.atmel.com.

Table 8-6: Fuse Byte (High)

Fuse High Byte	Bit No.	Description	Default Value
OCDEN	7	Enable OCD	1 (unprogrammed)
JTAGEN	6	Enable JTAG	0 (programmed)
SPIEN	5	Enable SPI serial program and data downloading	0 (programmed)
CKOPT	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the chip erase	1 (unprogrammed)
BOOTSZ1	2	Select boot size	0 (programmed)
BOOTSZ0	1	Select boot size	0 (programmed)
BOOTRST	0	Select reset vector	1 (unprogrammed)

Table 8-7: Fuse Byte (Low)

Fuse High Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown-out detector trigger level	1 (unprogrammed)
BODEN	6	Brown-out detector enable	1 (unprogrammed)
SUT1	5	Select start-up time	1 (unprogrammed)
SUT0	4	Select start-up time	0 (programmed)
CKSEL3	3	Select clock source	0 (programmed)
CKSEL2	2	Select clock source	0 (programmed)
CKSEL1	1	Select clock source	0 (programmed)
CKSEL0	0	Select clock source	1 (unprogrammed)

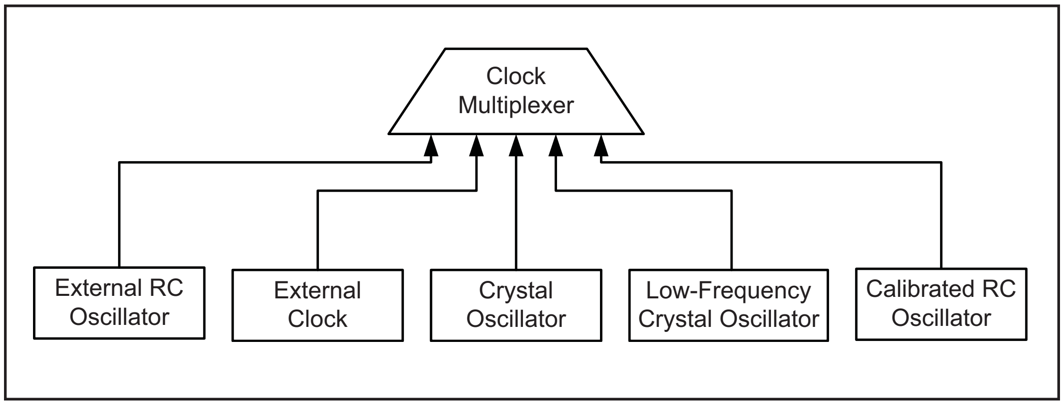


Figure 8-4. ATmega32 Clock Sources

Fuse bits and oscillator clock source

As you see in Figure 8-4, there are different clock sources in AVR. You can choose one by setting or clearing any of the bits CKSEL0 to CKSEL3.

Table 8-8: Internal RC Oscillator Operation Modes

CKSEL3...0	Frequency
0001	1 MHz
0010	2 MHz
0011	4 MHz
0100	8 MHz

CKSEL0–CKSEL3

The four bits of CKSEL3, CKSEL2, CKSEL1, and CKSEL0 are used to select the clock source to the CPU. The default choice is internal RC (0001), which uses the on-chip RC oscillator. In this option there is no need to connect an external crystal and capacitors to the chip. As you see in Table 8-8, by changing the values of CKSEL0–CKSEL3 we can choose among 1, 2, 4, or 8 MHz internal RC frequencies; but it must be noted that using an internal RC oscillator can cause about 3% inaccuracy and is not recommended in applications that need precise timing.

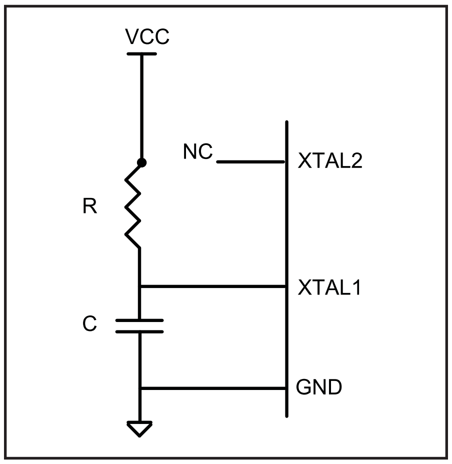


Figure 8-5 External RC

Table 8-9: External RC Oscillator Operation Modes

CKSEL3...0	Frequency (MHz)
0101	<0.9
0110	0.9–3.0
0111	3.0–8.0
1000	8.0–12.0

The external RC oscillator is another source to the CPU. As you see in Figure 8-5, to use the external RC oscillator, you have to connect an external resistor and capacitors to the XTAL1 pin. The values of R and C determine the clock speed. The frequency of the RC oscillator circuit is estimated by the equation $f = 1/(3RC)$. When you need a variable clock source you can use the external RC and replace the resistor with a potentiometer. By turning the potentiometer you will be able to change the frequency. Notice that the capacitor value should be at least 22 pF. Also, notice that by programming the CKOPT fuse, you can enable an internal 36 pF capacitor between XTAL1 and GND, and remove the external capacitor. As you see in Table 8-9, by changing the values of CKSEL0–CKSEL3, we can choose different frequency ranges.

By setting CKSEL0...3 bits to 0000, we can use an external clock source for the CPU. In Figure 8-6a you see the connection to an external clock source.

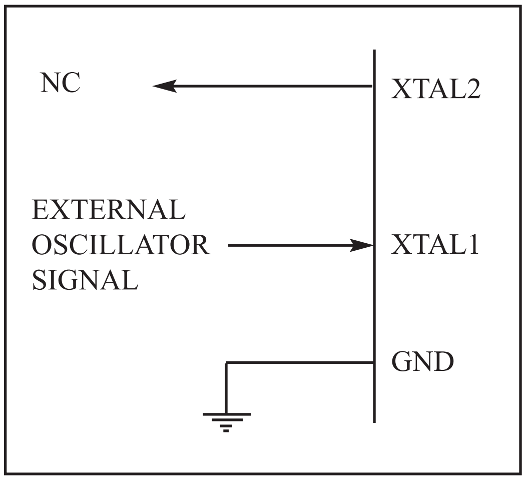


Figure 8-6a. XTAL1 Connection to an External Clock Source

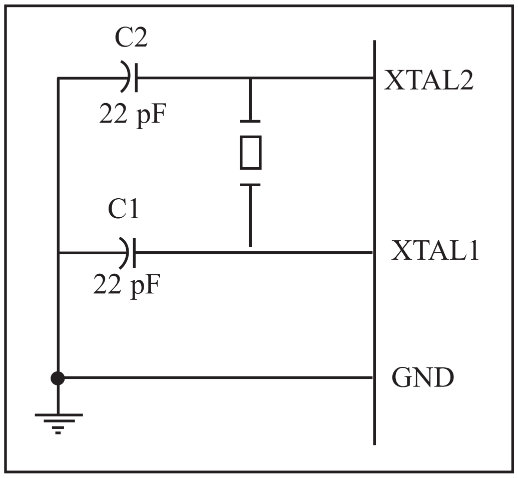


Figure 8-6b. XTAL1–XTAL2 Connection to Crystal Oscillator

The most widely used option is to connect the XTAL1 and XTAL2 pins to a crystal (or ceramic) oscillator, as shown in Figure 8-6b. In this mode, when CKOPT is programmed, the oscillator output will oscillate with a full rail-to-rail swing on the output, causing a more powerful clock signal. This is suitable when the chip drives a second clock buffer or operates in a very noisy environment. As you see in Table 8-10, this mode has a wide frequency range. When CKOPT is not programmed, the oscillator has a smaller output swing and a limited frequency range. This mode cannot be used to drive other clock buffers, but it does reduce power consumption considerably. There are four choices for the crystal oscillator option. Table 8-10 shows all of these choices. Notice that mode 101 cannot be used with crystals, and only ceramic resonators can be used. Example 8-1 shows the relation between crystal frequency and instruction cycle time.

Table 8-10: ATmega32 Crystal Oscillator Frequency Choices and Capacitor Range

CKOPT	CKSEL3...1	Frequency (MHz)	C1 and C2 (pF)
1	101	0.4–0.9	Not for crystals
1	110	0.9–3.0	12–22
1	111	3.0–8.0	12–22
0	101, 110, 111	More than 1.0	12–22

Example 8-1

Find the instruction cycle time for the ATmega32 chip with the following crystal oscillators connected to the XTAL1 and XTAL2 pins.
(a) 4 MHz (b) 8 MHz (c) 10 MHz

Solution:
(a) Instruction cycle time is 1/(4 MHz) = 250 ns
(b) Instruction cycle time is 1/(8 MHz) = 125 ns
(c) Instruction cycle time is 1/(10 MHz) = 100 ns

Fuse bits and reset delay

The most difficult time for a system is during power-up. The CPU needs both a stable clock source and a stable voltage level to function properly. In AVRs, after all reset sources have gone inactive, a delay counter is activated to make the reset longer. This short delay allows the power to become stable before normal operation starts. You can choose the delay time through the SUT1, SUT0, and CKSEL0 fuses. Table 8-11 shows start-up times for the different values of SUT1, SUT0, and CKSEL fuse bits and also the recommended usage of each combination. Notice that the third column of Table 8-11 shows start-up time from power-down mode. Power-down mode is not discussed in this book.

Brown-out detector

Occasionally, the power source provided to the V_{CC} pin fluctuates, causing the CPU to malfunction. The ATmega family has a provision for this, called *brown-out detection*. The BOD circuit compares VCC with BOD-Level and resets the chip if VCC falls below the BOD-Level. The BOD-Level can be either 2.7 V when the BODLEVEL fuse bit is one (not programmed) or 4.0 V when the BODLEVEL fuse is zero (programmed). You can enable the BOD circuit by programming the BODEN fuse bit. When VCC increases above the trigger level, the BOD circuit releases the reset, and the MCU starts working after the time-out period has expired.

A good rule of thumb

There is a good rule of thumb for selecting the values of fuse bits. If you are using an external crystal with a frequency of more than 1 MHz you can set the CKSEL3, CKSEL2, CKSEL1, SUT1, and SUT0 bits to 1 (not programmed) and clear CKOPT to 0 (programmed).

Table 8-11: Startup Time for Crystal Oscillator and Recommended Usage

CKSEL0	SUT1...0	Start-Up Time from Power-Down	Delay from Reset (VCC = 5)	Recommended Usage
0	00	258 CK	4.1	Ceramic resonator, fast rising power
0	01	258 CK	65	Ceramic resonator, slowly rising power
0	10	1K CK	-	Ceramic resonator, BOD enabled
0	11	1K CK	4.1	Ceramic resonator, fast rising power
1	00	1K CK	65	Ceramic resonator, slowly rising power
1	01	16K CK	-	Crystal oscillator, BOD enabled
1	10	16K CK	4.1	Crystal oscillator, fast rising power
1	11	16K CK	65	Crystal oscillator, slowly rising power

Putting it all together

Many of the programs we showed in the first seven chapters were intended to be simulated. Now that we know what we should write in the fuse bits and how we should connect the ATmega32 pins, we can download the hex output file provided by the AVR Studio assembler into the Flash memory of the AVR chip using an AVR programmer.

We can use the following skeleton source code for the programs that we intend to download into a chip. Notice that you have to modify the first line if you use a chip other than ATmega32. As you can see in the comments, if you want to enable interrupts you have to modify "`.ORG 0`", and if you do not use call the instruction in your code, you can omit the codes that set the stack pointer.

```
.INCLUDE "M32DEF.INC"   ;change it according to your chip
.ORG 0                  ;change it if you use interrupt
LDI   R16,HIGH(RAMEND)  ;set the high byte of stack pointer to
OUT   SPH,R16           ;the high address of RAMEND
LDI   R16,LOW(RAMEND)   ;set the low byte of stack pointer to
OUT   SPL,R16           ;low address of RAMEND

...                     ;place your code here
```

As an example, examine Program 8-1. It will toggle all the bits of Port B with some delay between the "on" and "off" states.

```
;Test Program 8-1: Toggling PORTB for the Atmega32
.INCLUDE "M32DEF.INC"         ;using Atmega32
.ORG 0
      LDI   R16,HIGH(RAMEND)  ;set up stack
      OUT   SPH,R16
      LDI   R16,LOW(RAMEND)
      OUT   SPL,R16
      LDI   R16,0xFF          ;load R16 with 0xFF
      OUT   DDRB,R16          ;Port B is output
BACK:
      COM   R16               ;complement R16
      OUT   PORTB,R16         ;send it to Port B
      CALL  DELAY             ;time delay
      RJMP  BACK              ;keep doing this indefinitely

DELAY:
      LDI   R20,16
L1:   LDI   R21,200
L2:   LDI   R22,250
L3:
      NOP
      NOP
      DEC   R22
      BRNE  L3
      DEC   R21
      BRNE  L2

      DEC   R20
      BRNE  L1
      RET
```

Program 8-1: Toggling Port B in Assembly

Toggle program in C

In Chapter 7 we covered C programming of the AVR using the AVR GCC compiler. Program 8-2 shows the toggle program written in C. It will toggle all the bits of Port B with some delay between the "on" and "off" states.

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

void delay_ms(int d);

int main(void)
{
      DDRB = 0xFF;                     //Port B is output
      while (1)
      {                                //do forever
            PORTB = 0x55;
            delay_ms(1000);            //delay 1 second
            PORTB = 0xAA;
            delay_ms(1000);            //delay 1 second
      }
      return 0;
}

void delay_ms(int d)
{
      _delay_ms(d);                    //delay 1000 us
}
```

Program 8-2: Toggling Port B in C

Review Questions

1. A given ATmega32-based system has a crystal frequency of 16 MHz. What is the instruction cycle time for the CPU?
2. How many fuse bytes are available in ATmega32?
3. True or false. Upon power-up, both voltage and frequency are stable instantly.
4. The internal RC oscilator works for the frequency range of _____ to ______ MHz.
5. Which fuse bit is used to disable the BOD?
6. True or false. Upon power-up, the CPU starts working immediately.
7. What is the rule of thumb for ATmega32 fuse bits?
8. The brown-out detection voltage can be set at _______ or ______ by________ fuse bit.
9. True or false. The higher the clock frequency for the system, the lower the power dissipation.

SECTION 8.3: EXPLAINING THE HEX FILE FOR AVR

Intel Hex is a widely used file format designed to standardize the loading (transferring) of executable machine code into a chip. Therefore, the loaders that come with every ROM burner (programmer) support the Intel Hex file format. In many Windows-based assemblers such as AVR Studio, the Intel Hex file is produced according to the settings you set. In the AVR Studio environment, the object file is fed into the linker program to produce the Intel hex file. The hex file is used by a programmer such as the AVRISP to transfer (load) the file into the Flash memory. The AVR Studio assembler can produce three types of hex files. They are (a) Intel Intellec 8/MDS (Intel Hex), (b) Motorola S-record, and (c) Generic. See Table 8-12. In this section we will explain Intel Hex with some examples. We recommend that you do not use AVR GCC if you want to test the programs in this section on your computer. It is better to use a simple .asm file like toggle.asm to understand this concept better.

Table 8-12: Intel Hex File Formats Produced by AVR Studio

Format Name	File Extension	Max. ROM Address
Extended Intel Hex file	.hex	20-bit address
Motorola S-record	.mot	32-bit address
Generic	.gen	24-bit address

Analyzing the Intel Hex file

We choose the hex type of Intel Hex, Motorola S-record, or Generic by using the command-line invocation options or setting the options in the AVR Studio assembler itself. If we do not choose one, the AVR Studio assembler selects Intel Hex by default. Intel Hex supports up to 16-bit addressing and is not applicable for programs more than 64K bytes in size. To overcome this limitation AVR Studio uses extended Intel Hex files, which support type 02 records to extend address space to 1M. We will explain extended Intel Hex file format in this section. Figure 8-10 shows the Intel Hex file of the test program whose list file is given in Figure 8-8. Since the programmer (loader) uses the Hex file to download the opcode into Flash, the hex file must provide the following: (1) the number of bytes of information to be loaded, (2) the information itself, and (3) the starting address where the information must be placed. Each record (line) of the Hex file consists of six parts as follows:

:BBAAAATTHHHHH.......HHHHCC

The following describes each part:

1. ":" Each line starts with a colon.
2. BB, the count byte. This tells the loader how many bytes are in the line.
3. AAAA is for the record address. This is a 16-bit address. The loader places the first byte of record data into this Flash location. This is the case in files that are less than 64 KB. For files that are more than 64 KB the address field shows the record address in the current segment.

4. TT is for type. This field is 00, 01, or 02. If it is 00, it means that there are more lines to come after this line. If it is 01, it means that this is the last line and the loading should stop after this line. If it is 02, it indicates the current segment address. To calculate the absolute address of each record (line), we have to shift the current segment address 4 bits to left and then add it to the record address. Examples 8-2 and 8-3 show how to calculate the absolute address of a record in extended Intel hex file.
5. HH......H is the real information (data or code). The loader places this information into successive memory locations of Flash. The information in this field is presented as low byte followed by the high byte.
6. CC is a single byte. This last byte is the checksum byte for everything in that line. The checksum byte is used for error checking. Checksum bytes are discussed in detail in Chapters 6 and 7. Notice that the checksum byte at the end of each line represents the checksum byte for everything in that line, and not just for the data portion.

Example 8-2

What is the absolute address of the first byte of a record that has 0025 in the address field if the last type 02 record before it has the segment address 0030?

Solution:

To calculate the absolute address of each record (line), we have to shift the segment address (0030) four bits to the left and then add it to the record address (0025):

```
0030 (2 bytes segment address) shifted 4 bits to the left   -->      00300
0025 (record address)                                              +    25
                                                                   ---------
=>    (absolute address)                                             00325
```

Example 8-3

What is the absolute address of the first byte of the second record below?

```
:02**0000**020000FC
:10**0000**0008E00EBF0FE50DBF0FEF07BB05E500953C
```

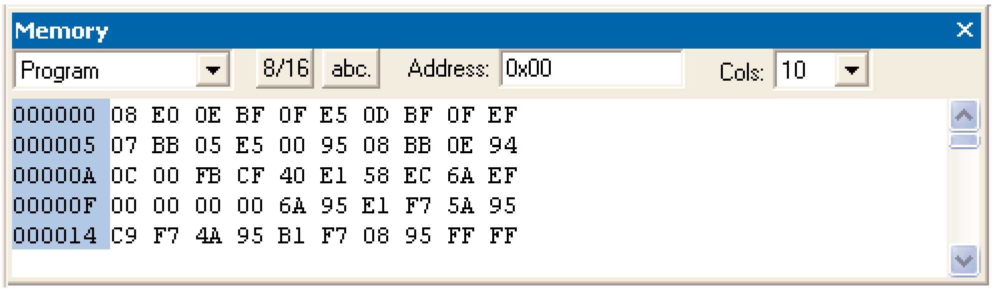
Solution:

To calculate the absolute address of the first byte of the second record, we have to shift left the segment address (0000, as you see in the first record) four bits and then add it to the second record address (0000, as you see in the second record).

```
0000 (segment address) shift 4 bits to the left   -->   00000
                                                      + 0000      (record address)
                                                      ---------
                                                      000000      (absolute address)
```

Analyzing the bytes in the Flash memory vs. list file

The data in the Flash memory of the AVR is recorded in a way that is called *Little-endian*. This means that the high byte of the code is located in the higher address location of Flash memory, and the low byte of the code is located in the lower address location of Flash memory. Compare the first word of code (`e008`) in Figure 8-8 with the first two bytes of Flash memory (`08e0`) in Figure 8-7. As you see, 08, which is the low byte of the first instruction (`LDI R16,HIGH(RAMEND)`) in the code, is placed in the lower location of Flash memory, and e0, which is the high byte of the instruction in the code, is placed in the next location of program space just after 08.



```
000000 08 E0 0E BF 0F E5 0D BF 0F EF
000005 07 BB 05 E5 00 95 08 BB 0E 94
00000A 0C 00 FB CF 40 E1 58 EC 6A EF
00000F 00 00 00 00 6A 95 E1 F7 5A 95
000014 C9 F7 4A 95 B1 F7 08 95 FF FF
```

Figure 8-7. AVR Flash Memory Contents

```
LOC    OBJ            LINE
                      .ORG 0x000
000000 e008                 LDI    R16,HIGH(RAMEND)
000001 bf0e                 OUT    SPH,R16
000002 e50f                 LDI    R16,LOW(RAMEND)
000003 bf0d                 OUT    SPL,R16

000004 ef0f                 LDI    R16,0xFF
000005 bb07                 OUT DDRB,R16
000006 e505                 LDI    R16,0x55
                      BACK:
000007 9500                 COM          R16
000008 bb08                 OUT    PORTB,R16
000009 940e 000c            CALL   DELAY_1S
00000b cffb                 RJMP   BACK
                      DELAY_1S:
00000c e140                 LDI    R20,16
00000d ec58           L1:   LDI    R21,200
00000e ef6a           L2:   LDI    R22,250
                      L3:
00000f 0000                 NOP
000010 0000                 NOP
000011 956a                 DEC    R22
000012 f7e1                 BRNE   L3
000013 955a                 DEC    R21
000014 f7c9                 BRNE   L2
000015 954a                 DEC    R20
000016 f7b1                 BRNE   L1
000017 9508                 RET
```

Figure 8-8. List File for Test Program
(Comments and other lines are deleted, and some spaces are added for simplicity.)

As we mentioned in Chapter 2, each Flash location in the AVR is 2 bytes long. So, for example, the first byte of Flash location #2 is Byte #4 of the code. See Figure 8-9.

Flash Memory

Location #0	Byte #0	Byte #1
Location #1	Byte #2	Byte #3
Location #2	Byte #4	Byte #5
Location #3	Byte #6	Byte #7

Figure 8-9. AVR Flash Memory Locations

In Figure 8-10 you see the hex file of the toggle code. The first record (line) is a type 02 record and indicates the current segment address, which is `0000`. The next record (line) is a type 00 record and contains the data (the code to be loaded into the chip). After ':' the record starts with `10`, which means that the data field contains 10 (16 decimal) bytes of data. The next field is the address field (`0000`), and it indicates that the first byte of the data field will be placed in address location 0 in the current segment. So the first byte of code will be loaded into location 0 of Flash memory. (Reexamine Example 8-3 if needed.) Also, notice the use of `.ORG 0x000` in the code. The next field is the data field, which contains the code to be loaded into the chip. The first byte of the data field is `08`, which is the low byte of the first instruction (`LDI R16,HIGH(RAMEND)`). See Figure 8-8. The last field of the record is the checksum byte of the record. Notice that the checksum byte at the end of each line represents the checksum byte for everything in that line, and not just for the data portion.

Pay attention to the address field of the next record (`0010`) in Figure 8-10 and compare it with the address of the `bb08` instruction in the list file in Figure 8-8. As you can see, the address in the list file is `000008`, which is exactly half of the address of the `bb08` instruction in the hex file, which is `0010`. That is because each Flash location (word) contains 2 bytes.

```
:020000020000FC
:1000000008E00EBF0FE50DBF0FEF07BB05E500953C
:1000100008BB0E940C00FBCF40E158EC6AEF0000E7
:1000200000006A95E1F75A95C9F74A95B1F7089526
:00000001FF

Separating the fields, we get the following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH          CC
:02 0000 02 0000                                      FC
:10 0000 00 08E00EBF0FE50DBF0FEF07BB05E50095          3C
:10 0010 00 08BB0E940C00FBCF40E158EC6AEF0000          E7
:10 0020 00 00006A95E1F75A95C9F74A95B1F70895          26
:00 0000 01                                           FF
```

Figure 8-10. Intel Hex File Test Program with the Intel Hex Option

Examine Examples 8-4 through 8-6 to gain insight into the Intel Hex file format.

Example 8-4

From Figure 8-10, analyze the six parts of line 3.

Solution:

After the colon (:), we have 10, which means that 16 bytes of data are in this line. 0010H is the record address, and means that 08, which is the first byte of the record, is placed in address location 10H (16 decimal). Next, 00 means that this is not the last line of the record. Then the data, which is 16 bytes, is as follows: `08BB0E940C00FBCF40E158EC6AEF0000`. Finally, the last byte, E7, is the checksum byte.

Example 8-5

Compare the data portion of the Intel Hex file of Figure 8-10 with the opcodes in the list file of the test program given in Figure 8-8. Do they match?

Solution:

In the second line of Figure 8-10, the data portion starts with 08E0H, where the low byte is followed by the high byte. That means it is E008, the opcode for the instruction "`LDI   R16,HIGH(RAMEND)`", as shown in the list file of Figure 8-8. The last byte of the data in line 5 is 0895, which is the opcode for the "`RET`" instruction in the list file.

Example 8-6

(a) Verify the checksum byte for line 3 of Figure 8-10. (b) Verify also that the information is not corrupted.

Solution:

(a) `10 + 00 + 00 + 00 + 08 + E0 + 0E + BF + 0F + E5 + 0D + BF + 0F + EF + 07 + BB + 05 + E5 + 00 + 95 = 6C4` in hex. Dropping the carries (6) gives C4H, and its 2's complement is 3CH, which is the last byte of line 3.

(b) If we add all the information in line 2, including the checksum byte, and drop the carries we should get `10 + 00 + 00 + 00 + 08 + E0 + 0E + BF + 0F + E5 + 0D + BF + 0F + EF + 07 + BB + 05 + E5 + 00 + 95 + 3C = 700.` Dropping the carries (7) gives 00H, which means OK.

Review Questions

1. True or false. The Intel Hex file format does not use the checksum byte method to ensure data integrity.
2. The first byte of a line in an Intel Hex file represents ____.
3. The last byte of a line in an Intel Hex file represents ____.
4. In the TT field of an Intel Hex file, we have 00. What does it indicate?
5. Find the checksum byte for the following values: 22H, 76H, 5FH, 8CH, 99H.
6. In Question 5, add all the values and the checksum byte. What do you get?

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

In this section, we show various ways of loading a hex file into the AVR microcontroller. We also discuss the connection for a simple AVR trainer.

Atmel has skillfully designed AVR microcontrollers for maximum flexibility of loading programs. The three primary ways to load a program are:

1. Parallel programming. In this way a device burner loads the program into the microcontroller separate from the system. This is useful on a manufacturing floor where a gang programmer is used to program many chips at one time. Most mainstream device burners support the AVR families: EETools is a popular one. The device programming method is straightforward: The chip is programmed before it is inserted into the circuit. Or, the chip can be removed and reprogrammed if it is in a socket. A ZIF (zero insertion force) socket is even quicker and less damaging than a standard socket. When removing and reinserting, we must observe ESD (electrostatic discharge) procedures. Although AVR devices are rugged, there is always a risk when handling them. Using this method allows all of the device's resources to be utilized in the design. No pins are shared, nor are internal resources of the chip used as is the case in the other two methods. This allows the embedded designer to use the minimum board space for the design.
2. An in-circuit serial programmer (ISP) allows the developer to program and debug their microcontroller while it is in the system. This is done by a few wires with a system setup to accept this configuration. In-circuit serial programming is excellent for designs that change or require periodic updating. AVR has two methods of ISP. They are SPI and JTAG. Most of the ATmega family supports both methods. The SPI uses 3 pins, one for send, one for receive, and one for clock. These pins can be used as I/O after the device is programmed. The designer must make sure that these pins do not conflict with the programmer. Notice that SPI stands for "serial peripheral interface" and is a protocol. But ISP stands for "in-circuit serial programming" and is a method of code loading. AVRISP and many other devices support ISP. To connect AVRISP to your device you also need to connect VCC, GND, and RESET pins. You must bring the pins to a header on the board so that the programmer can connect to it. Figure 8-11 shows the pin connections.

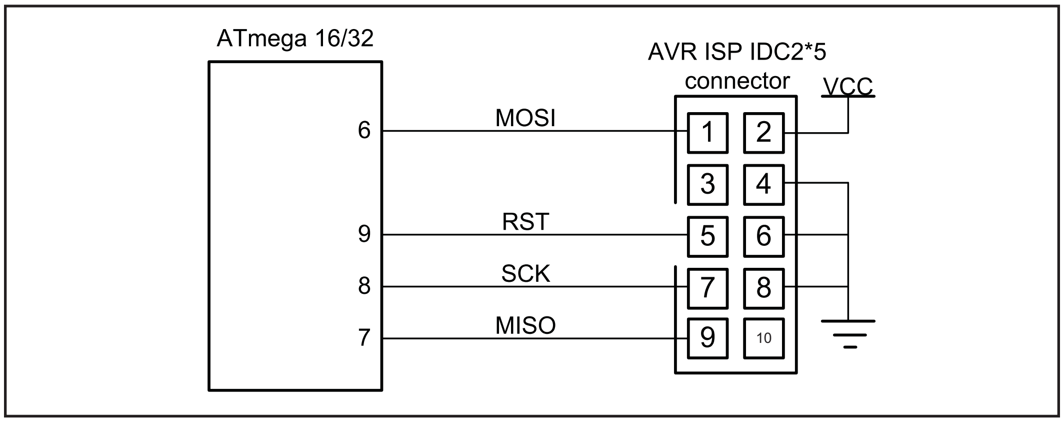


Figure 8-11. ISP 10-pin Connections (See www.Atmel.com for 6-pin version)

Another method of ISP is JTAG. JTAG is another protocol that supports in-circuit programming and debugging. It means that in addition to programming you can trace your program on the chip line by line and watch or change the values of memory locations, ports, or registers while your program is running on the chip.

3. A boot loader is a piece of code burned into the microcontroller's program Flash. Its purpose is to communicate with the user's board to load the program. A boot loader can be written to communicate via a serial port, a CAN port, a USB port, or even a network connection. A boot loader can also be designed to debug a system, similar to the JTAG. This method of programming is excellent for the developer who does not always have a device programmer or a JTAG available. There are several application notes on writing boot loaders on the Web. The main drawback of the boot loader is that it does require a communication port and program code space on the microcontroller. Also, the boot loader has to be programmed into the device before it can be used, usually by one of the two previous ways.

   The boot loader method is ideal for the developer who needs to quickly program and test code. This method also allows the update of devices in the field without the need of a programmer. All one needs is a computer with a port that is compatible with the board. (The serial port is one of the most commonly used and discussed, but a CAN or USB boot loader can also be written.) This method also consumes the largest amount of resources. Code space must be reserved and protected, and external devices are needed to connect and communicate with the PC. Developing projects using this method really helps programmers test their code. For mature designs that do not change, the other two methods are better suited.

AVR trainers

There are many popular trainers for the AVR chip. The vast majority of them have a built-in ISP programmer. See the following website for more information and support about the AVR trainers. For more information about how to use an AVR trainer you can visit the www.MicroDigitalEd.com website.

Review Questions

1. Which method(s) to program the AVR microcontroller is/are the best for the manufacturing of large-scale boards?
2. Which method(s) allow(s) for debugging a system?
3. Which method(s) would allow a small company to develop a prototype and test an embedded system for a variety of customers?
4. True or false. The ATmega32 has Flash program ROM.
5. Which pin is used for reset in the ATmega32?
6. What is the status of the RESET pin when it is not activated?

The information about the trainer board can be found at:
www.MicroDigitalEd.com

SUMMARY

This chapter began by describing the function of each pin of the ATmega32. A simple connection for ATmega32 was shown. Then, the fuse bytes were discussed. We use fuse bytes to enable features such as BOD and clock source and frequency. We also explained the Intel Hex file format and discussed each part of a record in a hex file using an example. Then, we explained list files in detail. The various ways of loading a hex file into a chip were discussed in the last section. The connections to a ISP device were shown.

PROBLEMS

SECTION 8.2: AVR FUSE BITS

17. How many clock sources does the AVR have?
18. What fuse bits are used to select clock source?
19. Which clock source do you suggest if you need a variable clock source?
20. Which clock source do you suggest if you need to build a system with minimum external hardware?
21. Which clock source do you suggest if you need a precise clock source?
22. How many fuse bytes are there in the AVR?

23. Which fuse bit is used to set the brown-out detection voltage for the ATmega32?
24. Which fuse bit is used to enable and disable the brown-out detection voltage for the ATmega32?
25. If the brown-out detection voltage is set to 4.0 V, what does it mean to the system?

SECTION 8.3: EXPLAINING THE INTEL HEX FILE FOR AVR

26. True or false. The Hex option can be set in AVR Studio.
27. True or false. The extended Intel Hex file can be used for ROM sizes of less than 64 kilobytes.
28. True or false. The extended Intel Hex file can be used for ROM sizes of more than 64 kilobytes.
29. Analyze the six parts of line 3 of Figure 8-10.
30. Verify the checksum byte for line 3 of Figure 8-10. Verify also that the information is not corrupted.
31. What is the difference between Intel Hex files and extended Intel Hex files?

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

32. True or false. To use a parallel programmer, we must remove the AVR chip from the system and place it into the programmer.
33. True or false. ISP can work only with Flash chips.

34. What are the different ways of loading a code into an AVR chip?
35. True or false. A boot loader is a kind of parallel programmer.

ANSWERS TO REVIEW QUESTIONS

SECTION 8.2: AVR FUSE BITS

1. 1/16 MHz = 62.5 ns
2. 16 bits = 2 bytes
3. False
4. 1, 8
5. BODEN
6. False
7. If you are using an external crystal with a frequency of more than 1 MHz you can set the CKSEL3, CKSEL2, CKSEL1, SUT1, and SUT0 bits to 1 (not programmed) and clear CKOPT to 0 (programmed).
8. 2.7 V, 4 V, BODLEVEL
9. False

SECTION 8.3: EXPLAINING THE INTEL HEX FILE FOR AVR

1. False
2. The number of bytes of data in the line
3. The checksum byte of all the bytes in that line
4. 00 means this is not the last line and that more lines of data follow.
5. 22H + 76H + 5FH + 8CH + 99H = 21CH. Dropping the carries we have 1CH and its 2's complement, which is E4H.
6. 22H + 76H + 5FH + 8CH + 99H + E4H = 300H. Dropping the carries, we have 00, which means that the data is not corrupted.

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

1 Device burner
2. JTAG and boot loader
3. ISP
4. True
5. Pin 9
6. HIGH

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.

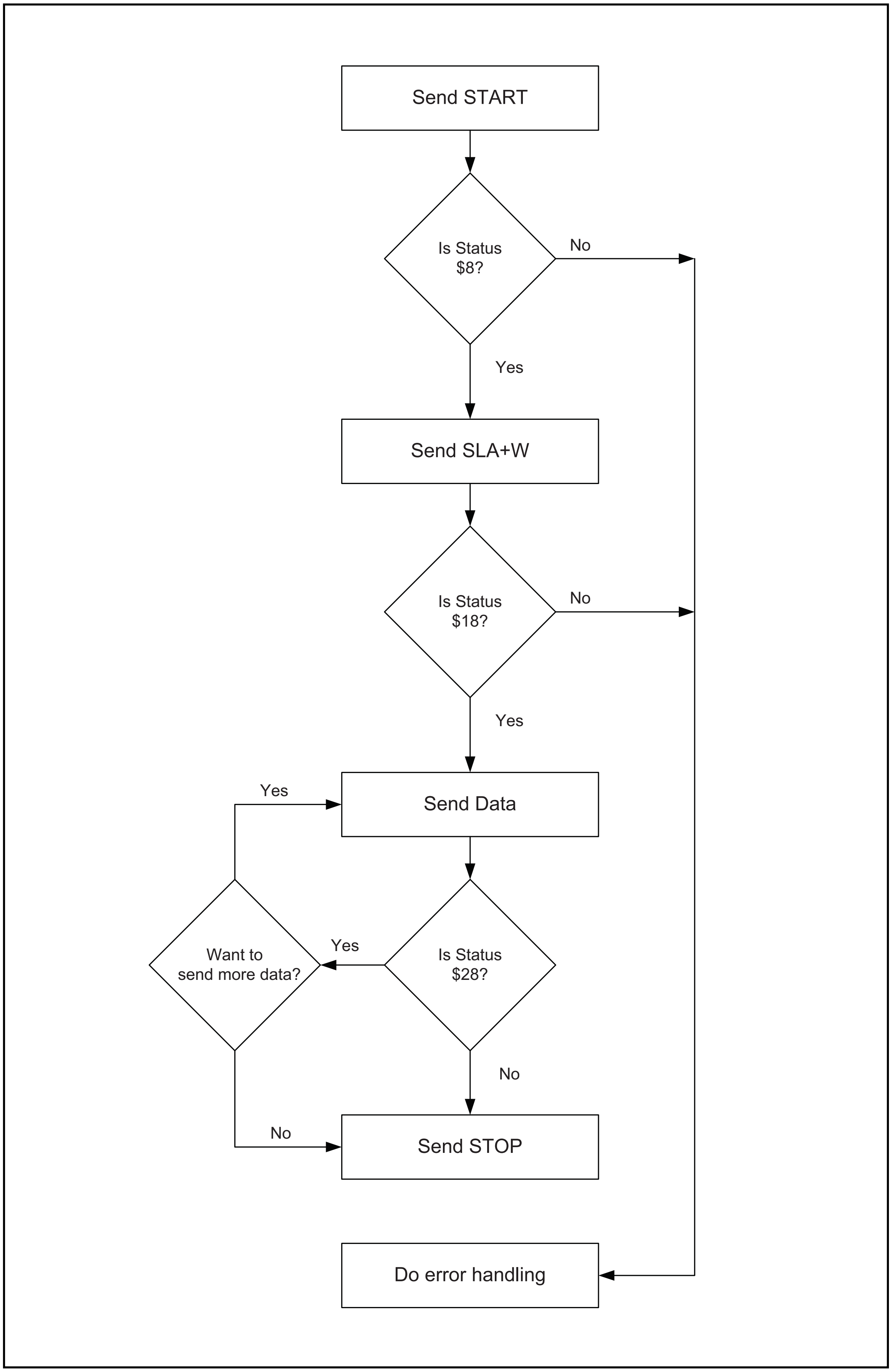


Figure 18-18. Programming Steps of Master Transmitter Mode with Checking of Flags

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

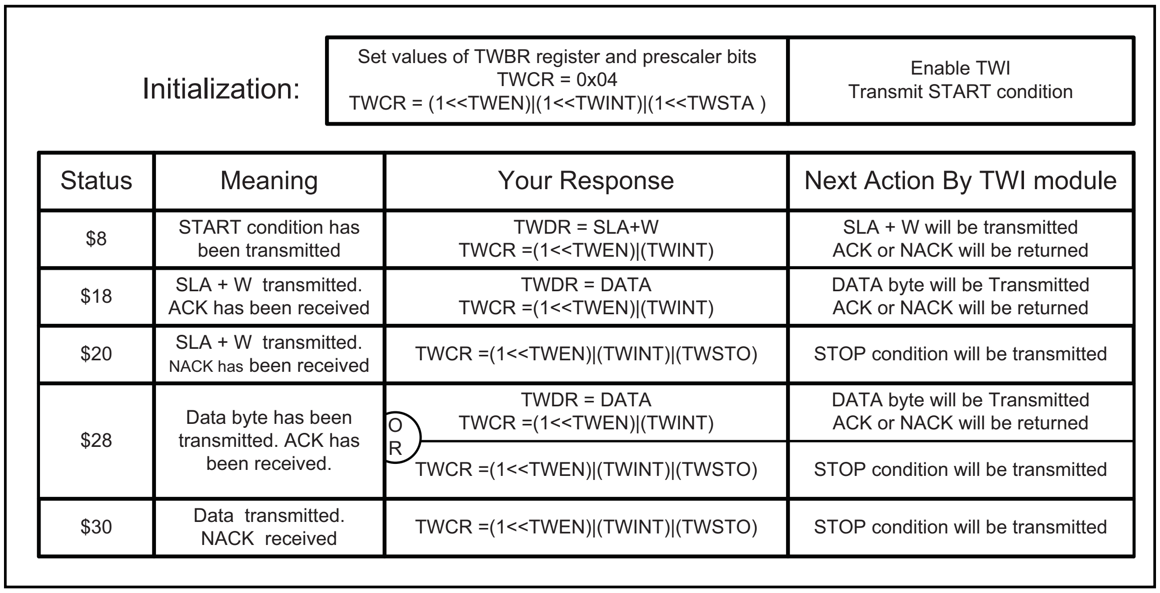
To send data, after transmitting of SLA + W, we should do the following steps:

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)
			DATA byte will be Transmitted
			ACK or NACK will be returned

		OR	
		TWCR =(1<<TWEN)|(TWINT)|(TWSTO)	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, 0x48        ;move 0x48 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 0
      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 0
      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=0x48;                //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-

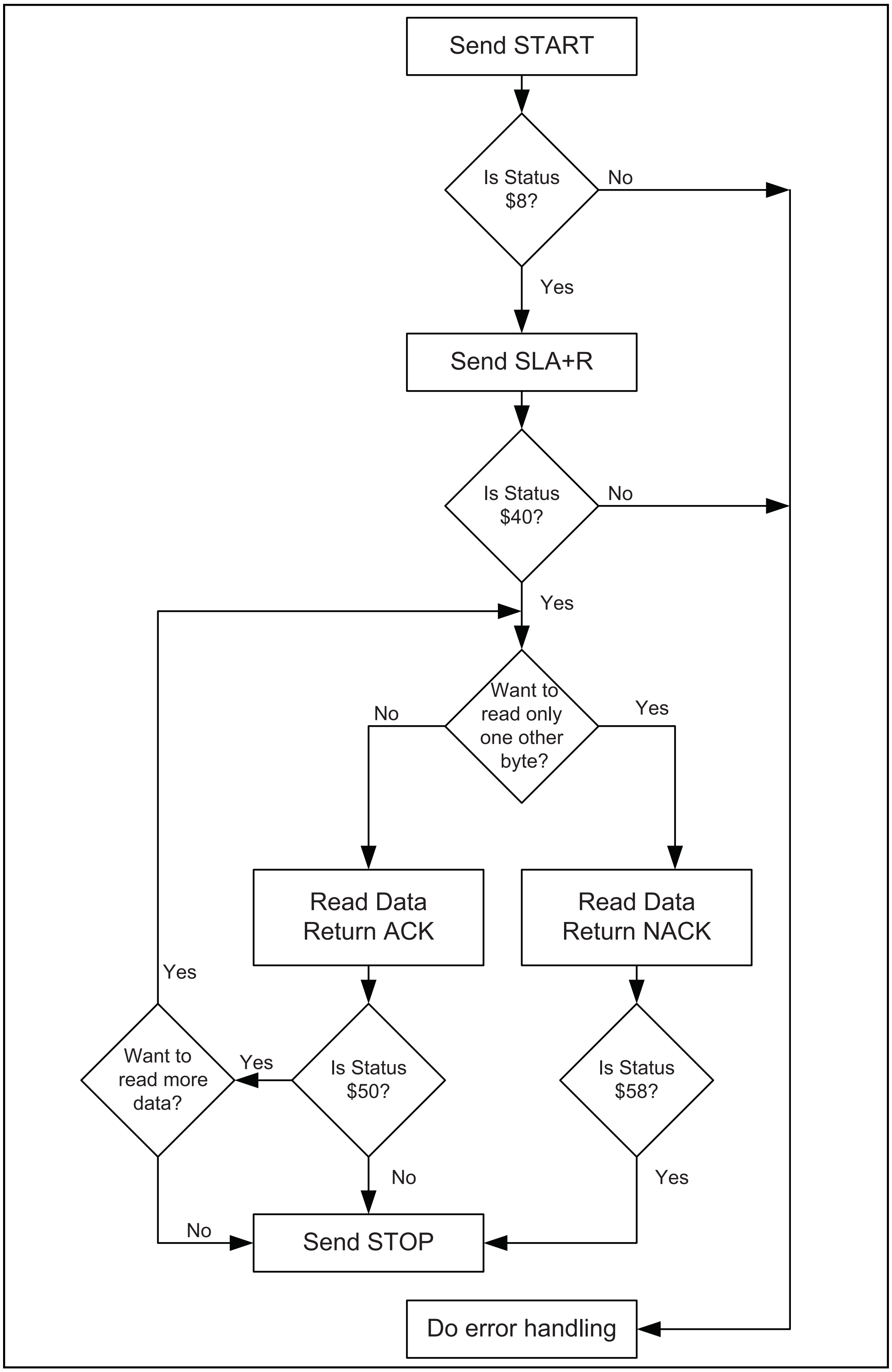


Figure 18-20. TWI Programming Steps of Master Receiver Mode with Checking of Flags

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

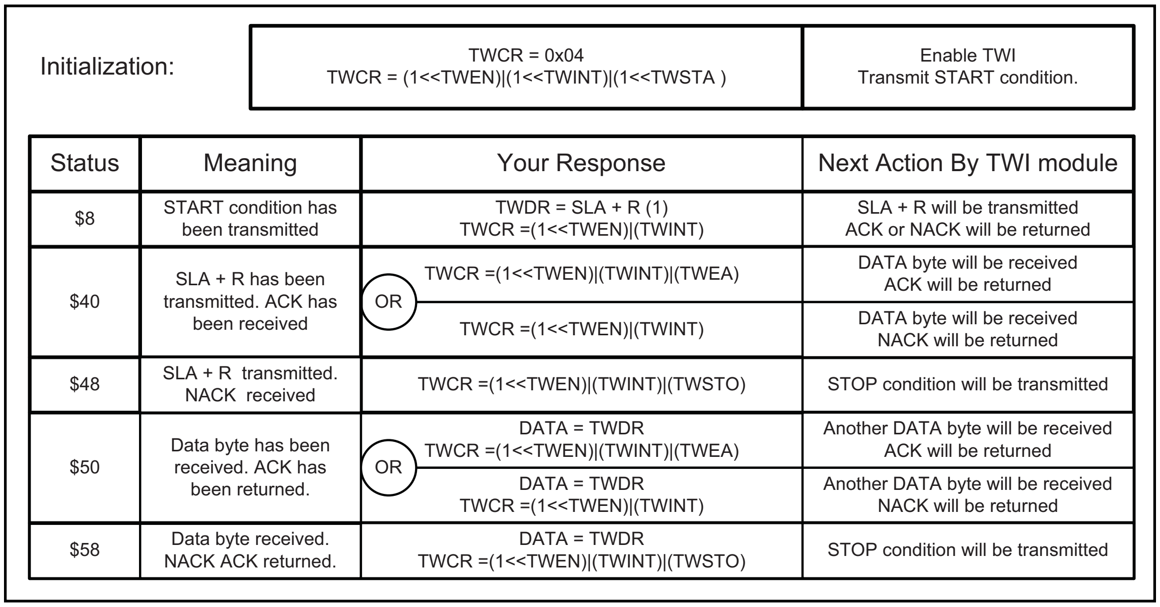
If we want to receive more than one byte of data, we should receive data and return ACK by doing the following steps:

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, 0x48        ;move 0x48 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 0
     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 0
      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=0x48;              //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

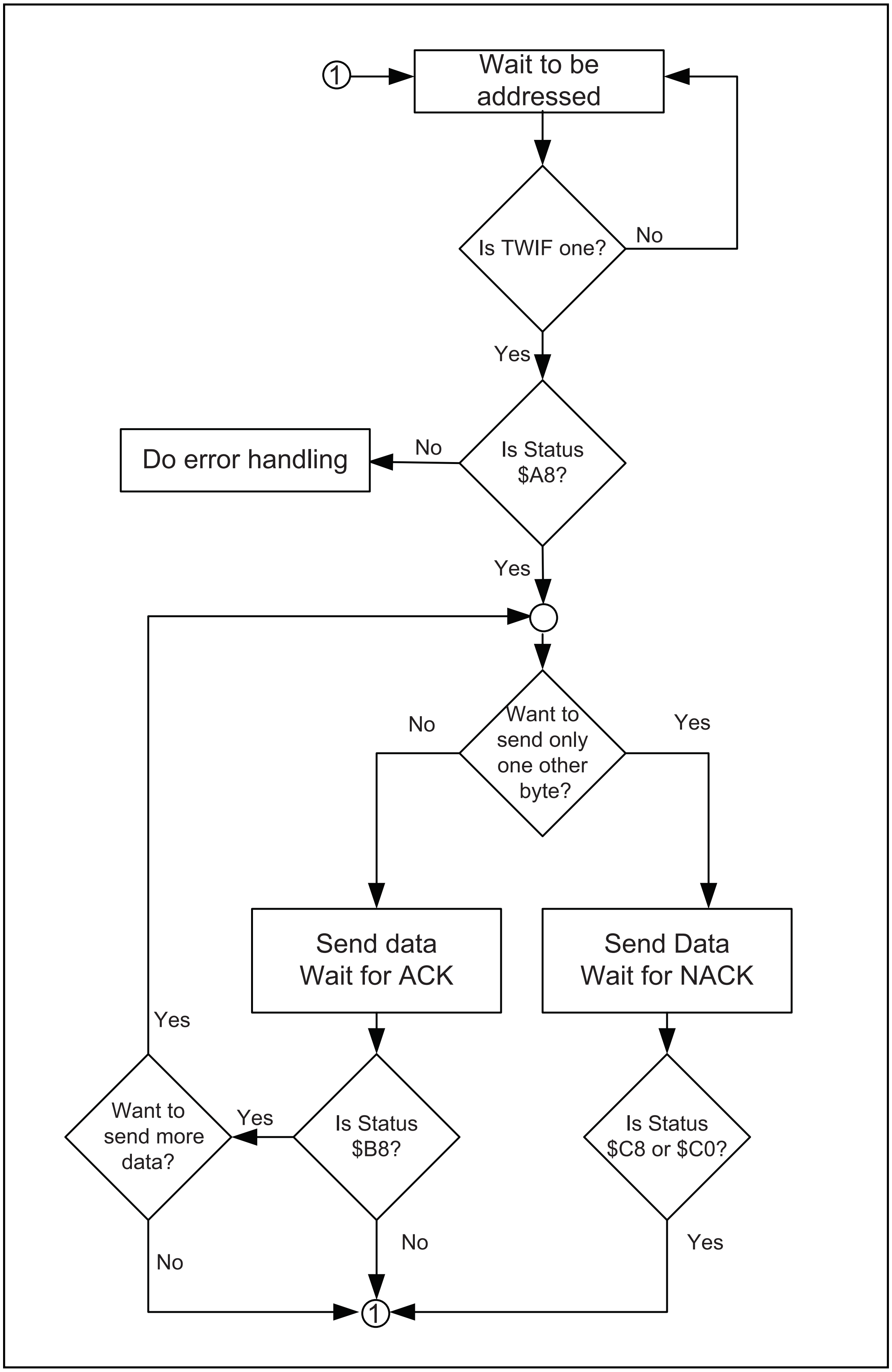


Figure 18-22. TWI Programming Steps of Slave Transmitter Mode with Checking of Flags

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	OR	TWDR = DATA
			TWCR =(1<<TWEN)|(TWINT)|(TWEA)
				DATA byte will be transmitted
				Wait for ACK

			TWDR = DATA
			TWCR =(1<<TWEN)|(TWINT)
				DATA byte will be transmitted
				Wait for NACK

$B8	Data has been transmitted ACK received	OR	TWDR = DATA
			TWCR =(1<<TWEN)|(TWINT)|(TWEA)
				DATA byte will be transmitted
				Wait for ACK

			TWDR = DATA
			TWCR =(1<<TWEN)|(TWINT)
				DATA byte will be transmitted
				Wait for NACK

$C0	Data has been transmitted NACK received	OR	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Start from beginning and wait to be addressed
			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)	OR	TWCR =(1<<TWEN)|(TWINT)|(TWEA)	Start from beginning and wait to be addressed
			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.

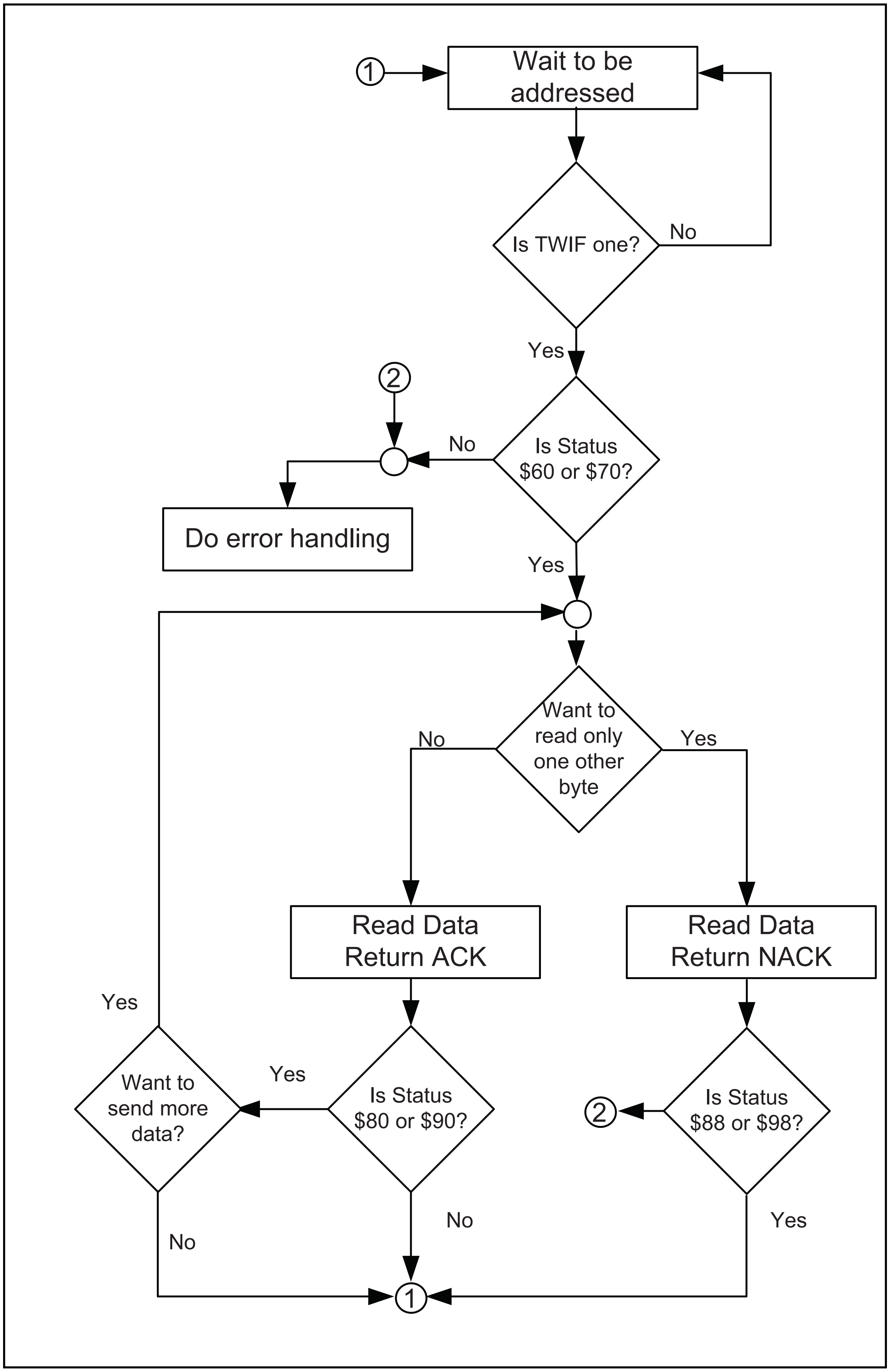


Figure 18-24. TWI Programming Steps of Slave Receiver Mode with Checking of Flags

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?

APPENDIX A

AVR INSTRUCTIONS EXPLAINED

OVERVIEW

In this appendix, we describe each intruction of the ATmega32. In many cases, a simple code example is given to clarify the instruction.

At the end there is a table that shows all the registers and their bits.

SECTION A.1: INSTRUCTION SUMMARY

DATA TRANSFER INSTRUCTIONS

Mnemonics	Operands	Description	Operation	Flags
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None
MOVW	Rd, Rr	Copy Register Word	Rd + 1:Rd ← Rr + 1:Rr	None
LDI	Rd, K	Load Immediate	Rd ← K	None
LD	Rd, X	Load Indirect	Rd ← (X)	None
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None
LD	Rd, –X	Load Indirect and Pre-Dec.	X ← X – 1, Rd ← (X)	None
LD	Rd, Y	Load Indirect	Rd ← (Y)	None
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None
LD	Rd, –Y	Load Indirect and Pre-Dec.	Y ← Y – 1, Rd ← (Y)	None
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None
LD	Rd, Z	Load Indirect	Rd ← (Z)	None
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None
LD	Rd, –Z	Load Indirect and Pre-Dec.	Z ← Z – 1, Rd ← (Z)	None
LDD	Rd, Z + q	Load Indirect with Displacement	Rd ← (Z + q)	None
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None
ST	X, Rr	Store Indirect	(X) ← Rr	None
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None
ST	–X, Rr	Store Indirect and Pre-Dec.	X ← X – 1, (X) ← Rr	None
ST	Y, Rr	Store Indirect	(Y) ← Rr	None
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None
ST	–Y, Rr	Store Indirect and Pre-Dec.	Y ← Y – 1, (Y) ← Rr	None
STD	Y + q, Rr	Store Indirect with Displacement	(Y + q) ← Rr	None
ST	Z, Rr	Store Indirect	(Z) ← Rr	None
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None
ST	–Z, Rr	Store Indirect and Pre-Dec.	Z ← Z – 1, (Z) ← Rr	None
STD	Z + q, Rr	Store Indirect with Displacement	(Z + q) ← Rr	None
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None
LPM		Load Program Memory	R0 ← (Z)	None
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None
LPM	Rd, Z+	Load Program Memory and Post-Inc.	Rd ← (Z), Z ← Z+1	None
SPM		Store Program Memory	(Z) ← R1:R0	None
IN	Rd, P	In Port	Rd ← P	None
OUT	P, Rr	Out Port	P ← Rr	None
PUSH	Rr	Push Register on Stack	Stack ← Rr	None
POP	Rd	Pop Register from Stack	Rd ← Stack	None

BRANCH INSTRUCTIONS

Mnem.	Oper.	Description	Operation	Flags
RJMP	k	Relative Jump	PC ← PC + k + 1	None
IJMP		Indirect Jump to (Z)	PC ← Z	None
JMP	k	Direct Jump	PC ← k	None
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None
ICALL		Indirect Call to (Z)	PC ← Z	None
CALL	k	Direct Subroutine Call	PC ← k	None
RET		Subroutine Return	PC ← Stack	None
RETI		Interrupt Return	PC ← Stack	I
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None
CP	Rd,Rr	Compare	Rd − Rr	Z,N,V,C,H
CPC	Rd,Rr	Compare with Carry	Rd − Rr − C	Z,N,V,C,H
CPI	Rd,K	Compare Register with Immediate	Rd − K	Z,N,V,C,H
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None
BRBS	s, k	Branch if Status Flag Set	if (SREG(s)=1) then PC←PC+k+1	None
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s)=0) then PC←PC+k+1	None
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None
BRGE	k	Branch if Greater or Equal,Signed	if (N and V= 0) then PC←PC + k +1	None
BRLT	k	Branch if Less Than Zero, Signed	if (N and V= 1) then PC←PC + k +1	None
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None

BIT AND BIT-TEST INSTRUCTIONS

Mnem.	Operan.	Description	Operation	Flags
SBI	P, b	Set Bit in I/O Register	I/O(P, b) ← 1	None
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) ← 0	None
LSL	Rd	Logical Shift Left	Rd(n + 1) ← Rd(n), Rd(0) ← 0	Z,C,N,V
LSR	Rd	Logical Shift Right	Rd(n)←Rd(n+1), Rd(7)←0	Z,C,N,V
ROL	Rd	Rotate Left Through Carry	Rd(0)←C, Rd(n+1)←Rd(n), C←Rd(7)	Z,C,N,V
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C, Rd(n) ← Rd(n + 1), C ← Rd(0)	Z,C,N,V
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n + 1), n = 0..6	Z,C,N,V
SWAP	Rd	Swap Nibbles	Rd(3..0) ← Rd(7..4), Rd(7..4) ← Rd(3..0)	None
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None
SEC		Set Carry	C ← 1	C
CLC		Clear Carry	C ← 0	C
SEN		Set Negative Flag	N ←1	N
CLN		Clear Negative Flag	N ← 0	N
SEZ		Set Zero Flag	Z ←1	Z
CLZ		Clear Zero Flag	Z ← 0	Z
SEI		Global Interrupt Enable	I ← 1	I
CLI		Global Interrupt Disable	I ← 0	I
SES		Set Signed Test Flag	S ← 1	S
CLS		Clear Signed Test Flag	S ← 0	S
SEV		Set Two's Complement Overflow	V ← 1	V
CLV		Clear Two's Complement Overflow	V ← 0	V
SET		Set T in SREG	T ← 1	T
CLT		Clear T in SREG	T ← 0	T
SEH		Set Half Carry Flag in SREG	H ←1	H
CLH		Clear Half Carry Flag in SREG	H ← 0	H

ARITHMETIC AND LOGIC INSTRUCTIONS

Mnem.	Operands	Description	Operation	Flags
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H
ADIW	Rdl, K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd – Rr	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd – K	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd – Rr – C	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd – K – C	Z,C,N,V,H
SBIW	Rdl, K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl – K	Z,C,N,V,S
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd Rr	Z,N,V
COM	Rd	One’s Complement	Rd ← $FF − Rd	Z,C,N,V
NEG	Rd	Two’s Complement	Rd ← $00 − Rd	Z,C,N,V,H
SBR	Rd, K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V
CBR	Rd, K	Clear Bit(s) in Register	Rd ← Rd • ($FF – K)	Z,N,V
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V
DEC	Rd	Decrement	Rd ← Rd − 1	Z,N,V
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V
CLR	Rd	Clear Register	Rd ← $00	Z,N,V
SER	Rd	Set Register	Rd ← $FF	None
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr)<< 1	Z,C
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr)<< 1	Z,C
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr)<< 1	Z,C

MCU CONTROL INSTRUCTIONS

Mnemonics	Operands	Description	Operation	Flags
NOP		No Operation		None
SLEEP		Sleep	(see specific descr. for Sleep function)	None
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None
BREAK		Break	For On-Chip Debug Only	None

SECTION A.2: AVR INSTRUCTIONS FORMAT

ADC Rd, Rr ; Add with carry
$0 \le d \le 31, 0 \le r \le 31$; Rd ← Rd + Rr + C

Adds two registers and the contents of the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C Cycles: 1

Example:
```
                        ;Add R1:R0 to R3:R2
     add r2,r0          ;Add low byte
     adc r3,r1          ;Add with carry high byte
```

ADD Rd, Rr ; Add without carry
$0 \le d \le 31, 0 \le r \le 31$; Rd ← Rd + Rr

Adds two registers without the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C Cycles: 1

Example:
```
     add r1,r2          ;Add r2 to r1 (r1=r1+r2)
     add r28,r28        ;Add r28 to itself (r28=r28+r28)
```

ADIW Rd+1:Rd, K ; Add Immediate to Word
$d \in \{24,26,28,30\}, 0 \le K \le 63$; Rd + 1:Rd ← Rd + 1:Rd + K

Adds an immediate value (0–63) to a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers.

Flags: S, V, N, Z, C Cycles: 2

Example:
```
     adiw r25:24,1      ;Add 1 to r25:r24
     adiw ZH:ZL,63      ;Add 63 to the Z-pointer (r31:r30)
```

AND Rd, Rr ; Logical AND
$0 \le d \le 31, 0 \le r \le 31$; Rd ← Rd • Rr

Performs the logical AND between the contents of register Rd and register Rr and places the result in the destination register Rd.

Flags: S, V ← 0, N, Z Cycles: 1

Example:
```
     and r2,r3          ;Bitwise and r2 and r3, result in r2
     ldi r16,1          ;Set bitmask 0000 0001 in r16
     and r2,r16         ;Isolate bit 0 in r2
```

ANDI Rd, K ; Logical AND with Immediate
$16 \le d \le 31, 0 \le K \le 255$; Rd ← Rd • K

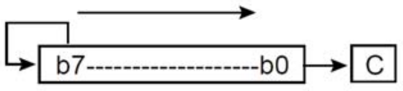
Performs the logical AND between the contents of register Rd and a constant and places the result in the destination register Rd.

Flags: S, V ← 0, N, Z Cycles: 1

Example:
```
      andi r17,$0F          ;Clear upper nibble of r17
      andi r18,$10          ;Isolate bit 4 in r18
```

ASR Rd **; Arithmetic Shift Right**
0 ≤ d ≤ 31

Shifts all bits in Rd one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C flag of the SREG. This operation effectively divides a signed value by two without changing its sign. The Carry flag can be used to round the result.



Flags: S, V, N, Z, C Cycles: 1

Example:
```
      ldi r16,$10           ;Load decimal 16 into r16
      asr r16               ;r16=r16 / 2
      ldi r17,$FC           ;Load -4 in r17
      asr r17               ;r17=r17/2
```

BCLR s **; Bit Clear in SREG**
0 ≤ s ≤7 **; SREG(s) ← 0**

Clears a single flag in SREG (Status Register).

Flags: I, T, H, S, V, N, Z, C Cycles: 1

Example:
```
      bclr 0                ;Clear Carry flag
      bclr 7                ;Disable interrupts
```

BLD Rd, b **; Bit Load from the T Flag in SREG to a Bit in Register**
0 ≤ d ≤ 31, 0 ≤ b ≤7 **; Rd(b) ← T**

Copies the T flag in the SREG (Status Register) to bit b in register Rd.

Flags: --- Cycles: 1

Example:
```
      bst r1,2              ;Store bit 2 of r1 in T flag
      bld r0,4              ;Load T flag into bit 4 of r0
```

BRBC s, k **; Branch if Bit in SREG is Cleared**
0 ≤ s ≤ 7, –64 ≤ k ≤ +63 **; If SREG(s) = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests a single bit in SREG (Status Register) and branches relatively to PC if the bit is set.

Flags: --- Cycles: 1or 2

Example:
```
      cpi r20,5             ;Compare r20 to the value 5
      brbc 1,noteq          ;Branch if Zero flag cleared
      ...
noteq:nop                   ;Branch destination (do nothing)
```

BRBS s, k **; Branch if Bit in SREG is Set**
0 ≤ s ≤ 7, –64 ≤ k ≤ +63 **; If SREG(s) = 1 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests a single bit in SREG (Status Register) and branches relatively to PC if the bit is set.

Flags: ---　　　　　　　　Cycles: 1 or 2

Example:

```
      bst r0,3              ;Load T bit with bit 3 of r0
      brbs 6,bitset         ;Branch T bit was set
      ...
      bitset: nop           ;Branch destination (do nothing)
```

BRCC k　　　　　　　　　; Branch if Carry Cleared

–64 ≤ k ≤ +63　　　　　　; If C = 0 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is cleared.

Flags: ---　　　　　　　　Cycles: 1 or 2

Example:

```
            add r22,r23     ;Add r23 to r22
            brcc nocarry    ;Branch if carry cleared
            ...
nocarry:    nop             ;Branch destination (do nothing)
```

BRCS k　　　　　　　　　; Branch if Carry Set

–64 ≤ k ≤ +63　　　　　　; If C = 1 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is set.

Flags: ---　　　　　　　　Cycles: 1 or 2

Example:

```
            cpi r26,$56     ;Compare r26 with $56
            brcs carry      ;Branch if carry set
            ...
carry:      nop             ;Branch destination (do nothing)
```

BREAK　　　　　　　　　; Break

The BREAK instruction is used by the on-chip debug system, and is normally not used in the application software. When the BREAK instruction is executed, the AVR CPU is set in the stopped mode. This gives the on-chip debugger access to internal resources.

Flags: ---　　　　　　　　Cycles: 1

Example:　　---

BREQ k　　　　　　　　　; Branch if Equal

–64 ≤ k ≤ +63　　; If Rd = Rr (Z = 1) then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Zero flag (Z) and branches relatively to PC if Z is set. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned or signed binary number represented in Rd was equal to the unsigned or signed binary number represented in Rr.

Flags: ---　　　　　　　　Cycles: 1 or 2

Example:

```
            ccp r1,r0       ;Compare registers r1 and r0
            breq equal      ;Branch if registers equal
            ...
equal:      nop             ;Branch destination (do nothing)
```

BRGE k ; Branch if Greater or Equal (Signed)
–64 ≤ k ≤ +63 ; If Rd ≥ Rr (N⊕V = 0) then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Signed flag (S) and branches relatively to PC if S is cleared. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the signed binary number represented in Rd was greater than or equal to the signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:
```
            cp r11,r12         ;Compare registers r11 and r12
            brge greateq       ;Branch if r11 ≥ r12 (signed)
            ...
greateq:    nop                ;Branch destination (do nothing)
```

BRHC k ; Branch if Half Carry Flag is Cleared
–64 ≤ k ≤ +63 ; If H = 0 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Half Carry flag (H) and branches relatively to PC if H is cleared.

Flags: --- Cycles: 1 or 2

Example:
```
            brhc hclear        ;Branch if Half Carry flag cleared
            ...
hclear:     nop                ;Branch destination (do nothing)
```

BRHS k ; Branch if Half Carry Flag is Set
–64 ≤ k ≤ +63 ; If H = 1 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Half Carry flag (H) and branches relatively to PC if H is set.

Flags: --- Cycles: 1 or 2

Example:
```
            brhs hset          ;Branch if Half Carry flag set
            ...
hset:       nop                ;Branch destination (do nothing)
```

BRID k ; Branch if Global Interrupt is Disabled
–64 ≤ k ≤ +63 ; If I = 0 then PC←PC + k + 1, else PC←PC + 1

Conditional relative branch. Tests the Global Interrupt flag (I) and branches relatively to PC if I is cleared.

Flags: --- Cycles: 1 or 2

Example:
```
            brid intdis        ;Branch if interrupt disabled
            ...
intdis:     nop                ;Branch destination (do nothing)
```

BRIE k ; Branch if Global Interrupt is Enabled
–64 ≤ k ≤ +63 ; If I = 1 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Global Interrupt flag (I) and branches relatively to PC if I is set.

Flags: --- Cycles: 1 or 2

Example:

```
            brie inten        ;Branch if interrupt enabled
            ...
inten:      nop               ;Branch destination (do nothing)
```

BRLO k **; Branch if Lower (Unsigned)**
$-64 \le k \le +63$ **; If Rd < Rr (C = 1) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is set. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned binary number represented in Rd was smaller than the unsigned binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            eor r19,r19       ;Clear r19
loop:       inc r19           ;Increment r19
            ...
            cpi r19,$10       ;Compare r19 with $10
            brlo loop         ;Branch if r19 < $10 (unsigned)
            nop               ;Exit from loop (do nothing)
```

BRLT k **; Branch if Less Than (Signed)**
$-64 \le k \le +63$ **; If Rd < Rr (N ⊕ V = 1) then PC← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Signed flag (S) and branches relatively to PC if S is set. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the signed binary number represented in Rd was less than the signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            bcp r16,r1        ;Compare r16 to r1
            brlt less         ;Branch if r16 < r1 (signed)
            ...
less:       nop               ;Branch destination (do nothing)
```

BRMI k **; Branch if Minus**
$-64 \le k \le +63$ **; If N=1 then PC←PC + k + 1, else PC←PC + 1**

Conditional relative branch. Tests the Negative flag (N) and branches relatively to PC if N is set.

Flags: --- Cycles: 1 or 2

Example:

```
            subi r18,4        ;Subtract 4 from r18
            brmi negative     ;Branch if result negative
            ...
negative:   nop               ;Branch destination (do nothing)
```

BRNE k **; Branch if Not Equal**
$-64 \le k \le +63$ **; If Rd ≠ Rr (Z = 0) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Zero flag (Z) and branches relatively to PC if Z is cleared. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned or signed binary number represented in Rd was not equal to the unsigned or signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            eor r27,r27       ;Clear r27
loop:       inc r27           ;Increment r27
            ...
            cpi r27,5         ;Compare r27 to 5
            brne loop         ;Branch if r27 not equal 5
            nop               ;Loop exit (do nothing)
```

BRPL k ; Branch if Plus
–64 ≤ k ≤ +63 ; If N = 0 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Negative flag (N) and branches relatively to PC if N is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            subi r26,$50      ;Subtract $50 from r26
            brpl positive     ;Branch if r26 positive
            ...
positive:   nop               ;Branch destination (do nothing)
```

BRSH k ; Branch if Same or Higher (Unsigned)
–64 ≤ k ≤ +63 ; If Rd ≥Rr (C = 0) then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is cleared. If the instruction is executed immediately after execution of any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned binary number represented in Rd was greater than or equal to the unsigned binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            subi r19,4        ;Subtract 4 from r19
            brsh highsm       ;Branch if r19 >= 4 (unsigned)
            ...
highsm:     nop               ;Branch destination (do nothing)
```

BRTC k ; Branch if the T Flag is Cleared
–64 ≤ k ≤ +63 ; If T = 0 then PC ← PC + k + 1, else PC ← PC + 1

Conditional relative branch. Tests the T flag and branches relatively to PC if T is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            bst r3,5          ;Store bit 5 of r3 in T flag
            brtc tclear       ;Branch if this bit was cleared
            ...
tclear:     nop               ;Branch destination (do nothing)
```

BRTS k ; **Branch if the T Flag is Set**
–64 ≤ k ≤ +63 ; **If T = 1 then PC←PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the T flag and branches relatively to PC if T is set.

Flags: --- Cycles: 1 or 2

Example:

```
            bst r3,5            ;Store bit 5 of r3 in T flag
            brts tset           ;Branch if this bit was set
            ...
tset:       nop                 ;Branch destination (do nothing)
```

BRVC k ; **Branch if Overflow Cleared**
–64 ≤ k ≤ +63 ; **If V = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Overflow flag (V) and branches relatively to PC if V is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            add r3,r4           ;Add r4 to r3
            brvc noover         ;Branch if no overflow
            ...
noover:     nop                 ;Branch destination (do nothing)
```

BRVS k ; **Branch if Overflow Set**
–64 ≤ k ≤ +63 ; **If V=1 then PC←PC + k + 1, else PC←PC + 1**

Conditional relative branch. Tests the Overflow flag (V) and branches relatively to PC if V is set.

Flags: --- Cycles: 1 or 2

Example:

```
            add r3,r4           ;Add r4 to r3
            brvs overfl         ;Branch if overflow
            ...
overfl:     nop                 ;Branch destination (do nothing)
```

BSET s ; **Bit Set in SREG**
0 ≤ s ≤ 7 ; **SREG(s) ← 1**

Sets a single flag or bit in SREG (Status Register).

Flags: Any of the flags. Cycles: 1

Example:

```
            bset 6              ;Set T flag
            bset 7              ;Enable interrupt
```

BST Rd,b ; **Bit Store from Register to T Flag in SREG**
0 ≤ d ≤ 31, 0 ≤ b ≤ 7 ; **T ← Rd(b)**

Stores bit b from Rd to the T flag in SREG (Status Register).

Flags: T Cycles: 1

Example:

```
                                ;Copy bit
            bst r1,2            ;Store bit 2 of r1 in T flag
            bld r0,4            ;Load T into bit 4 of r0t
```

CALL k ; Long Call to a Subroutine
0 ≤ k < 64K (Devices with 16 bits PC) or 0 ≤ k < 4M (Devices with 22 bits PC)

Calls to a subroutine within the entire program memory. The return address (to the instruction after the CALL) will be stored onto the stack. (See also RCALL.) The stack pointer uses a post-decrement scheme during CALL.

Flags: --- Cycles: 4

Example:

```
            mov r16,r0          ;Copy r0 to r16
            call check          ;Call subroutine
            nop                 ;Continue (do nothing)
            ...
check:      cpi r16,$42         ;Check if r16 has a special value
            breq error          ;Branch if equal
            ret                 ;Return from subroutine
            ...
error:      rjmp error          ;Infinite loop
```

CBI A, b ; Clear Bit in I/O Register
0 ≤ A ≤ 31, 0 ≤ b ≤7 ; I/O(A,b) ← 0

Clears a specified bit in an I/O Register. This instruction operates on the lower 32 I/O registers (addresses 0–31).

Flags: --- Cycles: 2

Example:

```
            cbi $12,7           ;Clear bit 7 in Port D
```

CBR Rd, k ; Clear Bits in Register
16 ≤ d ≤ 31, 0 ≤ K ≤ 255 ; Rd ← Rd • ($FF – K)

Clears the specified bits in register Rd. Performs the logical AND between the contents of register Rd and the complement of the constant mask K.

Flags: S, N, V ← 0, Z Cycles: 1

Example:

```
            cbr r16,$F0         ;Clear upper nibble of r16
            cbr r18,1           ;Clear bit 0 in r18
```

CLC ; Clear Carry Flag
; C ← 0

Clears the Carry flag (C) in SREG (Status Register).

Flags: C ← 0. Cycles: 1

Example:

```
            add r0,r0           ;Add r0 to itself
            clc                 ;Clear Carry flag
```

CLH ; Clear Half Carry Flag
; H ← 0

Clears the Half Carry flag (H) in SREG (Status Register).

Flags: H ← 0. Cycles: 1

Example:

```
            clh                 ;Clear the Half Carry flag
```

CLI **; Clear Global Interrupt Flag**
; I ← 0

Clears the Global Interrupt flag (I) in SREG (Status Register). The interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction.

Flags: I ← 0. Cycles: 1

Example:

```
            in temp, SREG     ;Store SREG value
                              ;(temp must be defined by user)
            cli               ;Disable interrupts during timed sequence
            sbi EECR, EEMWE   ;Start EEPROM write
            sbi EECR, EEWE    ;
            out SREG, temp    ;Restore SREG value (I-flag)
```

CLN **; Clear Negative Flag**
; N ← 0

Clears the Negative flag (N) in SREG (Status Register).

Flags: N ← 0. Cycles: 1

Example:

```
            add r2,r3         ;Add r3 to r2
            cln               ;Clear Negative flag
```

CLR Rd **; Clear Register**
$0 \leq d \leq 31$ **; Rd ← Rd ⊕ Rd**

Clears a register. This instruction performs an Exclusive-OR between a register and itself. This will clear all bits in the register..

Flags: S ← 0 , N ← 0, V ← 0, Z ← 0 Cycles: 1

Example:

```
            clr r18           ;Clear r18
loop:       inc r18           ;Increment r18
            ...
            cpi r18,$50       ;Compare r18 to $50
            brne loop
```

CLS **; Clear Signed Flag**
; S ← 0

Clears the Signed flag (S) in SREG (Status Register).

Flags: S ← 0. Cycles: 1

Example:

```
            add r2,r3         ;Add r3 to r2
            cls               ;Clear Signed flag
```

CLT **; Clear T Flag**
; T ← 0

Clears the T flag in SREG (Status Register).

Flags: T ← 0. Cycles: 1

Example:

```
            clt               ;Clear T flag
```

CLV **; Clear Overflow Flag**
; V ← 0

Clears the Overflow flag (V) in SREG (Status Register).
Flags: V ← 0. Cycles: 1
Example:
```
          add r2,r3         ;Add r3 to r2
          clv               ;Clear Overflow flag
```

CLZ **; Clear Zero Flag**
; Z ← 0

Clears the Zero flag (Z) in SREG (Status Register).
Flags: Z ← 0. Cycles: 1
Example:
```
          clz               ;Clear zero
```

COM Rd **; One's Complement**
$0 \leq d \leq 31$ **; Rd ← $FF – Rd**

This instruction performs a one's complement of register Rd.
Flags: S, V ← 0, N , Z ← 1, C. Cycles: 1
Example:
```
          com r4            ;Take one's complement of r4
          breq zero         ;Branch if zero
          ...
zero:     nop               ;Branch destination (do nothing)
```

CP Rd,Rr **; Compare**
$0 \leq d \leq 31, 0 \leq r \leq 31$ **; Rd – Rr**

This instruction performs a compare between two registers, Rd and Rr. None of the registers are changed. All conditional branches can be used after this instruction.
Flags: H, S,V, N, Z, C. Cycles: 1
Example:
```
          cp r4,r19         ;Compare r4 with r19
          brne noteq        ;Branch if r4 not equal r19
          ...
noteq:    nop               ;Branch destination (do nothing)
```

CPC Rd,Rr **; Compare with Carry**
$0 \leq d \leq 31, 0 \leq r \leq 31$ **; Rd – Rr – C**

This instruction performs a compare between two registers, Rd and Rr, and also takes into account the previous carry. None of the registers are changed. All conditional branches can be used after this instruction.
Flags: H, S, V, N, Z, C. Cycles: 1
Example:
```
                            ;Compare r3:r2 with r1:r0
          cp r2,r0          ;Compare low byte
          cpc r3,r1         ;Compare high byte
          brne noteq        ;Branch if not equal
          ...
noteq:    nop               ;Branch destination (do nothing)
```

CPI Rd,K ; **Compare with Immediate**
16 ≤ d ≤ 31, 0 ≤ K ≤ 255 ; **Rd – K**

This instruction performs a compare between register Rd and a constant. The register is not changed. All conditional branches can be used after this instruction.

Flags: H, S,V, N, Z, C. Cycles: 1

Example:

```
            cpi r19,3           ;Compare r19 with 3
            brne error          ;Branch if r19 not equal 3
            ...
error:      nop                 ;Branch destination (do nothing)
```

CPSE Rd,Rr ; **Compare Skip if Equal**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; **If Rd = Rr then PC ← PC + 2 or 3 else PC ← PC + 1**

This instruction performs a compare between two registers Rd and Rr, and skips the next instruction if Rd = Rr.

Flags:--- Cycles: 1, 2, or 3

Example:

```
            inc r4              ;Increment r4
            cpse r4,r0          ;Compare r4 to r0
            neg r4              ;Only executed if r4 not equal r0
            nop                 ;Continue (do nothing)
```

DEC Rd ; **Decrement**
0 ≤ d ≤ 31 ; **Rd ← Rd – 1**

Subtracts one from the contents of register Rd and places the result in the destination register Rd.

The C flag in SREG is not affected by the operation, thus allowing the DEC instruction to be used on a loop counter in multiple-precision computations.

When operating on unsigned values, only BREQ and BRNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

Flags: S,V, N, Z. Cycles: 1

Example:

```
            ldi r17,$10         ;Load constant in r17
loop:       add r1,r2           ;Add r2 to r1
            dec r17             ;Decrement r17
            brne loop           ;Branch if r17 not equal 0
            nop                 ;Continue (do nothing)
```

EOR Rd,Rr ; **Exclusive OR**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; **Rd ← Rd ⊕ Rr**

Performs the logical Exclusive OR between the contents of register Rd and register Rr and places the result in the destination register Rd.

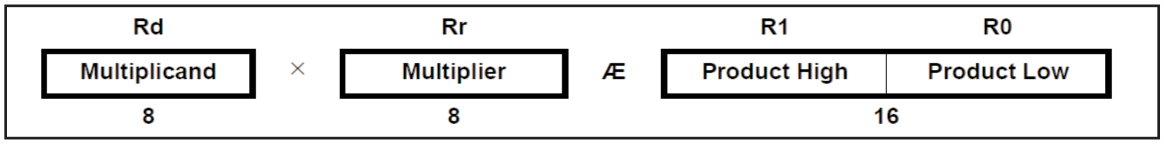
Flags: S, V, Z ← 0, N, Z. Cycles: 1

Example:

```
            eor r4,r4           ;Clear r4
            eor r0,r22          ;Bitwise XOR between r0 and r22
```

FMUL Rd,Rr ; **Fractional Multiply Unsigned**
16 ≤ d ≤ 23, 16 ≤ r ≤ 23 ; **R1:R0 ← Rd × Rr (unsigned ← unsigned × unsigned)**

This instruction performs 8-bit × 8-bit → 16-bit unsigned multiplication and shifts the result one bit left.



Let (N.Q) denote a fractional number with N binary digits left of the radix point, and Q binary digits right of the radix point. A multiplication between two numbers in the formats (N1.Q1) and (N2.Q2) results in the format ((N1 + N2).(Q1 + Q2)). For signal processing applications, the (1.7) format is widely used for the inputs, resulting in a (2.14) format for the product. A left shift is required for the high byte of the product to be in the same format as the inputs. The FMUL instruction incorporates the shift operation in the same number of cycles as MUL.

The (1.7) format is most commonly used with signed numbers, while FMUL performs an unsigned multiplication. This instruction is therefore most useful for calculating one of the partial products when performing a signed multiplication with 16-bit inputs in the (1.15) format, yielding a result in the (1.31) format. (Note: The result of the FMUL operation may suffer from a 2's complement overflow if interpreted as a number in the (1.15) format.) The MSB of the multiplication before shifting must be taken into account, and is found in the carry bit. See the following example.

The multiplicand Rd and the multiplier Rr are two registers containing unsigned fractional numbers where the implicit radix point lies between bit 6 and bit 7. The 16-bit unsigned fractional product with the implicit radix point between bit 14 and bit 15 is placed in R1 (high byte) and R0 (low byte).

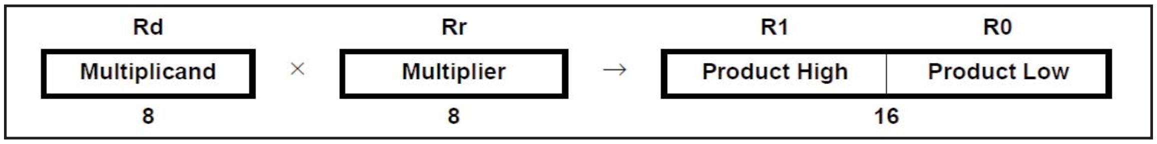
Flags: Z, C. Cycles: 2

Example:

```
;*****************************************************************
;* DESCRIPTION
;* Signed fractional multiply of two 16-bit numbers with 32-bit result.
;* r19:r18:r17:r16 = ( r23:r22 * r21:r20 ) << 1
;*****************************************************************
            fmuls 16x16_32:
            clr r2
            fmuls r23, r21          ;((signed)ah *(signed)bh) << 1
            movw r19:r18, r1:r0
            fmul r22, r20           ;(al * bl) << 1
            adc r18, r2
            movwr17:r16, r1:r0
            fmulsu r23, r20         ;((signed)ah * bl) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
            fmulsu r21, r22         ;((signed)bh * al) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
```

FMULS Rd,Rr ; Fractional Multiply Signed
$16 \leq d \leq 23$, $16 \leq r \leq 23$; R1:R0 ← Rd × Rr (signed ← signed × signed)

This instruction performs 8-bit × 8-bit → 16-bit signed multiplication and shifts the result one bit left.



Let (N.Q) denote a fractional number with N binary digits left of the radix point, and Q binary digits right of the radix point. A multiplication between two numbers in the formats (N1.Q1) and (N2.Q2) results in the format ((N1 + N2).(Q1 + Q2)). For signal processing applications, the (1.7) format is widely used for the inputs, resulting in a (2.14) format for the product. A left shift is required for the high byte of the product to be in the same format as the inputs. The FMULS instruction incorporates the shift operation in the same number of cycles as MULS.

The multiplicand Rd and the multiplier Rr are two registers containing signed fractional numbers where the implicit radix point lies between bit 6 and bit 7. The 16-bit signed fractional product with the implicit radix point between bit 14 and bit 15 is placed in R1 (high byte) and R0 (low byte).

Note that when multiplying 0x80 (–1) with 0x80 (–1), the result of the shift operation is 0x8000 (–1). The shift operation thus gives a two's complement overflow. This must be checked and handled by software.

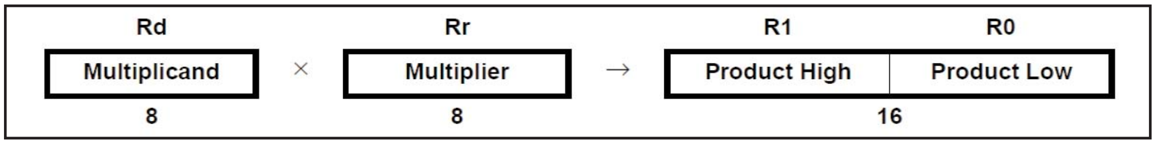
This instruction is not available in all devices. Refer to the device-specific instruction set summary.

Flags: Z, C. Cycles: 2

Example:
```
fmuls r23,r22          ;Multiply signed r23 and r22 in
                       ;(1.7) format, result in (1.15) format
movw r23:r22,r1:r0     ;Copy result back in r23:r22
```

FMULSU Rd,Rr ; Fractional Multiply Signed with Unsigned
$16 \leq d \leq 23$, $16 \leq r \leq 23$; R1:R0 ← Rd × Rr

This instruction performs 8-bit × 8-bit → 16-bit signed multiplication and shifts the result one bit left.



Let (N.Q) denote a fractional number with N binary digits left of the radix point, and Q binary digits right of the radix point. A multiplication between two numbers in the formats (N1.Q1) and (N2.Q2) results in the format ((N1 + N2).(Q1 + Q2)). For signal processing applications, the (1.7) format is widely used for the inputs, resulting in a (2.14) format for the product. A left shift is required for the high byte of the product to be in the same format as the inputs. The FMULSU instruction incorporates the shift operation in the same number of cycles as MULSU.

The (1.7) format is most commonly used with signed numbers, while FMULSU performs a multiplication with one unsigned and one signed input. This instruction is therefore most useful for calculating two of the partial products when performing a signed multiplication with 16-bit inputs in the (1.15) format, yielding a result in the (1.31) format. (Note: The result of the FMULSU operation may suffer from a 2's complement overflow if interpreted as a number in the (1.15) format.) The MSB of the multiplication before shifting must be taken into account, and is found in the carry bit. See the following example.

The multiplicand Rd and the multiplier Rr are two registers containing fractional numbers where the implicit radix point lies between bit 6 and bit 7. The multiplicand Rd is a signed fractional number, and the multiplier Rr is an unsigned fractional number. The 16-bit signed fractional product with the implicit radix point between bit 14 and bit 15 is placed in R1 (high byte) and R0 (low byte).

This instruction is not available in all devices. Refer to the device-specific instruction set summary.

Flags: Z, C. Cycles: 2

Example:

```
;***************************************************************
;* DESCRIPTION
;* Signed fractional multiply of two 16-bit numbers with 32-bit result.
;* r19:r18:r17:r16 = ( r23:r22 * r21:r20 ) << 1
;***************************************************************
fmuls16x16_32:
            clrr2
            fmuls r23, r21    ;((signed)ah * (signed)bh) << 1
            movwr19:r18, r1:r0
            fmul r22, r20     ;(al * bl) << 1
            adc r18, r2
            movwr17:r16, r1:r0
            fmulsu r 23, r20  ;((signed)ah * bl) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
            fmulsu r21, r22   ;((signed)bh * al) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
```

ICALL ; Indirect Call to Subroutine

Indirect call of a subroutine pointed to by the Z (16 bits) pointer register in the register file. The Z-pointer register is 16 bits wide and allows calls to a subroutine within the lowest 64K words (128K bytes) section in the program memory space. The stack pointer uses a post-decrement scheme during ICALL.

This instruction is not available in all devices. Refer to the device-specific instruction set summary.

Flags: --- Cycles: 3

Example:

```
      mov r30,r0        ;Set offset to call table
      icall             ;Call routine pointed to by r31:r30
```

IJMP ; **Indirect Jump**

Indirect jump to the address pointed to by the Z (16 bits) pointer register in the register file. The Z-pointer register is 16 bits wide and allows jumps within the lowest 64K words (128K bytes) of the program memory.

This instruction is not available in all devices. Refer to the device-specific instruction set summary.

Flags:--- Cycles: 2

Example:

```
            mov r30,r0          ;Set offset to jump table
            ijmp                ;Jump to routine pointed to by r31:r30
```

IN Rd,A ; **Load an I/O Location to Register**
$0 \le d \le 31$, $0 \le A \le 63$; **Rd ← I/O(A)**

Loads data from the I/O space (ports, timers, configuration registers, etc.) into register Rd in the register file.

Flags:--- Cycles: 1

Example:

```
            in r25,$16          ;Read Port B
            cpi r25,4           ;Compare read value to constant
            breq exit           ;Branch if r25=4
            ...
exit:       nop                 ;Branch destination (do nothing)
```

INC Rd ; **Increment**
$0 \le d \le 31$; **Rd ← Rd + 1**

Adds one to the contents of register Rd and places the result in the destination register Rd.

The C flag in SREG is not affected by the operation, thus allowing the INC instruction to be used on a loop counter in multiple-precision computations.

When operating on unsigned numbers, only BREQ and BRNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

Flags: S, V, N, Z. Cycles: 1

Example:

```
            clr r22             ;Clear r22
loop:       inc r22             ;Increment r22
            ...
            cpi r22,$4F         ;Compare r22 to $4f
            brne loop           ;Branch if not equal
            nop                 ;Continue (do nothing)
```

JMP k ; **Jump**
$0 \le k < 4M$; **PC ← k**

Jump to an address within the entire 4M (words) program memory. See also RJMP.

Flags:--- Cycles: 3

Example:

```
            mov r1,r0         ;Copy r0 to r1
            jmp farplc        ;Unconditional jump
            ...
farplc:     nop               ;Jump destination (do nothing)
```

LD ; Load Indirect from Data Space to Register ; using Index X

Loads one byte indirect from the data space to a register. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the X (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPX in register in the I/O area has to be changed.

The X-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented.

These features are especially suited for accessing arrays, tables, and stack pointer usage of the X-pointer register. Note that only the low byte of the X-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPX register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement is added to the entire 24-bit address on such devices.

Syntax:	Operation:	Comment:
(i) LD Rd, X	Rd ← (X)	X: Unchanged
(ii) LD Rd, X+	Rd ← (X) , X ← X + 1	X: Post-incremented
(iii) LD Rd, –X	X ← X – 1, Rd ← (X)	X: Pre-decremented

Flags:--- Cycles: 2

Example:

```
            clr r27           ;Clear X high byte
            ldi r26,$60       ;Set X low byte to $60
            ld r0,X+          ;Load r0 with data space loc. $60
                              ;X post inc)
            ld r1,X           ;Load r1 with data space loc. $61
            ldi r26,$63       ;Set X low byte to $63
            ld r2,X           ;Load r2 with data space loc. $63
            ld r3,-X          ;Load r3 with data space loc.
                              ;$62(X pre dec)
```

LD (LDD) ; Load Indirect from Data Space to Register ; using Index Y

Loads one byte indirect with or without displacement from the data space to a register. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Y (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPY in register in the I/O area has to be changed.

The Y-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for accessing arrays, tables, and stack pointer usage of the Y-pointer register. Note that only the low byte of the Y-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPY register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement/displacement is added to the entire 24-bit address on such devices.

Syntax:	Operation:	Comment:
(i) LD Rd, Y	Rd ← (Y)	Y: Unchanged
(ii) LD Rd, Y+	Rd ← (Y) ,Y ← Y + 1	Y: Postincremented
(iii) LD Rd, –Y	Y ← Y – 1, Rd ← (Y)	Y: Predecremented
(iiii) LDD Rd, Y + q	Rd ← (Y + q)	Y: Unchanged, q: Displacement

Flags:---   Cycles: 2

Example:

```
clr r29          ;Clear Y high byte
ldi r28,$60      ;Set Y low byte to $60
ld r0,Y+         ;Load r0 with data space loc. $60(Y post inc)
ld r1,Y          ;Load r1 with data space loc. $61
ldi r28,$63      ;Set Y low byte to $63
ld r2,Y          ;Load r2 with data space loc. $63
ld r3,-Y         ;Load r3 with data space loc. $62(Y pre dec)
ldd r4,Y+2       ;Load r4 with data space loc. $64
```

LD (LDD) ; Load Indirect from Data Space to Register ; using Index Z

Loads one byte indirect with or without displacement from the data space to a register. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Z (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPZ in register in the I/O area has to be changed.

The Z-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for stack pointer usage of the Z-pointer register, however because the Z-pointer register can be used for indirect subroutine calls, indirect jumps, and table lookup, it is often more convenient to use the X or Y-pointer as a dedicated stack pointer. Note that only the low byte of the Z-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPZ register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/decrement/displacement is added to the entire 24-bit address on such devices.

Syntax:	Operation:	Comment:
(i) LD Rd, Z	Rd ← (Z)	Z: Unchanged
(ii) LD Rd, Z+	Rd ← (Z) Z ← Z + 1	Z: Postincrement
(iii) LD Rd, –Z	Z ← Z – 1 Rd ← (Z)	Z: Predecrement
(iiii) LDD Rd, Z + q	Rd ← (Z + q)	Z: Unchanged, q: Displacement

Flags:---　　　　　　　Cycles: 2

Example:

```
      clr r31             ;Clear Z high byte
      ldi r30,$60         ;Set Z low byte to $60
      ld r0,Z+            ;Load r0 with data space loc.$60(Z postinc.)
      ld r1,Z             ;Load r1 with data space loc. $61
      ldi r30,$63         ;Set Z low byte to $63
      ld r2,Z             ;Load r2 with data space loc. $63
      ld r3,-Z            ;Load r3 with data space loc. $62(Z predec.)
      ldd r4,Z+2          ;Load r4 with data space loc. $64
```

LDI Rd,K　　　　　　　　　; Load Immediate
16 ≤ d ≤ 31, 0 ≤ K ≤ 255　　　; Rd ← K

Loads an 8-bit constant directly to registers 16 to 31.

Flags:---　　　　　　　Cycles: 1

Example:

```
            clr r31             ;Clear Z high byte
            ldi r30,$F0         ;Set Z low byte to $F0
            lpm                 ;Load constant from program
                                ;memory pointed to by Z
```

LDS Rd,k　　　　　　　　　; Load Direct from Data Space
0 ≤ d ≤ 31, 0 ≤ k ≤ 65535　　; Rd ← (k)

Loads one byte from the data space to a register. The data space consists of the register file, I/O memory, and SRAM.

Flags:---　　　　　　　Cycles: 2

Example:

```
            lds r2,$FF00        ;Load r2 with the contents of
                                ;data space location $FF00
            add r2,r1           ;add r1 to r2
            sts $FF00,r2        ;Write back
```

LPM　　　　　　　　　　　; Load Program Memory

Loads one byte pointed to by the Z-register into the destination register Rd. This instruction features a 100% space effective constant initialization or constant data fetch. The program memory is organized in 16-bit words while the Z-pointer is a byte address. Thus, the least significant bit of the Z-pointer selects either the low byte (ZLSB = 0) or the high byte (ZLSB = 1). This instruction can address the first 64K bytes (32K words) of program memory. The Z-pointer register can either be left unchanged by the operation, or it can be incremented. The incrementation does not apply to the RAMPZ register.

Devices with self-programming capability can use the LPM instruction to read the Fuse and Lock bit values. Refer to the device documentation for a detailed description.

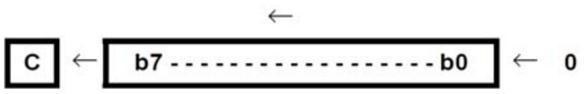
Syntax:	Operation:	Comment:
(i) LPM	R0 ← (Z)	Z: Unchanged, R0 implied Rd
(ii) LPM Rd, Z	Rd ← (Z)	Z: Unchanged
(iii) LPM Rd, Z+	Rd ← (Z), Z ← Z + 1	Z: Postincremented

Flags:--- Cycles: 3

Example:

```
          ldi ZH, high(Table_1<<1);Initialize Z-pointer
          ldi ZL, low(Table_1<<1)
          lpm r16, Z             ;Load constant from program
                                 ;Memory pointed to by Z (r31:r30)
          ...
Table_1:
.dw 0x5876                       ;0x76 is addresses when ZLSB = 0
                                 ;0x58 is addresses when ZLSB = 1
          ...
```

LSL Rd **; Logical Shift Left**
0 ≤ d ≤ 31



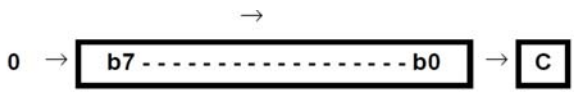
Shifts all bits in Rd one place to the left. Bit 0 is cleared. Bit 7 is loaded into the C flag of the SREG (Status Register). This operation effectively multiplies signed and unsigned values by two.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
          add r0,r4        ;Add r4 to r0
          lsl r0           ;Multiply r0 by 2
```

LSR Rd **; Logical Shift Left**
0 ≤ d ≤ 31



Shifts all bits in Rd one place to the right. Bit 7 is cleared. Bit 0 is loaded into the C flag of the SREG. This operation effectively divides an unsigned value by two. The C flag can be used to round the result.

Flags: S, V, N ← 0, Z, C. Cycles: 1

Example:

```
          add r0,r4        ;Add r4 to r0
          lsr r0           ;Divide r0 by 2
```

MOV Rd,Rr **; Copy Register**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 **; Rd ← Rr**

This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.

Flags: --- Cycles: 1

Example:

```
            mov r16,r0          ;Copy r0 to r16
            call check          ;Call subroutine
            ...
check:      cpi r16,$11         ;Compare r16 to $11
            ...
            ret                 ;Return from subroutine
```

MOVW Rd + 1:Rd,Rr + 1:Rrd ; Copy RegisterWord
d ∈ {0,2,...,30}, r ∈ {0,2,...,30} ; Rd + 1:Rd ← Rr + 1:Rr

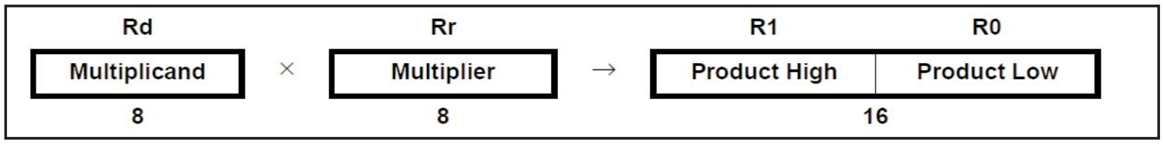
This instruction makes a copy of one register pair into another register pair. The source register pair Rr + 1:Rr is left unchanged, while the destination register pair Rd + 1:Rd is loaded with a copy of Rr + 1:Rr.

Flags: --- Cycles: 1

Example:

```
            movw r17:16,r1:r0 ;Copy r1:r0 to r17:r16
            call check          ;Call subroutine
            ...
check:      cpi r16,$11         ;Compare r16 to $11
            ...
            cpi r17,$32         ;Compare r17 to $32
            ...
            ret                 ;Return from subroutine
```

MUL Rd,Rr ; Multiply Unsigned
$0 \leq d \leq 31$, $0 \leq r \leq 31$; R1:R0 ← Rd × Rr(unsigned ← unsigned × unsigned)



This instruction performs 8-bit × 8-bit → 16-bit unsigned multiplication.

The multiplicand Rd and the multiplier Rr are two registers containing unsigned numbers. The 16-bit unsigned product is placed in R1 (high byte) and R0 (low byte). Note that if the multiplicand or the multiplier is selected from R0 or R1 the result will overwrite those after multiplication.

Flags: Z, C. Cycles: 2

Example:

```
            mul r5,r4           ;Multiply unsigned r5 and r4
            movw r4,r0          ;Copy result back in r5:r4
```

MULS Rd,Rr ; Multiply Signed
$16 \leq d \leq 31$, $16 \leq r \leq 31$; R1:R0 ← Rd × Rr(signed ← signed × signed)

This instruction performs 8-bit × 8-bit → 16-bit signed multiplication.

The multiplicand Rd and the multiplier Rr are two registers containing signed numbers. The 16-bit signed product is placed in R1 (high byte) and R0 (low byte).

Flags: Z, C. Cycles: 2

Example:

```
            muls r21,r20        ;Multiply signed r21 and r20
            movw r20,r0         ;Copy result back in r21:r20
```

MULSU Rd,Rr ; **Multiply Signed with Unsigned**
16 ≤ d ≤ 31, 16 ≤ r ≤ 31 ; **R1:R0 ← Rd × Rr (signed ← signed × unsigned)**

This instruction performs 8-bit × 8-bit → 16-bit multiplication of a signed and an unsigned number.

The multiplicand Rd and the multiplier Rr are two registers. The multiplicand Rd is a signed number, and the multiplier Rr is unsigned. The 16-bit signed product is placed in R1 (high byte) and R0 (low byte).

Flags: Z, C. Cycles: 2

Example:---

NEG Rd ; **Two's Complement**
0 ≤ d ≤ 31 ; **Rd ← $00 – Rd**

Replaces the contents of register Rd with its two's complement; the value $80 is left unchanged.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:
```
            sub r11,r0          ;Subtract r0 from r11
            brpl positive       ;Branch if result positive
            neg r11             ;Take two's complement of r11
positive:   nop                 ;Branch destination (do nothing)
```

NOP ; **No Operation**

This instruction performs a single-cycle No Operation.

Flags: ---. Cycles: 1

Example:
```
            clr r16             ;Clear r16
            ser r17             ;Set r17
            out $18,r16         ;Write zeros to Port B
            nop                 ;Wait (do nothing)
            out $18,r17         ;Write ones to Port B
```

OR Rd,Rr ; **Logical OR**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; **Rd ← Rd OR Rr**

Performs the logical OR between the contents of register Rd and register Rr and places the result in the destination register Rd.

Flags: S, V ← 0, N, Z. Cycles: 1

Example:
```
            or r15,r16          ;Do bitwise or between registers
            bst r15,6           ;Store bit 6 of r15 in T flag
            brts ok             ;Branch if T flag set
            ...
ok:         nop                 ;Branch destination (do nothing)
```

ORI Rd,K ; Logical OR with Immediate
16 ≤ d ≤ 31, 0 ≤ K ≤ 255 ; Rd ← Rd OR K

Performs the logical OR between the contents of register Rd and a constant and places the result in the destination register Rd.

Flags: S, V ← 0, N, Z. Cycles: 1

Example:

```
            ori r16,$F0       ;Set high nibble of r16
            ori r17,1         ;Set bit 0 of r17
```

OUT A,Rr ; Store Register to I/O Location
0 ≤ r ≤ 31, 0 ≤ A ≤ 63 ; I/O(A) ← Rr

Stores data from register Rr in the register file to I/O space (ports, timers, configuration registers, etc.).

Flags: ---. Cycles: 1

Example:

```
            clr r16           ;Clear r16
            ser r17           ;Set r17
            out $18,r16       ;Write zeros to Port B
            nop               ;Wait (do nothing)
            out $18,r17       ;Write ones to Port B
```

POP Rd ; Pop Register from Stack
0 ≤ d ≤ 31 ; Rd ← STACK

This instruction loads register Rd with a byte from the STACK. The stack pointer is pre-incremented by 1 before the POP.

Flags: ---. Cycles: 2

Example:

```
            call routine      ;Call subroutine
            ...
routine:    push r14          ;Save r14 on the stack
            push r13          ;Save r13 on the stack
            ...
            pop r13           ;Restore r13
            pop r14           ;Restore r14
            ret               ;Return from subroutine
```

PUSH Rr ; Push Register on Stack
0 ≤ d ≤ 31 ; STACK ← Rr

This instruction stores the contents of register Rr on the STACK. The stack pointer is post-decremented by 1 after the PUSH.

Flags: ---. Cycles: 2

Example:

```
            call routine      ;Call subroutine
            ...
routine:    push r14          ;Save r14 on the stack
            push r13          ;Save r13 on the stack
            ...
            pop r13           ;Restore r13
            pop r14           ;Restore r14
            ret               ;Return from subroutine
```

RCALL k ; **Relative Call to Subroutine**
–2K ≤ k < 2K ; **PC ← PC + k + 1**

Relative call to an address within PC – 2K + 1 and PC + 2K (words). The return address (the instruction after the RCALL) is stored onto the stack. (See also CALL.) In the assembler, labels are used instead of relative operands. For AVR microcontrollers with program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location. The stack pointer uses a post-decrement scheme during RCALL.

Flags: ---. Cycles: 3

Example:

```
            rcall routine       ;Call subroutine
            ...
routine:    push r14            ;Save r14 on the stack
            ...
            pop r14             ;Restore r14
            ret                 ;Return from subroutine
```

RET ; **Return from Subroutine**

Returns from subroutine. The return address is loaded from the stack. The stack pointer uses a pre-increment scheme during RET.

Flags: ---. Cycles: 4

Example:

```
            call routine        ;Call subroutine
            ...
routine:    push r14            ;Save r14 on the stack
            ...
            pop r14             ;Restore r14
            ret                 ;Return from subroutine
```

RETI ; **Return from Interrupt**

Returns from interrupt. The return address is loaded from the stack and the Global Interrupt flag is set.

Note that the Status Register is not automatically stored when entering an interrupt routine, and it is not restored when returning from an interrupt routine. This must be handled by the application program. The stack pointer uses a pre-increment scheme during RETI.

Flags: ---. Cycles: 4

Example:

```
            ...
extint:     push r0             ;Save r0 on the stack
            ...
            pop r0              ;Restore r0
            reti                ;Return and enable interrupts
```

RJMP k ; **Relative Jump**
–2K ≤ k < 2K ; **PC ← PC + k + 1**

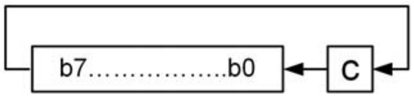
Relative jump to an address within PC – 2K +1 and PC + 2K (words). In the assembler, labels are used instead of relative operands. For AVR microcontrollers with program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location.

Flags: ---. Cycles: 2

Example:

```
            cpi r16,$42       ;Compare r16 to $42
            brne error        ;Branch if r16 not equal $42
            rjmp ok           ;Unconditional branch
error:      add r16,r17       ;Add r17 to r16
            inc r16           ;Increment r16
ok:         nop               ;Destination for rjmp (do nothing)
```

ROL Rd ; **Rotate Left through Carry**
0 ≤ d ≤ 31



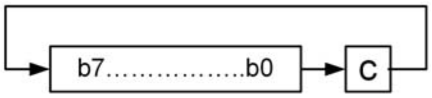
Shifts all bits in Rd one place to the left. The C flag is shifted into bit 0 of Rd. Bit 7 is shifted into the C flag. This operation combined with LSL effectively multiplies multibyte signed and unsigned values by two.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
            lsl r18           ;Multiply r19:r18 by two
            rol r19           ;r19:r18 is a signed or unsigned word
            brcs oneenc       ;Branch if carry set
            ...
oneenc:     nop               ;Branch destination (do nothing)
```

ROR Rd ; **Rotate Right through Carry**
0 ≤ d ≤ 31



Shifts all bits in Rd one place to the right. The C flag is shifted into bit 7 of Rd. Bit 0 is shifted into the C flag. This operation combined with ASR effectively divides multibyte signed values by two. Combined with LSR, it effectively divides multibyte unsigned values by two. The Carry flag can be used to round the result.

Flags: S, V, N, Z, C. Cycles: 1

Example:

```
            lsr r19           ;Divide r19:r18 by two
            ror r18           ;r19:r18 is an unsigned two-byte integer
            brcc zeroenc1     ;Branch if carry cleared
            asr r17           ;Divide r17:r16 by two
            ror r16           ;r17:r16 is a signed two-byte integer
            brcc zeroenc2     ;Branch if carry cleared
            ...
zeroenc1:   nop               ;Branch destination (do nothing)
            ...
zeroenc2:   nop               ;Branch destination (do nothing)
```

SBC Rd,Rr ; Subtract with Carry
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; Rd ← Rd – Rr – C

Subtracts two registers and subtracts with the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
                        ;Subtract r1:r0 from r3:r2
        sub r2,r0       ;Subtract low byte
        sbc r3,r1       ;Subtract with carry high byte
```

SBCI Rd,K ; Subtract Immediate with Carry
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; Rd ← Rd – K – C

Subtracts a constant from a register and subtracts with the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
                        ;Subtract $4F23 from r17:r16
        subi r16,$23    ;Subtract low byte
        sbci r17,$4F    ;Subtract with carry high byte
```

SBI A,b ; Set Bit in I/O Register
0 ≤ A ≤ 31, 0 ≤ b ≤ 7 ; I/O(A,b) ← 1

Sets a specified bit in an I/O register. This instruction operates on the lower 32 I/O registers.

Flags: ---. Cycles: 2

Example:

```
        out $1E,r0      ;Write EEPROM address
        sbi $1C,0       ;Set read bit in EECR
        in r1,$1D       ;Read EEPROM data
```

SBIC A,b ; Skip if Bit in I/O Register is Cleared
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; If I/O(A,b) = 0 then PC ← PC + 2 (or 3) else PC ← PC + 1

This instruction tests a single bit in an I/O register and skips the next instruction if the bit is cleared. This instruction operates on the lower 32 I/O registers.

Flags:---. Cycles: 1/2/3

Example:

```
e2wait: sbic $1C,1      ;Skip next inst. if EEWE cleared
        rjmp e2wait     ;EEPROM write not finished
        nop             ;Continue (do nothing)
```

SBIS A,b ; Skip if Bit in I/O Register is Set
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; If I/O(A,b) = 1 then PC ← PC + 2 (or 3) else PC ← PC + 1

This instruction tests a single bit in an I/O register and skips the next instruction if the bit is set. This instruction operates on the lower 32 I/O registers.

Flags: ---. Cycles: 1/2/3

Example:

```
waitset: sbis $10,0     ;Skip next inst. if bit 0 in Port D set
         rjmp waitset   ;Bit not set
         nop            ;Continue (do nothing)
```

SBIW Rd + 1:Rd,K ; Subtract Immediate from Word
d ∈ {24,26,28,30}, 0 ≤ K ≤ 63 ; Rd + 1:Rd ← Rd + 1:Rd – K

Subtracts an immediate value (0–63) from a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers.

Flags: S, V, N, Z, C. Cycles: 2

Example:

```
        sbiw r25:r24,1    ;Subtract 1 from r25:r24
        sbiw YH:YL,63     ;Subtract 63 from the Y-pointer
```

SBR Rd,K ; Set Bits in Register
16 ≤ d ≤ 31, 0 ≤ K ≤ 255 ; Rd ← Rd OR K

Sets specified bits in register Rd. Performs the logical ORI between the contents of register Rd and a constant mask K and places the result in the destination register Rd.

Flags: S,V←0, N, Z. Cycles: 1

Example:

```
        sbr r16,3         ;Set bits 0 and 1 in r16
        sbr r17,$F0       ;Set 4 MSB in r17
```

SBRC Rr,b ; Skip if Bit in Register is Cleared
0 ≤ r ≤ 31, 0 ≤ b ≤7 ; If Rr(b) = 0 then PC ← PC + 2 or 3 else PC ← PC + 1

This instruction tests a single bit in an I/O register and skips the next instruction if the bit is set. This instruction operates on the lower 32 I/O registers.

Flags: --- Cycles: 1/2/3

Example:

```
        sub r0,r1         ;Subtract r1 from r0
        sbrc r0,7         ;Skip if bit 7 in r0 cleared
        sub r0,r1         ;Only executed if bit7 in r0 not cleared
        nop               ;Continue (do nothing)
```

SBRS Rr,b ; Skip if Bit in Register is Set
0 ≤ r ≤ 31, 0 ≤ b ≤7 ; If Rr(b) = 1 then PC ← PC + 2 or 3 else PC ← PC + 1

This instruction tests a single bit in a register and skips the next instruction if the bit is set.

Flags: H, S, V, N, Z, C. Cycles: 1/2/3

Example:

```
        sub r0,r1         ;Subtract r1 from r0
        sbrs r0,7         ;Skip if bit 7 in r0 set
        neg r0            ;Only executed if bit 7 in r0 not set
        nop               ;Continue (do nothing)
```

SEC ; Set Carry Flag
; C ← 1

Sets the Carry flag (C) in SREG (Status Register).

Flags: C ← 1. Cycles: 1

Example:

```
        sec               ;Set Carry flag
        adc r0,r1         ;r0=r0+r1+1
```

SEH **; Set Half Carry Flag**
; H ← 1

Sets the Half Carry (H) in SREG (Status Register).
Flags: H ← 1. Cycles: 1
Example:

```
        seh                 ;Set Half Carry flag
```

SEI **; Set Global Interrupt Flag**
; I ← 1

Sets the Global Interrupt flag (I) in SREG (Status Register). The instruction following SEI will be executed before any pending interrupts.
Flags: I ← 1. Cycles: 1
Example:

```
        sei                 ;Set global interrupt enable
        sec                 ;Set Carry flag
        ;Note: will set Carry flag before any pending interrupt
```

SEN **; Set Negative Flag**
; N ← 1

Sets the Negative flag (N) in SREG (Status Register).
Flags: N ← 1. Cycles: 1
Example:

```
        add r2,r19          ;Add r19 to r2
        sen                 ;Set Negative flag
```

SER Rd **; Set all Bits in Register**
16 ≤ d ≤ 31 **; Rd ← $FF**

Loads $FF directly to register Rd.
Flags: ---. Cycles: 1
Example:

```
        ser r17             ;Set r17
        out $18,r17         ;Write ones to Port B
```

SES **; Set Signed Flag**
; S ← 1

Sets the Signed flag (S) in SREG (Status Register).
Flags: S ← 1. Cycles: 1
Example:

```
        add r2,r19          ;Add r19 to r2
        ses                 ;Set Negative flag
```

SET **; Set T Flag**
; T ← 1

Sets the T flag in SREG (Status Register).
Flags: T ← 1. Cycles: 1
Example:

```
        set                 ;Set T flag
```

SEV **; Set Overflow Flag**
; V ← 1

Sets the Overflow flag (V) in SREG (Status Register).

Flags: V ← 1. Cycles: 1

Example:

```
        sev                 ;Set Overflow flag
```

SEZ **; Set Zero Flag**
; Z ← 1

Sets the Zero flag (Z) in SREG (Status Register).

Flags: Z ← 1. Cycles: 1

Example:

```
        sez                 ;Set Z flag
```

SLEEP

This instruction sets the circuit in sleep mode defined by the MCU control register.

Flags: ---. Cycles: 1

Example:

```
        mov r0,r11          ;Copy r11 to r0
        ldi r16,(1<<SE)     ;Enable sleep mode
        out MCUCR, r16
        sleep               ;Put MCU in sleep mode
```

SPM **; Store Program Memory**

SPM can be used to erase a page in the program memory, to write a page in the program memory (that is already erased), and to set Boot Loader Lock bits. In some devices, the program memory can be written one word at a time, in other devices an entire page can be programmed simultaneously after first filling a temporary page buffer. In all cases, the program memory must be erased one page at a time. When erasing the program memory, the RAMPZ and Z-register are used as page address. When writing the program memory, the RAMPZ and Z-register are used as page or word address, and the R1:R0 register pair is used as data(1). When setting the Boot Loader Lock bits, the R1:R0 register pair is used as data.

Refer to the device documentation for detailed description of SPM usage. This instruction can address the entire program memory.

Flags: ---. Cycles: depends on the operation

	Syntax:	Operation:	Comment:
(i)	SPM	(RAMPZ:Z) ← $ffff	Erase program memory page
(ii)	SPM	(RAMPZ:Z) ← R1:R0	Write program memory word
(iii)	SPM	(RAMPZ:Z) ← R1:R0	Write temporary page buffer
(iv)	SPM	(RAMPZ:Z) ← TEMP	Write temporary page buffer to program memory
(v)	SPM	BLBITS ← R1:R0	Set Boot Loader Lock bits

ST ; Store Indirect From Register to Data Space ; using Index X

Stores one byte indirect from a register to data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the X (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPX register in the I/O area has to be changed.

The X-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented.These features are especially suited for accessing arrays, tables, and stack pointer usage of the X-pointer register. Note that only the low byte of the X-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPX register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement is added to the entire 24-bit address on such devices.

Flags: ---. Cycles: 2

	Syntax:	Operation:	Comment:
(i)	ST X, Rr	(X) ← Rr	X: Unchanged
(ii)	ST X+, Rr	(X) ← Rr X ← X + 1	X: Postincremented
(iii)	ST –X, Rr	X ← X – 1 (X) ← Rr	X: Predecremented

Example:
```
        clr r27           ;Clear X high byte
        ldi r26,$60       ;Set X low byte to $60
        st X+,r0          ;Store r0 in data space loc. $60(X post inc)
        st X,r1           ;Store r1 in data space loc. $61
        ldi r26,$63       ;Set X low byte to $63
        st X,r2           ;Store r2 in data space loc. $63
        st -X,r3          ;Store r3 in data space loc. $62(X pre dec)
```

ST (STD) ; Store Indirect From Register to Data Space ; using Index Y

Stores one byte indirect with or without displacement from a register to data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Y (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPY register in the I/O area has to be changed.

The Y-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for accessing arrays, tables, and stack pointer usage of the Y-pointer register. Note that only the low byte of the Y-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPY register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement/displacement is added to the entire 24-bit address on such devices.

Flags: ---. Cycles:2

	Syntax:	Operation:	Comment:
(i)	ST Y, Rr	(Y) ← Rr	Y: Unchanged
(ii)	ST Y+, Rr	(Y) ← Rr Y ← Y + 1	Y: Postincremented
(iii)	ST –Y, Rr	Y ← Y – 1 (Y) ← Rr	Y: Predecremented
(iiii)	STD Y + q, Rr	(Y + q) ← Rr	Y: Unchanged q: Displacement

Example:

```
clr r29          ;Clear Y high byte
ldi r28,$60      ;Set Y low byte to $60
st Y+,r0         ;Store r0 in data space loc. $60 (Y postinc.)
st Y,r1          ;Store r1 in data space loc. $61
ldi r28,$63      ;Set Y low byte to $63
st Y,r2          ;Store r2 in data space loc. $63
st -Y,r3         ;Store r3 in data space loc. $62 (Y predec.)
std Y+2,r4       ;Store r4 in data space loc. $64
```

ST (STD) ; Store Indirect From Register to Data Space using Index Z

Stores one byte indirect with or without displacement from a register to data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Z (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPZ register in the I/O area has to be changed.

The Z-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for stack pointer usage of the Z-pointer register; however, because the Z-pointer register can be used for indirect subroutine calls, indirect jumps and table lookup, it is often more convenient to use the X or Y-pointer as a dedicated stack pointer. Note that only the low byte of the Z-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPZ register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/decrement/displacement is added to the entire 24-bit address on such devices.

Flags: ---. Cycles: 2

	Syntax:	Operation:	Comment:
(i)	ST Z, Rr	(Z) ←Rr	Z: Unchanged
(ii)	ST Z+, Rr	(Z) ← Rr Z ← Z + 1	Z: Postincremented
(iii)	ST –Z, Rr	Z ← Z – 1 (Z) ← Rr	Z: Predecremented
(iiii)	STD Z + q, Rr	(Z + q) ← Rr	Z: Unchanged, q: Displacement

Example:

```
      clr r31          ;Clear Z high byte
      ldi r30,$60      ;Set Z low byte to $60
      st Z+,r0         ;Store r0 in data space loc. $60 (Z postinc.)
      st Z,r1          ;Store r1 in data space loc. $61
      ldi r30,$63      ;Set Z low byte to $63
      st Z,r2          ;Store r2 in data space loc. $63
      st -Z,r3         ;Store r3 in data space loc. $62 (Z predec.)
      std Z+2,r4       ;Store r4 in data space loc. $64
```

STS k,Rr **; Store Direct to Data Space**
0 ≤ r ≤ 31, 0 ≤ k ≤ 65535 **; (k) ← Rr**

Stores one byte from a register to the data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

A 16-bit address must be supplied. Memory access is limited to the current data segment of 64K bytes. The STS instruction uses the RAMPD register to access memory above 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPD register in the I/O area has to be changed.

Flags:---. Cycles: 2

Example:

```
      lds r2,$FF00     ;Load r2 with the contents of location $FF00
      add r2,r1        ;Add r1 to r2
      sts $FF00,r2     ;Write back
```

SUB Rd,Rr **; Subtract without Carry**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 **; Rd ← Rd – Rr**

Subtracts two registers and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
            sub r13,r12      ;Subtract r12 from r13
            brne noteq       ;Branch if r12 not equal r13
            ...
noteq:      nop              ;Branch destination (do nothing)
```

SUBI Rd,K **; Subtract Immediate**
16 ≤ d ≤ 31, 0 ≤ K ≤ 255 **; Rd ← Rd – K**

Subtracts a register and a constant and places the result in the destination register Rd. This instruction works on registers R16 to R31 and is very well suited for operations on the X, Y, and Z-pointers.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:
```
            subi r22,$11      ;Subtract $11 from r22
            brne noteq        ;Branch if r22 not equal $11
            ...
noteq:      nop               ;Branch destination (do nothing)
```

SWAP Rd ; **Swap Nibbles**
0 ≤ d ≤ 31 ; **R(7:4) ← Rd(3:0), R(3:0) ← Rd(7:4)**

Swaps high and low nibbles in a register.

Flags:---. Cycles: 1

Example:
```
            inc r1            ;Increment r1
            swap r1           ;Swap high and low nibble of r1
            inc r1            ;Increment high nibble of r1
            swap r1           ;Swap back
```

TST Rd ; **Test for Zero or Minus**
0 ≤ d ≤ 31 ; **Rd ← Rd • Rd**

Tests if a register is zero or negative. Performs a logical AND between a register and itself. The register will remain unchanged.

Flags: S, V ← 1, N, Z. Cycles: 1

Example:
```
            tst r0            ;Test r0
            breq zero         ;Branch if r0=0
            ...
zero:       nop               ;Branch destination (do nothing)
```

WDR ; **Watchdog Reset**

This instruction resets the watchdog timer. This instruction must be executed within a limited time given by the WD prescaler.

Flags:---. Cycles: 1

Example:
```
            wdr               ;Reset watchdog timer
```

SECTION A.3: AVR REGISTER SUMMARY

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C
$3E ($5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
$3C ($5C)	OCR0	Timer/Counter0 Output Compare Register							
$3B ($5B)	GICR	INT1	INT0	INT2	–	–	–	IVSEL	IVCE
$3A ($5A)	GIFR	INTF1	INTF0	INTF2	–	–	–	–	–
$39 ($59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
$38 ($58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
$37 ($57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
$36 ($56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
$35 ($55)	MCUCR	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
$34 ($54)	MCUCSR	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF
$33 ($53)	TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
$32 ($52)	TCNT0	Timer/Counter0 (8 Bits)							
$31 ($51)	OSCCAL	Oscillator Calibration Register							
	OCDR	On-Chip Debug Register							
$30 ($50)	SFIOR	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10
$2F ($4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
$2E ($4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
$2D ($4D)	TCNT1H	Timer/Counter1 – Counter Register High Byte							
$2C ($4C)	TCNT1L	Timer/Counter1 – Counter Register Low Byte							
$2B ($4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte							
$2A ($4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte							
$29 ($49)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte							
$28 ($48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte							
$27 ($47)	ICR1H	Timer/Counter1 – Input Capture Register High Byte							
$26 ($46)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte							
$25 ($45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
$24 ($44)	TCNT2	Timer/Counter2 (8 Bits)							
$23 ($43)	OCR2	Timer/Counter2 Output Compare Register							
$22 ($42)	ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB
$21 ($41)	WDTCR	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0
$20 ($40)	UBRRH	URSEL	–	–	–	UBRR[11:8]			
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
$1F ($3F)	EEARH	–	–	–	–	–	–	EEAR9	EEAR8
$1E ($3E)	EEARL	EEPROM Address Register Low Byte							
$1D ($3D)	EEDR	EEPROM Data Register							
$1C ($3C)	EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE
$1B ($3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
$1A ($3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
$19 ($39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
$18 ($38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
$17 ($37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
$16 ($36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
$15 ($35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
$14 ($34)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
$13 ($33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
$0F ($2F)	SPDR	SPI Data Register							
$0E ($2E)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X
$0D ($2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
$0C ($2C)	UDR	USART I/O Data Register							
$0B ($2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
$0A ($2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
$09 ($29)	UBRRL	USART Baud Rate Register Low Byte							
$08 ($28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
$07 ($27)	ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
$06 ($26)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
$05 ($25)	ADCH	ADC Data Register High Byte							
$04 ($24)	ADCL	ADC Data Register Low Byte							
$03 ($23)	TWDR	Two-wire Serial Interface Data Register							
$02 ($22)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	TWA1	TWA0	TWGCE
$01 ($21)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	–	TWPS1	TWPS0
$00 ($20)	TWBR	Two-wire Serial Interface Bit Rate Register							

APPENDIX B

BASICS OF WIRE WRAPPING

OVERVIEW

This appendix shows the basics of wire wrapping.

BASICS OF WIRE WRAPPING

Note: For this tutorial appendix, you will need the following:
Wire-wrapping tool (Radio Shack part number 276-1570)
30-gauge (30-AWG) wire for wire wrapping
(Thanks to Shannon Looper and Greg Boyle for their assistance on this section.)

The following describes the basics of wire wrapping.

1. There are several different types of wire-wrap tools available. The best one is available from Radio Shack for less than $10. The part number for Radio Shack is 276-1570. This tool combines the wrap and unwrap functions in the same end of the tool and includes a separate stripper. We found this to be much easier to use than the tools that combined all these features on one two-ended shaft. There are also wire-wrap guns, which are, of course, more expensive.
2. Wire-wrapping wire is available prestripped in various lengths or in bulk on a spool. The prestripped wire is usually more expensive and you are restricted to the different wire lengths you can afford to buy. Bulk wire can be cut to any length you wish, which allows each wire to be custom fit.
3. Serveral different types of wire-wrap boards are available. These are usually called *perfboards* or *wire-wrap boards*. These types of boards are sold at many electronics stores (such as Radio Shack). The best type of board has plating around the holes on the bottom of the board. These boards are better because the sockets and pins can be soldered to the board, which makes the circuit more mechanically stable.
4. Choose a board that is large enough to accommodate all the parts in your design with room to spare so that the wiring does not become too cluttered. If you wish to expand your project in the future, you should be sure to include enough room on the original board for the complete circuit. Also, if possible, the layout of the IC on the board needs to be such that signals go from left to right just like the schematics.
5. To make the wiring easier and to keep pressure off the pins, install one standoff on each corner of the board. You may also wish to put standoffs on the top of the board to add stability when the board is on its back.
6. For power hook-up, use some type of standard binding post. Solder a few single wire-wrap pins to each power post to make circuit connections (to at least one pin for each IC in the circuit).
7. To further reduce problems with power, each IC must have its own connection to the main power of the board. If your perfboard does not have built-in power buses, run a separate power and ground wire from each IC to the main power. In other words, DO NOT daisy chain (making a chip-to-chip connection is called *daisy chaining*) power connections, as each connection down the line will have more wire and more resistance to get power through. However, daisy chaining is acceptable for other connections such as data, address, and control buses.
8. You must use wire-wrap sockets. These sockets have long square pins whose edges will cut into the wire as it is wrapped around the pin.
9. Wire wrapping will not work on round legs. If you need to wrap to compo-

nents, such as capacitors, that have round legs, you must also solder these connections. The best way to connect single components is to install individual wire-wrap pins into the board and then solder the components to the pins. An alternate method is to use an empty IC socket to hold small components such as resistors and wrap them to the socket.

10. The wire should be stripped about 1 inch. This will allow 7 to 10 turns for each connection. The first turn or turn-and-a-half should be insulated. This prevents stripped wire from coming in contact with other pins. This can be accomplished by inserting the wire as far as it will go into the tool before making the connection.
11. Try to keep wire lengths to a minimum. This prevents the circuit from looking like a bird nest. Be neat and use color coding as much as possible. Use only red wires for V_{CC} and black wires for ground connections. Also use different colors for data, address, and control signal connections. These suggestions will make troubleshooting much easier.
12. It is standard practice to connect all power lines first and check them for continuity. This will eliminate trouble later on.
13. It's also a good idea to mark the pin orientation on the bottom of the board. Plastic templates are available with pin numbers preprinted on them specifically for this purpose or you can make your own from paper. Forgetting to reverse pin order when looking at the bottom of the board is a very common mistake when wire wrapping circuits.
14. To prevent damage to your circuit, place a diode (such as IN5338) in reverse bias across the power supply. If the power gets hooked up backwards, the diode will be forward biased and will act as a short, keeping the reversed voltage from your circuit.
15. In digital circuits, there can be a problem with current demand on the power supply. To filter the noise on the power supply, a 100 μF electrolytic capacitor and a 0.1 μF monolithic capacitor are connected from V_{CC} to ground, in parallel with each other, at the entry point of the power supply to the board. These two together will filter both the high- and the low-frequency noises. Instead of using two capacitors in parallel, you can use a single 20–100 μF tantalum capacitor. Remember that the long lead is the positive one.
16. To filter the transient current, use a 0.1 μF monolithic capacitor for each IC. Place the 0.1 μF monolithic capacitor between V_{CC} and ground of each IC. Make sure the leads are as short as possible.

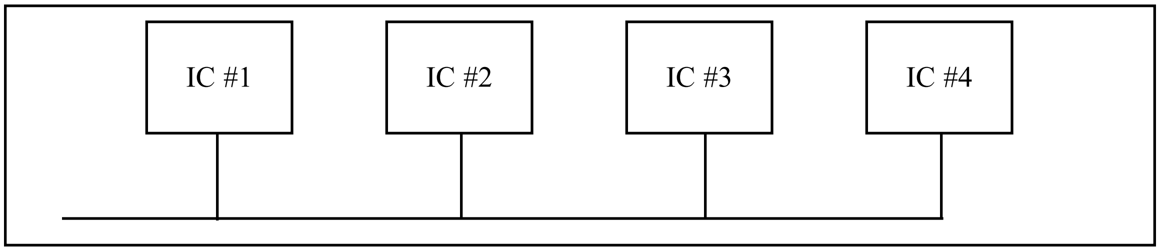


Figure B-1. Daisy Chain Connection (not recommended for power lines)

APPENDIX C

IC INTERFACING AND SYSTEM DESIGN ISSUES

OVERVIEW

This appendix provides an overview of IC technology and AVR interfacing. In addition, we look at the microcontroller-based system as a whole and examine some general issues in system design.

First, in Section C.1, we provide an overview of IC technology. Then, in Section C.2, the internal details of AVR I/O ports and interfacing are discussed. Section C.3 examines system design issues.

C.1: OVERVIEW OF IC TECHNOLOGY

In this section we examine IC technology and discuss some major developments in advanced logic families. Because this is an overview, it is assumed that the reader is familiar with logic families on the level presented in basic digital electronics books.

Transistors

The transistor was invented in 1947 by three scientists at Bell Laboratory. In the 1950s, transistors replaced vacuum tubes in many electronics systems, including computers. It was not until 1959 that the first integrated circuit was successfully fabricated and tested by Jack Kilby of Texas Instruments. Prior to the invention of the IC, the use of transistors, along with other discrete components such as capacitors and resistors, was common in computer design. Early transistors were made of germanium, which was later abandoned in favor of silicon. This was because the slightest rise in temperature resulted in massive current flows in germanium-based transistors. In semiconductor terms, it is because the band gap of germanium is much smaller than that of silicon, resulting in a massive flow of electrons from the valence band to the conduction band when the temperature rises even slightly. By the late 1960s and early 1970s, the use of the silicon-based IC was widespread in mainframes and minicomputers. Transistors and ICs at first were based on P-type materials. Later on, because the speed of electrons is much higher (about two-and-a-half times) than the speed of holes, N-type devices replaced P-type devices. By the mid-1970s, NPN and NMOS transistors had replaced the slower PNP and PMOS transistors in every sector of the electronics industry, including in the design of microprocessors and computers. Since the early 1980s, CMOS (complementary MOS) has become the dominant technology of IC design. Next we provide an overview of differences between MOS and bipolar transistors. See Figure C-1.

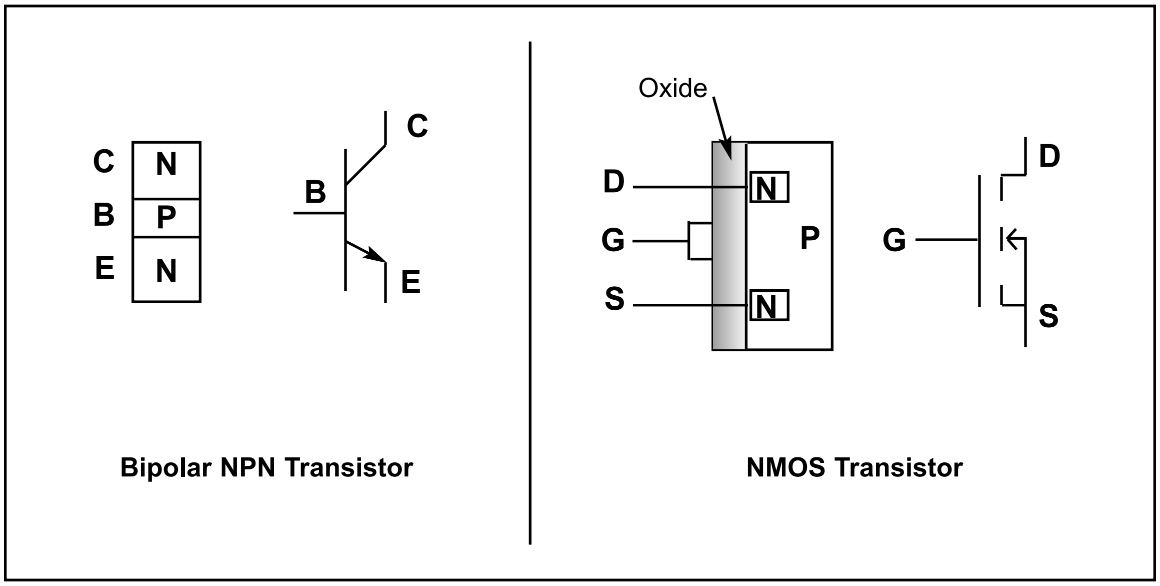


Figure C-1. Bipolar vs. MOS Transistors

MOS vs. bipolar transistors

There are two types of transistors: bipolar and MOS (metal-oxide semiconductor). Both have three leads. In bipolar transistors, the three leads are referred to as the *emitter*, *base*, and *collector*, while in MOS transistors they are named *source*, *gate*, and *drain*. In bipolar transistors, the carrier flows from the emitter to the collector, and the base is used as a flow controller. In MOS transistors, the carrier flows from the source to the drain, and the gate is used as a flow controller. In NPN-type bipolar transistors, the electron carrier leaving the emitter must overcome two voltage barriers before it reaches the collector (see Figure C-1). One is the N-P junction of the emitter-base and the other is the P-N junction of the base-collector. The voltage barrier of the base-collector is the most difficult one for the electrons to overcome (because it is reverse-biased) and it causes the most power dissipation. This led to the design of the unipolar type transistor called *MOS*. In N-channel MOS transistors, the electrons leave the source and reach the drain without going through any voltage barrier. The absence of any voltage barrier in the path of the carrier is one reason why MOS dissipates much less power than bipolar transistors. The low power dissipation of MOS allows millions of transistors to fit on a single IC chip. In today's technology, putting 10 million transistors into an IC is common, and it is all because of MOS technology. Without the MOS transistor, the advent of desktop personal computers would not have been possible, at least not so soon. The bipolar transistors in both the mainframes and minicomputers of the 1960s and 1970s were bulky and required expensive cooling systems and large rooms. MOS transistors do have one major drawback: They are slower than bipolar transistors. This is due partly to the gate capacitance of the MOS transistor. For a MOS to be turned on, the input capacitor of the gate takes time to charge up to the turn-on (threshold) voltage, leading to a longer propagation delay.

Overview of logic families

Logic families are judged according to (1) speed, (2) power dissipation, (3) noise immunity, (4) input/output interface compatibility, and (5) cost. Desirable qualities are high speed, low power dissipation, and high noise immunity (because it prevents the occurrence of false logic signals during switching transition). In interfacing logic families, the more inputs that can be driven by a single output, the better. This means that high-driving-capability outputs are desired. This, plus the fact that the input and output voltage levels of MOS and bipolar transistors are not compatible mean that one must be concerned with the ability of one logic family to drive the other one. In terms of the cost of a given logic family, it is high during the early years of its introduction but it declines as production and use rise.

The case of inverters

As an example of logic gates, we look at a simple inverter. In a one-transistor inverter, the transistor plays the role of a switch, and R is the pull-up resistor. See Figure C-2. For this inverter to work most effectively in digital circuits, however, the R value must be high when the transistor is “on” to limit the current flow from V_{CC} to ground in order to have low power dissipation (P = VI, where V

= 5 V). In other words, the lower the I, the lower the power dissipation. On the other hand, when the transistor is "off", R must be a small value to limit the voltage drop across R, thereby making sure that V_{OUT} is close to V_{CC}. This is a contradictory demand on R. This is one reason that logic gate designers use active components (transistors) instead of passive components (resistors) to implement the pull-up resistor R.

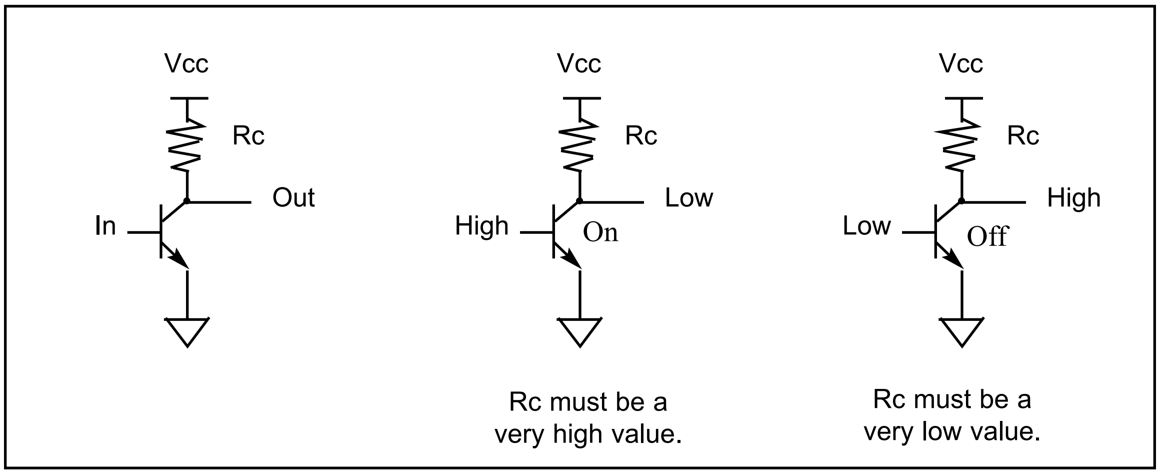


Figure C-2. One-Transistor Inverter with Pull-up Resistor

The case of a TTL inverter with totem-pole output is shown in Figure C-3. In Figure C-3, Q3 plays the role of a pull-up resistor.

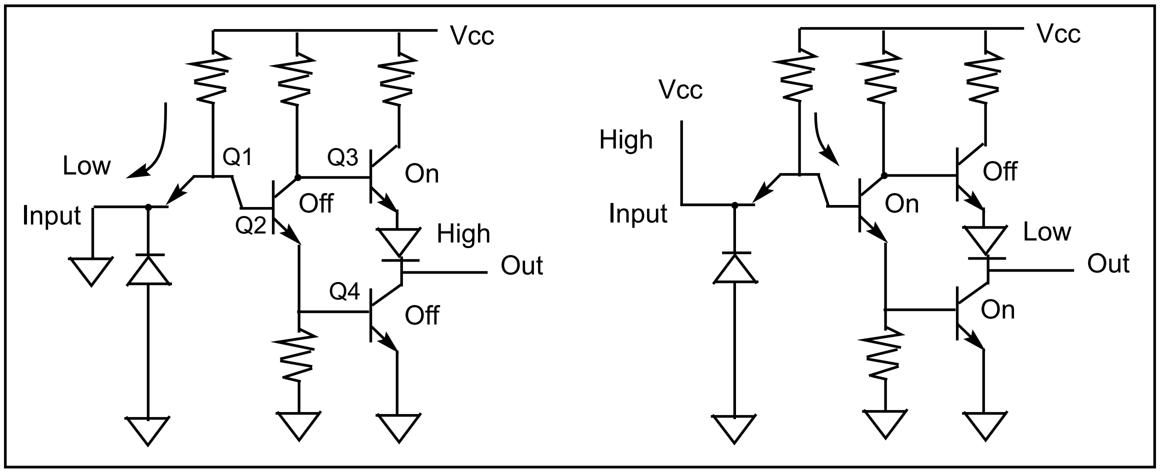


Figure C-3. TTL Inverter with Totem-Pole Output

CMOS inverter

In the case of CMOS-based logic gates, PMOS and NMOS are used to construct a CMOS (complementary MOS) inverter as shown in Figure C-4. In CMOS inverters, when the PMOS transistor is off, it provides a very high impedance path, making leakage current almost zero (about 10 nA); when the PMOS is on, it provides a low resistance on the path of V_{DD} to load. Because the speed of the hole is slower than that of the electron, the PMOS transistor is wider to compensate for this disparity; therefore, PMOS transistors take more space than NMOS transistors in the CMOS gates. At the end of this section we will see an open-collector gate in which the pull-up resistor is provided externally, thereby allowing system designers to choose the value of the pull-up resistor.

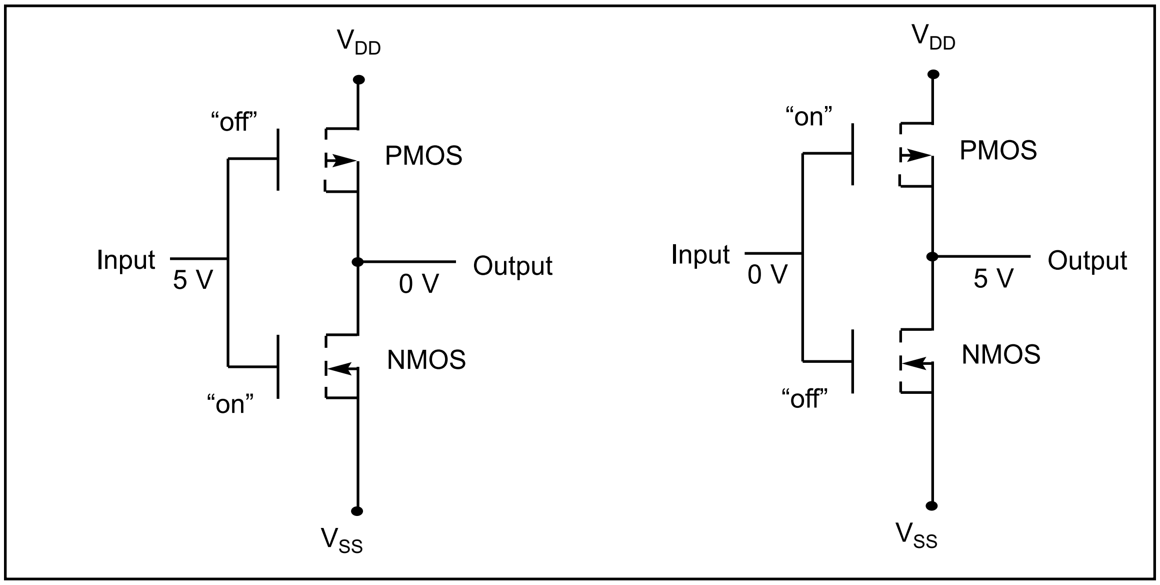


Figure C-4. CMOS Inverter

Input/output characteristics of some logic families

In 1968 the first logic family made of bipolar transistors was marketed. It was commonly referred to as the *standard TTL* (transistor-transistor logic) family. The first MOS-based logic family, the CD4000/74C series, was marketed in 1970. The addition of the Schottky diode to the base-collector of bipolar transistors in the early 1970s gave rise to the S family. The Schottky diode shortens the propagation delay of the TTL family by preventing the collector from going into what is called *deep saturation*. Table C-1 lists major characteristics of some logic families. In Table C-1, note that as the CMOS circuit's operating frequency rises, the power dissipation also increases. This is not the case for bipolar-based TTL.

Table C-1: Characteristics of Some Logic Families

Characteristic	**STD TTL**	**LSTTL**	**ALSTTL**	**HCMOS**
V_{CC}	5 V	5 V	5 V	5 V
V_{IH}	2.0 V	2.0 V	2.0 V	3.15 V
V_{IL}	0.8 V	0.8 V	0.8 V	1.1 V
V_{OH}	2.4 V	2.7 V	2.7 V	3.7 V
V_{OL}	0.4 V	0.5 V	0.4 V	0.4 V
I_{IL}	−1.6 mA	−0.36 mA	−0.2 mA	−1 μA
I_{IH}	40 μA	20 μA	20 μA	1 μA
I_{OL}	16 mA	8 mA	4 mA	4 mA
I_{OH}	−400 μA	−400 μA	−400 μA	4 mA
Propagation delay	10 ns	9.5 ns	4 ns	9 ns
Static power dissipation (f = 0)	10 mW	2 mW	1 mW	0.0025 nW
Dynamic power dissipation at f = 100 kHz	10 mW	2 mW	1 mW	0.17 mW

History of logic families

Early logic families and microprocessors required both positive and negative power voltages. In the mid-1970s, 5 V V_{CC} became standard. In the late 1970s, advances in IC technology allowed combining the speed and drive of the S family with the lower power of LS to form a new logic family called *FAST* (Fairchild Advanced Schottky TTL). In 1985, AC/ACT (Advanced CMOS Technology), a much higher speed version of HCMOS, was introduced. With the introduction of FCT (Fast CMOS Technology) in 1986, the speed gap between CMOS and TTL at last was closed. Because FCT is the CMOS version of FAST, it has the low power consumption of CMOS but the speed is comparable with TTL. Table C-2 provides an overview of logic families up to FCT.

Table C-2: Logic Family Overview

Product	Year Introduced	Speed (ns)	Static Supply Current (mA)	High/Low Family Drive (mA)
Std TTL	1968	40	30	−2/32
CD4K/74C	1970	70	0.3	−0.48/6.4
LS/S	1971	18	54	−15/24
HC/HCT	1977	25	0.08	−6/−6
FAST	1978	6.5	90	−15/64
AS	1980	6.2	90	−15/64
ALS	1980	10	27	−15/64
AC/ACT	1985	10	0.08	−24/24
FCT	1986	6.5	1.5	−15/64

Reprinted by permission of Electronic Design Magazine, c. 1991.

Recent advances in logic families

As the speed of high-performance microprocessors reached 25 MHz, it shortened the CPU's cycle time, leaving less time for the path delay. Designers normally allocate no more than 25% of a CPU's cycle time budget to path delay. Following this rule means that there must be a corresponding decline in the propagation delay of logic families used in the address and data path as the system frequency is increased. In recent years, many semiconductor manufacturers have responded to this need by providing logic families that have high speed, low noise, and high drive I/O. Table C-3 provides the characteristics of high-performance logic families introduced in recent years. ACQ/ACTQ are the second-generation advanced CMOS (ACMOS) with much lower noise. While ACQ has the CMOS input level, ACTQ is equipped with TTL-level input. The FCTx and FCTx-T are second-generation FCT with much higher speed. (The "x" in the FCTx and FCTx-T refers to various speed grades, such as A, B, and C, where A means low speed and C means high speed.) For designers who are well versed in using the FAST logic family, FASTr is an ideal choice because it is faster than FAST, has higher driving capability (I_{OL}, I_{OH}), and produces much lower noise than FAST. At the time of this writing, next to ECL and gallium arsenide logic gates, FASTr is the fastest logic family in the market (with the 5 V V_{CC}), but the power consumption is high relative to other logic families, as shown in Table C-3. The combining of high-speed bipolar TTL and the low power consumption of CMOS has given birth to what is called *BICMOS*. Although BICMOS seems to be the future trend in IC design, at this time it is expensive due to extra steps required in BICMOS IC fabrication, but in some cases there is no other choice. (For example, Intel's Pentium microprocessor, a BICMOS product, had to use high-speed bipolar transistors to speed up some of the internal functions.) Table C-3 provides advanced logic characteristics. The "x" is for different speeds designated as A, B, and C. A is the slowest one while C is the fastest one. The above data is for the 74244 buffer.

Table C-3: Advanced Logic General Characteristics

Family	Year	Number Suppliers	Tech Base	I/O Level	Speed (ns)	Static Current	I_{OH}/I_{OL}
ACQ	1989	2	CMOS	CMOS/CMOS	6.0	80 μA	−24/24 mA
ACTQ	1989	2	CMOS	TTL/CMOS	7.5	80 μA	−24/24 mA
FCTx	1987	3	CMOS	TTL/CMOS	4.1–4.8	1.5 mA	−15/64 mA
FCTxT	1990	2	CMOS	TTL/TTL	4.1–4.8	1.5 mA	−15/64 mA
FASTr	1990	1	Bipolar	TTL/TTL	3.9	50 mA	−15/64 mA
BCT	1987	2	BICMOS	TTL/TTL	5.5	10 mA	−15/64 mA

Reprinted by permission of Electronic Design Magazine, c. 1991.

Since the late 1970s, the use of a +5 V power supply has become standard in all microprocessors and microcontrollers. To reduce power consumption, 3.3 V V_{CC} is being embraced by many designers. The lowering of V_{CC} to 3.3 V has two major advantages: (1) It lowers the power consumption, prolonging the life of the battery in systems using a battery, and (2) it allows a further reduction of line size (design rule) to submicron dimensions. This reduction results in putting more transistors in a given die size. As fabrication processes improve, the decline in the line size is reaching submicron level and transistor densities are approaching 1 billion transistors.

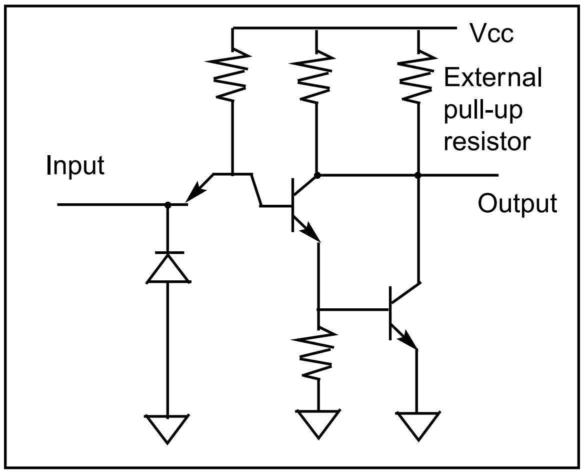


Figure C-5. Open Collector

Open-collector and open-drain gates

To allow multiple outputs to be connected together, we use open-collector logic gates. In such cases, an external resistor will serve as load. This is shown in Figures C-5 and C-6.

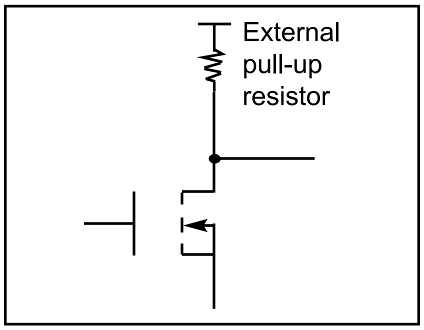


Figure C-6. Open Drain

SECTION C.2: AVR I/O PORT STRUCTURE AND INTERFACING

In interfacing the AVR microcontroller with other IC chips or devices, fan-out is the most important issue. To understand the AVR fan-out we must first understand the port structure of the AVR. This section provides a detailed discussion of the AVR port structure and its fan-out. It is very critical that we understand the I/O port structure of the AVR lest we damage it while trying to interface it with an external device.

IC fan-out

When connecting IC chips together, we need to find out how many input pins can be driven by a single output pin. This is a very important issue and involves the discussion of what is called *IC fan-out*. The IC fan-out must be addressed for both logic "0" and logic "1" outputs. See Example C-1. Fan-out for logic LOW and fan-out for logic HIGH are defined as follows:

$$\text{fan-out (of LOW)} = \frac{I_{OL}}{I_{IL}} \qquad \text{fan-out (of HIGH)} = \frac{I_{OH}}{I_{IH}}$$

Of the above two values, the lower number is used to ensure the proper noise margin. Figure C-7 shows the sinking and sourcing of current when ICs are connected together.

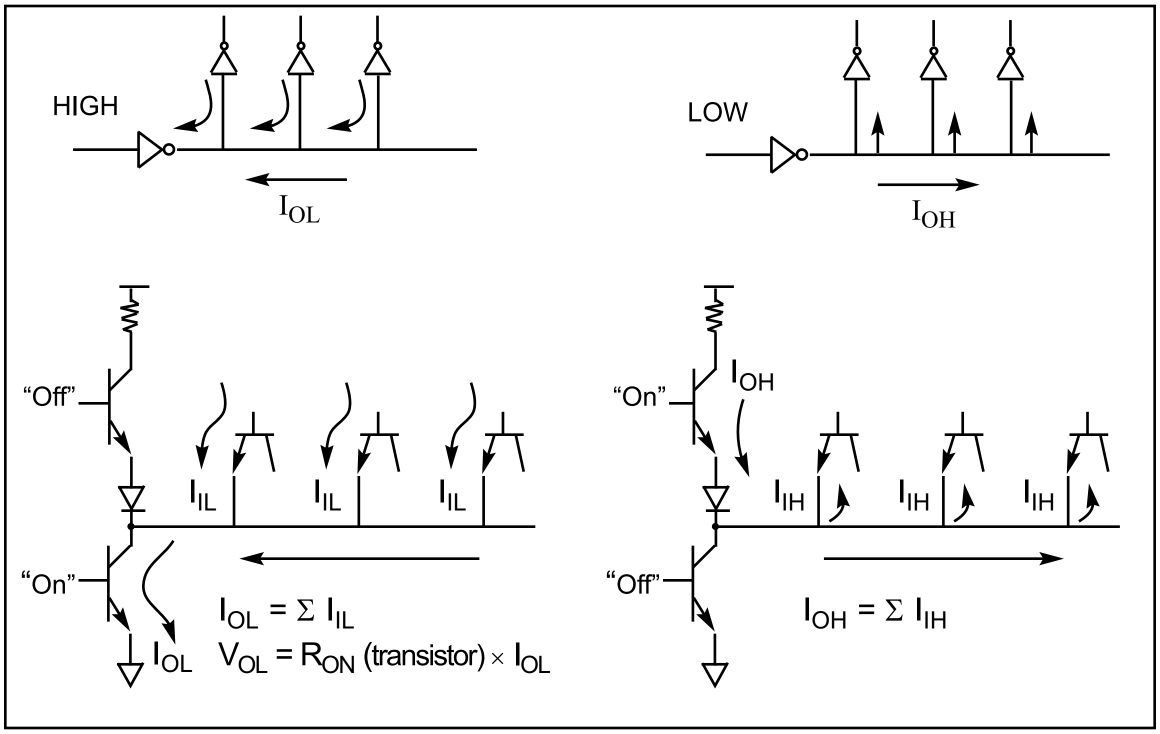


Figure C-7. Current Sinking and Sourcing in TTL

Notice that in Figure C-7, as the number of input pins connected to a single output increases, I_{OL} rises, which causes V_{OL} to rise. If this continues, the rise of V_{OL} makes the noise margin smaller, and this results in the occurrence of false logic due to the slightest noise.

Example C-1

Find how many unit loads (UL) can be driven by the output of the LS logic family.

Solution:

The unit load is defined as I_{IL} = 1.6 mA and I_{IH} = 40 μA. Table C-1 shows I_{OH} = 400 μA and I_{OL} = 8 mA for the LS family. Therefore, we have

$$\text{fan-out (LOW)} = \frac{I_{OL}}{I_{IL}} = \frac{8\text{ mA}}{1.6\text{ mA}} = 5$$

$$\text{fan-out (HIGH)} = \frac{I_{OH}}{I_{IH}} = \frac{400\ \mu\text{A}}{40\ \mu\text{A}} = 10$$

This means that the fan-out is 5. In other words, the LS output must not be connected to more than 5 inputs with unit load characteristics.

74LS244 and 74LS245 buffers/drivers

In cases where the receiver current requirements exceed the driver's capability, we must use buffers/drivers such as the 74LS245 and 74LS244. Figure C-8 shows the internal gates for the 74LS244 and 74LS245. The 74LS245 is used for bidirectional data buses, and the 74LS244 is used for unidirectional address buses.

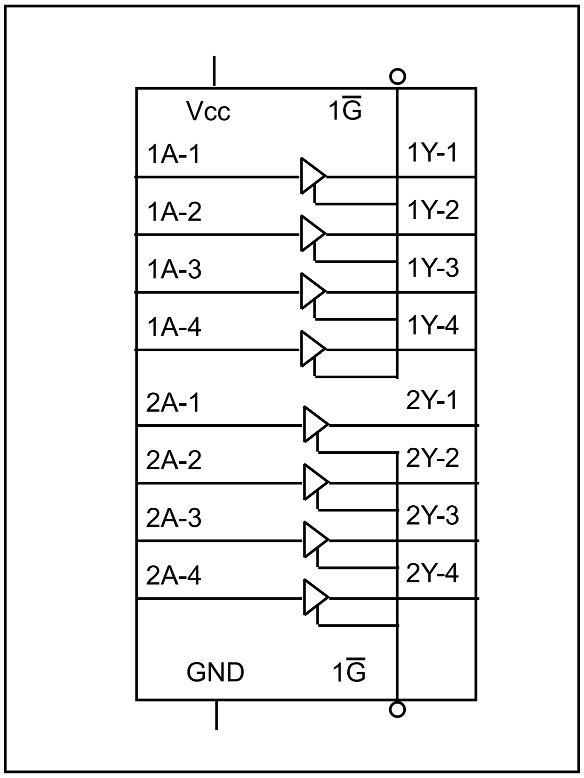
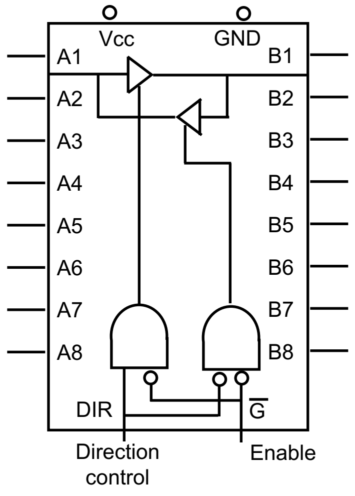


Figure C-8 (a). 74LS244 Octal Buffer

(Reprinted by permission of Texas Instruments, Copyright Texas Instruments, 1988)



Function Table

Enable $\overline{G}$	Direction control DIR	Operation
L	L	B Data to A Bus
L	H	A Data to B Bus
H	X	Isolation

Figure C-8 (b). 74LS245 Bidirectional Buffer

(Reprinted by permission of Texas Instruments, Copyright Texas Instruments, 1988)

Tri-state buffer

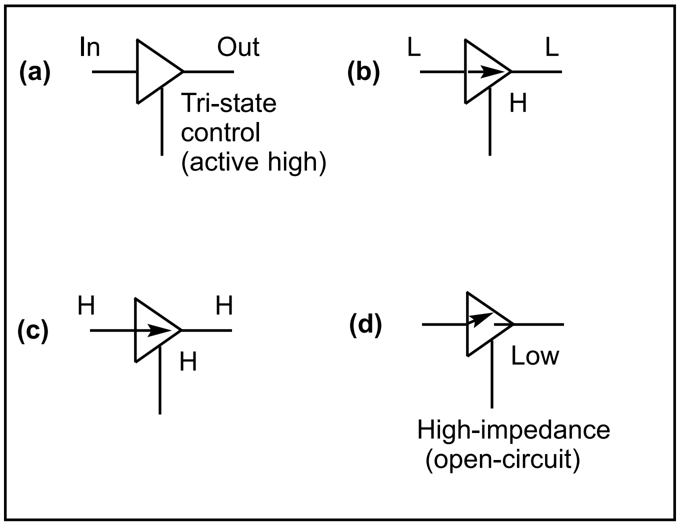

(a)
In
Out
Tri-state control (active high)
(b)
L
L
H
(c)
H
H
H
(d)
Low
High-impedance (open-circuit)

Figure C-9. Tri-State Buffer

Notice that the 74LS244 is simply 8 tri-state buffers in a single chip. As shown in Figure C-9 a tri-state buffer has a single input, a single output, and the enable control input. By activating the enable, data at the input is transferred to the output. The enable can be an active-LOW or an active-HIGH. Notice that the enable input for the 74LS244 is an active-LOW whereas the enable input pin for Figure C-9 is active-HIGH.

74LS245 and 74LS244 fan-out

It must be noted that the output of the 74LS245 and 74LS244 can sink and source a much larger amount of current than that of other LS gates. See Table C-4. That is the reason we use these buffers for driver when a signal is travelling a long distance through a cable or it has to drive many inputs.

Table C-4: Electrical Specifications for Buffers/Drivers

	I_{OH} (mA)	I_{OL} (mA)
74LS244	3	12
74LS245	3	12

After this background on the fan-out, next we discuss the structure of AVR ports.

AVR port structure and operation

All the ports of the AVR are bidirectional. They all have three registers that can be accessed by IN and OUT instructions. We will descuss each register in detail.

PORTx register

As you can see in Figure C-10, the PORTx register can be accessed using read and write operations. When we want to write to PORTx, we use the "OUT PORTx, Rr" instruction. In this case, the WR-PORTx pin is set high and Rr is loaded into PORTx.

When we want to read from PORTx, we use "IN Rd, PORTx". In this case, the PRx pin is set to HIGH, which enables the buffer and makes it possible to read from PORTx.

The output of PORTx is either connected to the Px pin of the chip or con-

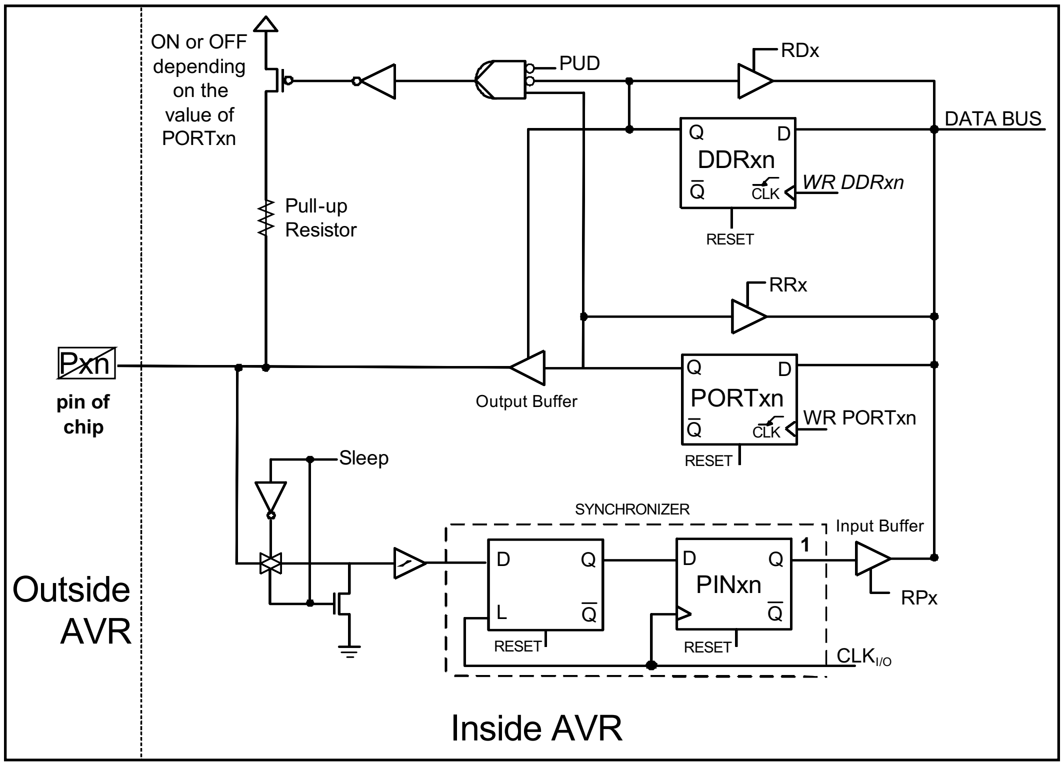


Figure C-10. The AVR Ports Structure

trols the pull-up resistor, as we will see next.

DDRx register

As shown in Figure C-10, the DDRx register can be accessed using read and write operations. When we want to write to DDRx, we use "OUT DDRx, Rr". In this case, the WR-DDRx pin is set to HIGH and enables writing to DDRx. When we want to read from DDRx, we use "IN Rd, DDRx". In this case, the RDx pin is set to LOW, which enables the buffer and makes it possible to read from DDRx.

The DDRx register controls the output buffer and the pull-up resistor. When the Q of DDRx is HIGH, it enables the output buffer and connects the Q of the PORTx register to the Px pin of the chip. In this case, the pin is configured as output. When the Q of DDRx is LOW, it disables the output buffer and configures the Px pin of the chip as input. In this case, assuming that the PUD bit is LOW, the Q of PORTx controls the pull-up resistor. When the Q of PORTx is HIGH, it enables the pull-up resistor, and when it is LOW, it disables the pull-up resistor.

PINx register

As you see in Figure C-10, when the AVR is not in sleep mode, the PINxn flip-flop is loaded with the value of the AVR pin on each machine cycle. Therefore, to read the current state of the Px pin of the chip, we should read the content of the PINx register. To do so, we use "IN Rd,PINx", which sets RPx high and enables the input buffer. In this case, the value of PINx passes through the internal data bus of AVR and will be loaded into the Rd register.

Reading the pin when DDRx.n = 0 (Input)

As we stated in Chapter 4, to make any bits of any port of the AVR an input port, we first must write a 0 (logic LOW) to the DDRx.n bit. Look at the following sequence of events to see why:

1. As can be seen from Figure C-11, if we write 0 to the DDRx.n, it will have "LOW" on its Q. This turns off the tri-state buffer.
2. When the tri-state buffer is off, it blocks the path from the Q of PORTx.n to the pin of chip, and the input signal is directed to the PINx.n buffer.
3. When reading the input port in instructions such as "`IN R16,PINB`" we are reading the data present at the pin. In other words, it is bringing into the CPU the status of the external pin. This instruction activates the read pin of the buffer and lets data at the pins flow into the CPU's internal bus. Figure C-11 shows how the input circuit works.

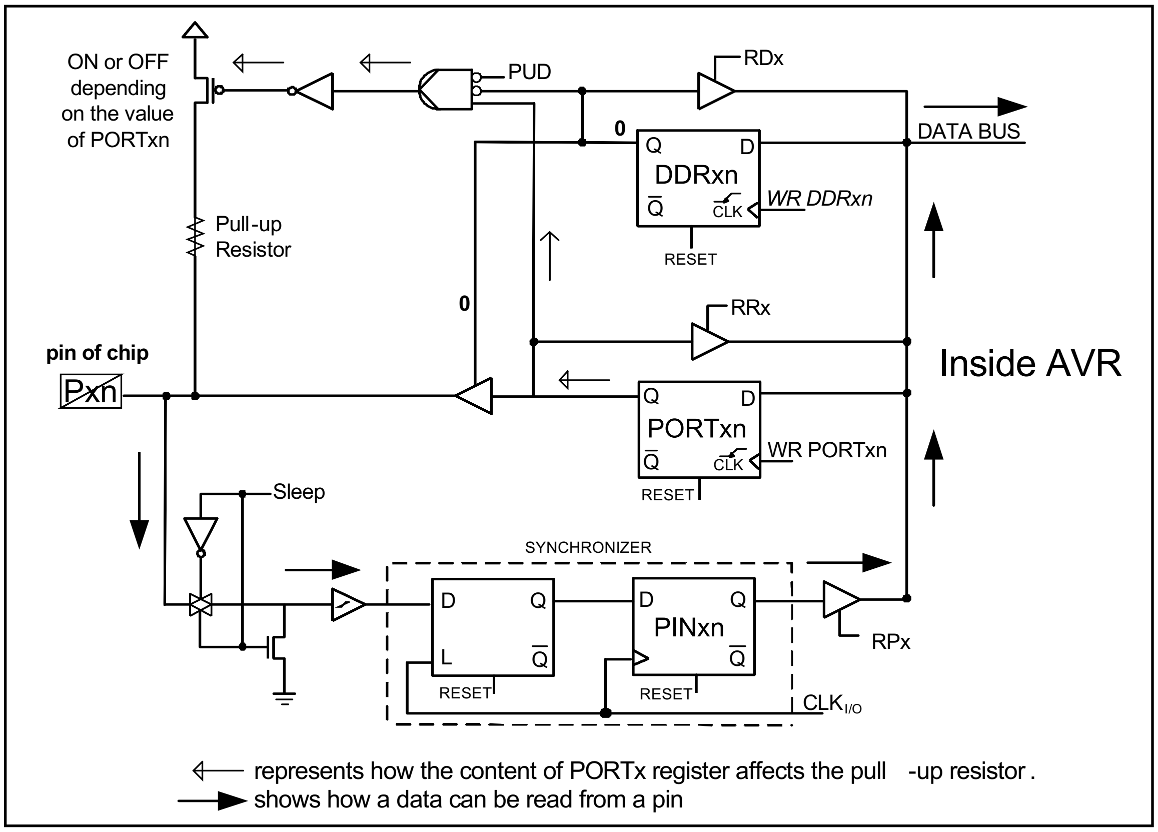


Figure C-11. Inputting (Reading) from a Pin via a PINx Register in the AVR

Writing to pin when DDRx.n = 1 (Output)

The above discussion showed why we must write a "LOW" to a port's DDRx.n bits in order to make it an input port. What happens if we write a "1" to DDRx.n that was configured as an input port? From Figure C-12 we see that when DDRx.n = 1, the DDRx.n latch has "HIGH" on its Q. This turns on the tri-state buffer, and the data of PORTx.n is transferred to the pin of chip.

From Figure C-12 we see that when DDRx.n = 1, if we write a 0 to the PORTx.n latch, then PORTx.n has "LOW" on its Q. This provides 0 to the pin of chip. Therefore, any attempt to read the input pin will always get the "LOW" ground signal. Figure C-13 shows what happens if we write "HIGH" to PORTx.n when DDRx.n = 1. Writing 1 to the PORTx.n makes Q = 1. As a result, a 1 is provided to the pin of the chip. Therefore, any attempt to read the input pin will always get the "HIGH" signal.

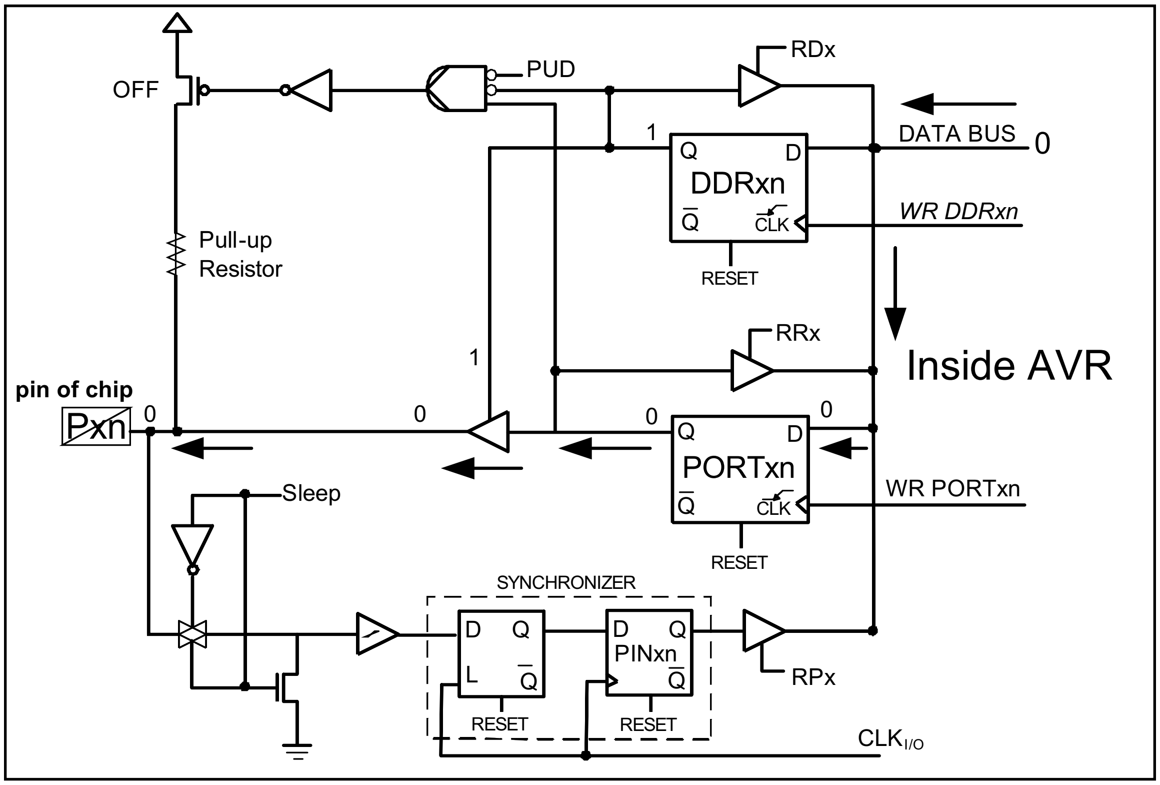


Figure C-12. Outputting (Writing) 0 to a Pin in the AVR

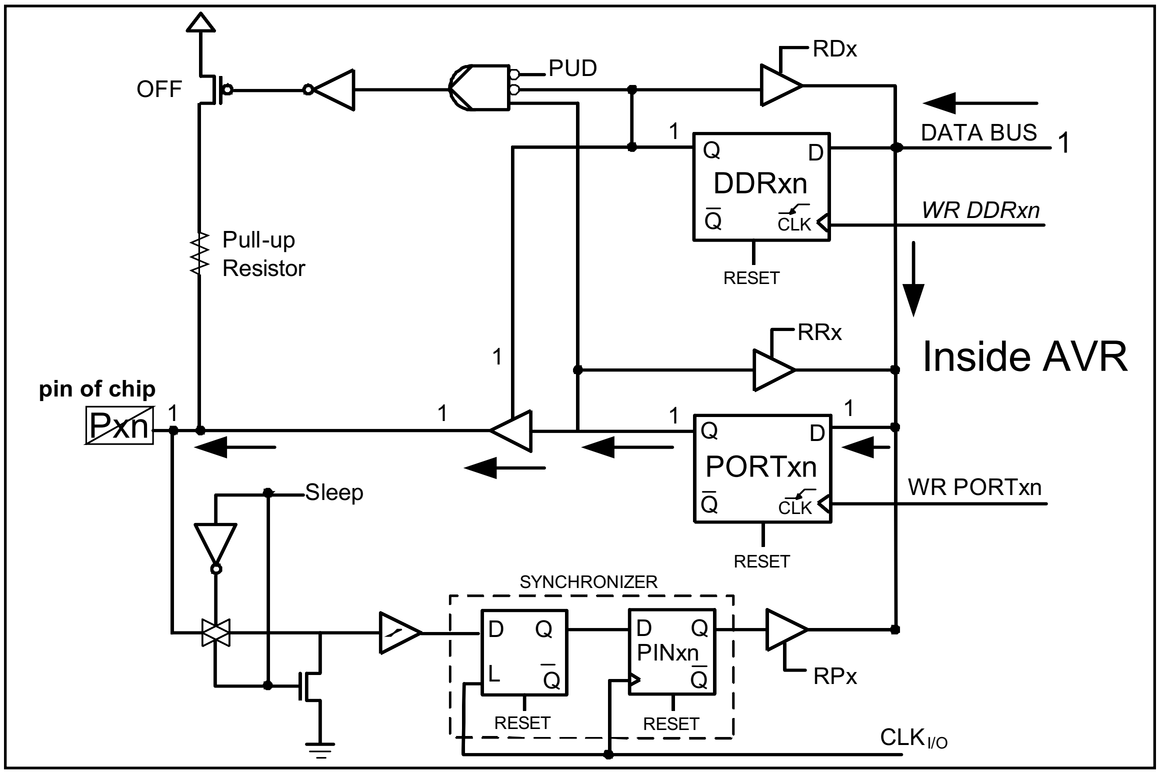


Figure C-13. Outputting (Writing) 1 to a Pin in the AVR

Notice that we should not make an I/O port output while it is externally connected to a voltage; otherwise, we might damage the ports.

For example, see Figure C-14. In this program, the PORTB.3 is mistakenly set as output. When the key is closed, the pin will be directly connected to ground while the AVR is trying to send out high. As a result, the AVR will be damaged when the key is closed. Also, the program will not work properly, as it will always read high while trying to read the pin.

The above points are extremely important and must be emphasized because many people damage their ports and afterwards wonder how it happened. We must also use the right instruction when we want to read the status of an input pin.

```
.INCLUDE "M32DEF.INC"

      SBI   DDRB, 3  ;PB3 as output
;Note: Since PB3 is connected to a
;switch it cannot be configured as
;output
      SBI   PORTB, 3 ;PB3 = high
HERE: SBIC  PINB, 3
      RJMP  HERE  ;stay in the loop
      ...
```

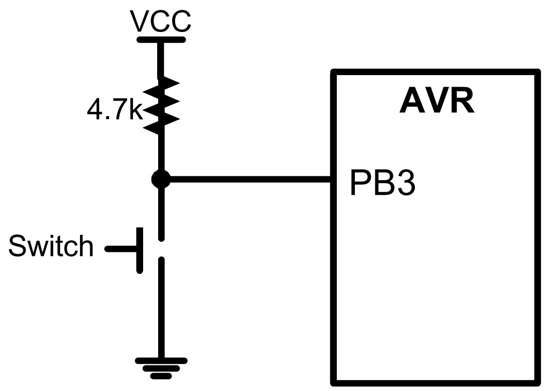


Figure C-14. A Common Mistake, Which Damages I/O Ports

AVR port fan-out

Now that we are familiar with the port structure of the AVR, we need to examine the fan-out for the AVR microcontroller. AVR microcontrollers are all based on CMOS technology. Note, however, that while the core of the AVR microcontroller is CMOS, the circuitry driving its pins is all TTL compatible. That is, the AVR is a CMOS-based product with TTL-compatible pins. Table C-5 provides the I/O characteristics of AVR ports.

Table C-5: Fan-out for AVR Ports

Pin	Fan-out
IOL	20 mA
IOH	−20 mA
IIL	−1 μA
IIH	1 μA

Note: Negative current is defined as current sourced by the pin.

SECTION C.3: SYSTEM DESIGN ISSUES

In addition to fan-out, the other issues related to system design are power dissipation, ground bounce, V_{CC} bounce, crosstalk, and transmission lines. In this section we provide an overview of these topics.

Power dissipation considerations

Power dissipation is a major concern of system designers, especially for laptop and hand-held systems in which batteries provide the power. Power dissipation is a function of frequency and voltage as shown below:

$$Q = CV$$

$$\frac{Q}{T} = \frac{CV}{T}$$

$$\textit{since} \quad F = \frac{1}{T} \qquad \textit{and} \quad I = \frac{Q}{T}$$

$$I = CVF$$

$$\textit{now} \quad P = VI = CV^2F$$

In the above equations, the effects of frequency and V_{CC} voltage should be noted. While the power dissipation goes up linearly with frequency, the impact of the power supply voltage is much more pronounced (squared). See Example C-2.

Example C-2

Compare the power consumption of two microcontroller-based systems. One uses 5 V and the other uses 3 V for V_{CC}.

Solution:
Because P = VI, by substituting I = V/R we have $P = V^2/R$. Assuming that R = 1, we have $P = 5^2 = 25$ W and $P = 3^2 = 9$ W. This results in using 16 W less power, which means power saving of 64% ($16/25 \times 100$) for systems using a 3 V power source.

Dynamic and static currents

Two major types of currents flow through an IC: dynamic and static. A dynamic current is I = CVF. It is a function of the frequency under which the component is working. This means that as the frequency goes up, the dynamic current and power dissipation go up. The static current, also called DC, is the current consumption of the component when it is inactive (not selected). The dynamic current dissipation is much higher than the static current consumption. To reduce power consumption, many microcontrollers, including the AVR, have power-saving modes. In the AVR, the power saving mode is called *sleep mode.* We describe the sleep mode next.

Sleep mode

In sleep mode the clocks of the CPU and some peripheral functions, such as serial ports, interrupts, and timers, are cut off. This brings power consumption down to an absolute minimum, while the contents of RAM and the SFR registers are saved and remain unchanged. The AVR provides six different sleeping modes, which enable you to choose which units will sleep. For more information see the AVR datasheets.

Ground bounce

One of the major issues that designers of high-frequency systems must grapple with is ground bounce. Before we define ground bounce, we will discuss lead inductance of IC pins. There is a certain amount of capacitance, resistance, and inductance associated with each pin of the IC. The size of these elements varies depending on many factors such as length, area, and so on.

The inductance of the pins is commonly referred to as *self-inductance* because there is also what is called *mutual inductance*, as we will show below. Of the three components of capacitor, resistor, and inductor, the property of self-inductance is the one that causes the most problems in high-frequency system design because it can result in ground bounce. Ground bounce occurs when a massive amount of current flows through the ground pin caused by many outputs changing from HIGH to LOW all at the same time. See Figure C-15 (a). The voltage is related to the inductance of the ground lead as follows:

$$V = L\frac{di}{dt}$$

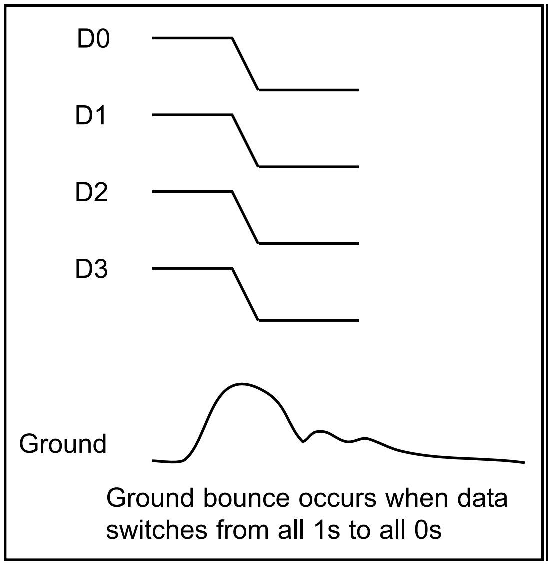
As we increase the system frequency, the rate of dynamic current, di/dt, is also increased, resulting in an increase in the inductance voltage L (di/dt) of the ground pin. Because the LOW state (ground) has a small noise margin, any extra voltage due to the inductance can cause a false signal. To reduce the effect of ground bounce, the following steps must be taken where possible:

1. The V_{CC} and ground pins of the chip must be located in the middle rather than at opposite ends of the IC chip (the 14-pin TTL logic IC uses pins 14 and 7 for ground and V_{CC}). This is exactly what we see in high-performance logic gates such as Texas Instruments' advanced logic AC11000 and ACT11000 families. For example, the ACT11013 is a 14-pin DIP chip in which pin numbers 4 and 11 are used for the ground and V_{CC}, instead of 7 and 14 as in the traditional TTL family. We can also use the SOIC packages instead of DIP.
2. Another solution is to use as many pins for ground and V_{CC} as possible to reduce the lead length. This is exactly why all high-performance microprocessors and logic families use many pins for V_{CC} and ground instead of the traditional single pin for V_{CC} and single pin for GND. For example, in the case of Intel's Pentium processor there are over 50 pins for ground, and another 50 pins for V_{CC}.

The above discussion of ground bounce is also applicable to V_{CC} when a large number of outputs changes from the LOW to the HIGH state; this is referred to as *V_{CC} bounce*. However, the effect of V_{CC} bounce is not as severe as ground bounce because the HIGH ("1") state has a wider noise margin than the LOW ("0") state.

Filtering the transient currents using decoupling capacitors

In the TTL family, the change of the output from LOW to HIGH can cause what is called *transient current*. In a totem-pole output in which the output is LOW, Q4 is on and saturated, whereas Q3 is off. By changing the output from the



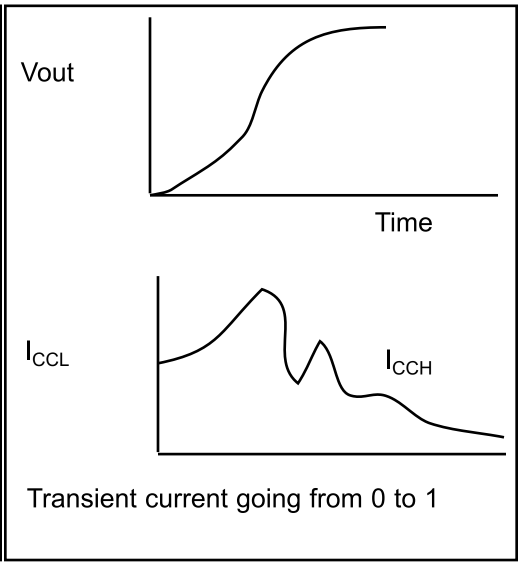


Figure C-15. (a) Ground Bounce **Figure C-15. (b) Transient Current**

LOW to the HIGH state, Q3 turns on and Q4 turns off. This means that there is a time when both transistors are on and drawing current from V_{CC}. The amount of current depends on the R_{ON} values of the two transistors, which in turn depend on the internal parameters of the transistors. The net effect of this, however, is a large amount of current in the form of a spike for the output current, as shown in Figure C-15 (b). To filter the transient current, a 0.01 μF or 0.1 μF ceramic disk capacitor can be placed between the V_{CC} and ground for each TTL IC. The lead for this capacitor, however, should be as small as possible because a long lead results in a large self-inductance, and that results in a spike on the V_{CC} line [V = L (di/dt)]. This spike is called *V_{CC} bounce*. The ceramic capacitor for each IC is referred to as a *decoupling capacitor*. There is also a bulk decoupling capacitor, as described next.

Bulk decoupling capacitor

If many IC chips change state at the same time, the combined currents drawn from the board's V_{CC} power supply can be massive and may cause a fluctuation of V_{CC} on the board where all the ICs are mounted. To eliminate this, a relatively large decoupling tantalum capacitor is placed between the V_{CC} and ground lines. The size and location of this tantalum capacitor vary depending on the number of ICs on the board and the amount of current drawn by each IC, but it is common to have a single 22 μF to 47 μF capacitor for each of the 16 devices, placed between the V_{CC} and ground lines.

Crosstalk

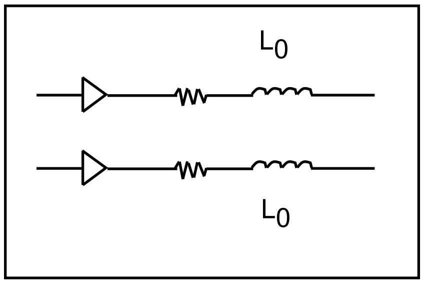


Figure C-16. Crosstalk (EMI)

Crosstalk is due to mutual inductance. See Figure C-16. Previously, we discussed self-inductance, which is inherent in a piece of conductor. *Mutual inductance* is caused by two electric lines running parallel to each other. The mutual inductance is a function of l, the length of two conductors running in parallel; d, the distance between them; and the medium material placed between them. The effect of crosstalk can be reduced by increasing the distance between the parallel or adjacent lines (in printed circuit boards, they will be traces). In many cases, such as printer and disk drive cables, there is a dedicated ground for each signal. Placing ground lines (traces) between signal lines reduces the effect of crosstalk. (This method is used even in some ACT logic families where a V_{CC} and a GND pin are next to each other.) Crosstalk is also called *EMI* (electromagnetic interference). This is in contrast to *ESI* (electrostatic interference), which is caused by capacitive coupling between two adjacent conductors.

Transmission line ringing

The square wave used in digital circuits is in reality made of a single fundamental pulse and many harmonics of various amplitudes. When this signal travels on the line, not all the harmonics respond in the same way to the capacitance, inductance, and resistance of the line. This causes what is called *ringing*, which depends on the thickness and the length of the line driver, among other factors. To reduce the effect of ringing, the line drivers are terminated by putting a resistor at the end of the line. See Figure C-17. There are three major methods of line driver termination: parallel, serial, and Thevenin.

In serial termination, resistors of 30–50 ohms are used to terminate the line. The parallel and Thevenin methods are used in cases where there is a need to match the impedance of the line with the load impedance. This requires a detailed analysis of the signal traces and load impedance, which is beyond the scope of this book. In high-frequency systems, wire traces on the printed circuit board (PCB) behave like transmission lines, causing ringing. The severity of this ringing depends on the speed and the logic family used. Table C-6 provides the trace length, beyond which the traces must be looked at as transmission lines.

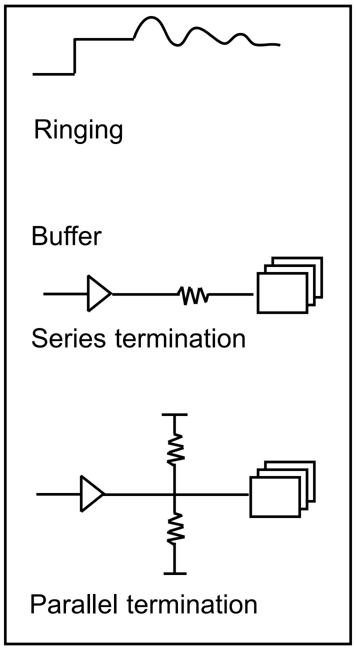


Figure C-17. Reducing Transmission Line Ringing

Table C-6: Line Length Beyond Which Traces Behave Like Transmission Lines

Logic Family	Line Length (in.)
LS	25
S, AS	11
F, ACT	8
AS, ECL	6
FCT, FCTA	5

(Reprinted by permission of Integrated Device Technology, copyright IDT 1991)

APPENDIX D

FLOWCHARTS AND PSEUDOCODE

OVERVIEW

This appendix provides an introduction to writing flowcharts and pseudocode.

Flowcharts

If you have taken any previous programming courses, you are probably familiar with flowcharting. Flowcharts use graphic symbols to represent different types of program operations. These symbols are connected together into a flowchart to show the flow of execution of a program. Figure D-1 shows some of the more commonly used symbols. Flowchart templates are available to help you draw the symbols quickly and neatly.

Pseudocode

Flowcharting has been standard practice in industry for decades. However, some find limitations in using flowcharts, such as the fact that you can't write much in the little boxes, and it is hard to get the “big picture” of what the program does without getting bogged down in the details. An alternative to using flowcharts is pseudocode, which involves writing brief descriptions of the flow of the code. Figures D-2 through D-6 show flowcharts and pseudocode for commonly used control structures.

Structured programming uses three basic types of program control structures: sequence, control, and itera-

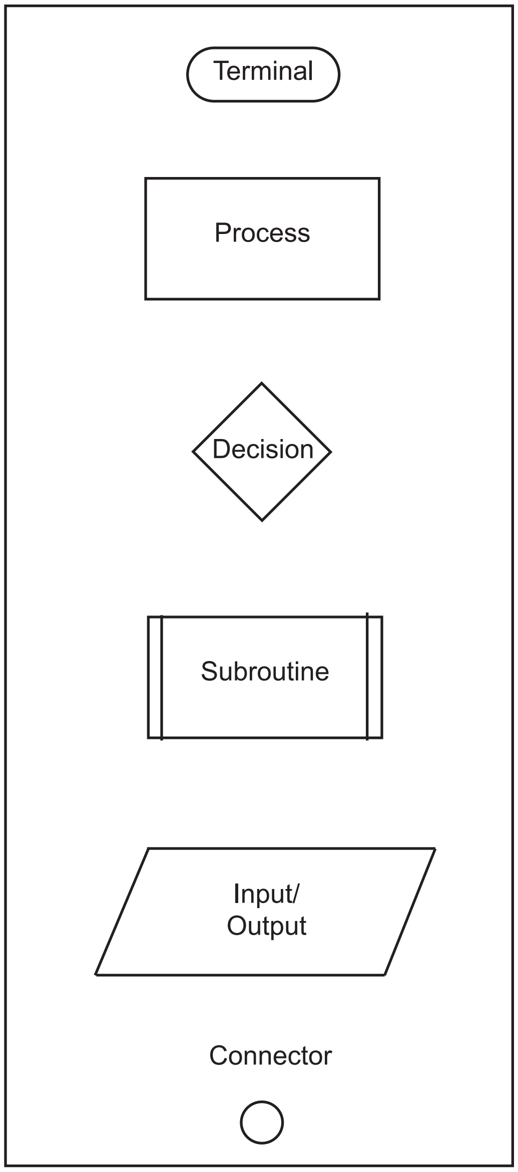


Figure D-1. Commonly Used Flowchart Symbols

```
Statement 1
Statement 2
```

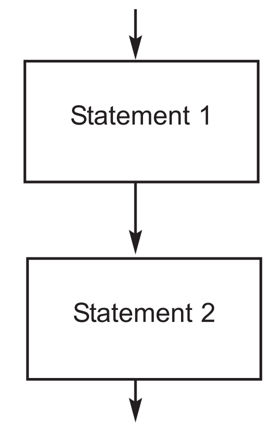


Figure D-2. SEQUENCE Pseudocode versus Flowchart

tion. Sequence is simply executing instructions one after another. Figure D-2 shows how sequence can be represented in pseudocode and flowcharts.

Figures D-3 and D-4 show two control programming structures: IF-THEN-ELSE and IF-THEN in both pseudocode and flowcharts.

Note in Figures D-2 through D-6 that "statement" can indicate one statement or a group of statements.

Figures D-5 and D-6 show two iteration control structures: REPEAT UNTIL and WHILE DO. Both structures execute a statement or group of statements repeatedly. The difference between them is that the REPEAT UNTIL structure always executes the statement(s) at least once, and checks the condition after each iteration, whereas the WHILE DO may not execute the statement(s) at all because the condition is checked at the beginning of each iteration.

```
IF (condition) THEN
     Statement 1
ELSE
     Statement 2
```

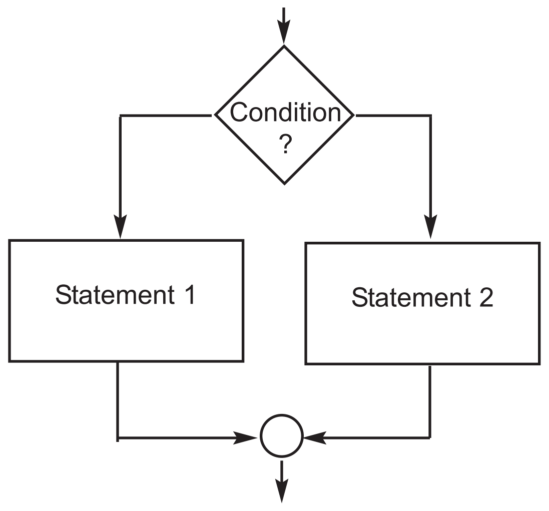


Figure D-3. IF THEN ELSE Pseudocode versus Flowchart

```
IF (condition) THEN
     Statement
```

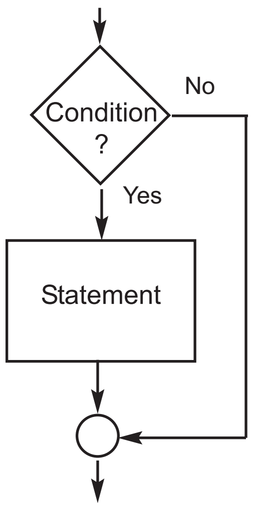


Figure D-4. IF THEN Pseudocode versus Flowchart

```
REPEAT
     Statement
UNTIL (condition)
```

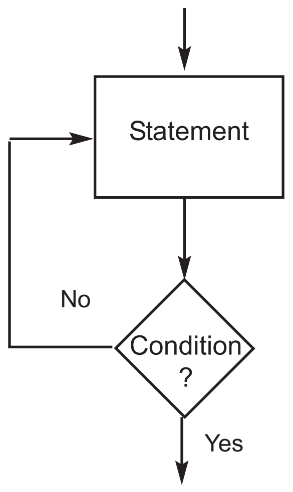


Figure D-5. REPEAT UNTIL Pseudocode versus Flowchart

```
WHILE (condition) DO
     Statement
```

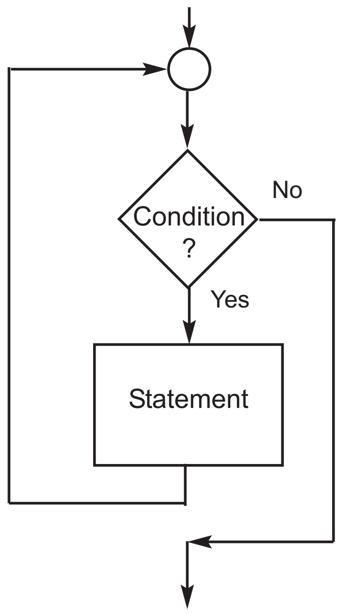


Figure D-6. WHILE DO Pseudocode versus Flowchart

Program D-1 finds the sum of a series of bytes. Compare the flowchart versus the pseudocode for Program D-1 (shown in Figure D-7). In this example, more program details are given than one usually finds. For example, this shows steps for initializing and decrementing counters. Another programmer may not include these steps in the flowchart or pseudocode. It is important to remember that the purpose of flowcharts or pseudocode is to show the flow of the program and what the program does, not the specific Assembly language instructions that accomplish the program's objectives. Notice also that the pseudocode gives the same information in a much more compact form than does the flowchart. It is important to note that sometimes pseudocode is written in layers, so that the outer level or layer shows the flow of the program and subsequent levels show more details of how the program accomplishes its assigned tasks.

```
Count = 5
Address = $140
Repeat
     Add next byte
     Increment address
     Decrement counter
Until Count = 0

Store Sum
```

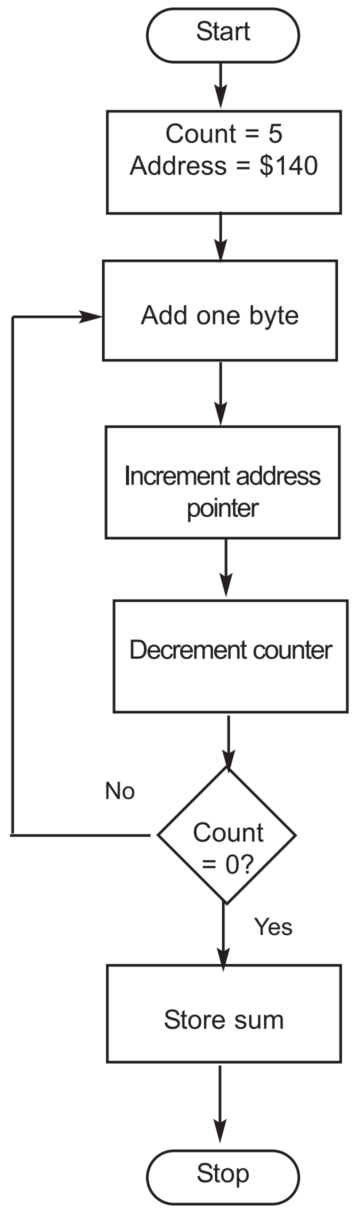


Figure D-7. Pseudocode versus Flowchart for Program D-1

```
#define      COUNTVAL    5      ;COUNT = 5
#define      COUNTER     R22
#define      SUM         R23
      LDI    COUNTER,COUNTVAL  ;R22 = 5
      CLR    SUM               ;SUM = 0
      LDI    R26,LOW($140)     ;load pointer to RAM data address
      LDI    R27,HIGH($140)
L1:   LD     R24,x+            ;copy RAM to R24 and increment pointer
      ADD    SUM,R24           ;add R24 to SUM
      DEC    COUNTER           ;decrement counter
      BRNE   L1                ;loop until counter = zero
HERE: RJMP   HERE              ;stay here forever
```

Program D-1

APPENDIX E

AVR PRIMER FOR 8051 PROGRAMMERS

	AVR	8051
8-bit registers:	32 general-purpose registers (R0 to R31)	A, B, R0, R1, R2, R3, R4, R5, R6, R7
16-bit (data pointer):	X, Y, Z	DPTR
Program Counter:	PC (up to 22-bit)	PC (16-bit)
Input:		
	`IN   Rn,PINx`	`MOV A,Pn ;(n = 0 - 3)`
	(Use R0, R1, ..., R31.)	
Output:		
	`OUT  PORTx,Rn`	`MOV Pn,A ;(n = 0 - 3)`
Loop:		
	`DEC  Rn`	`DJNZ R3,TARGET`
	`BRNE TARGET`	`(Using R0-R7)`
Stack pointer:		
	SP (16-bit)	SP (8-bit)
	As we PUSH data onto the stack, it decrements the SP.	As we PUSH data onto the stack, it increments the SP.
	As we POP data from the stack, it increments the SP.	As we POP data from the stack, it decrements the SP.
Data movement:		
From the code segment:		
	`LPM  Rn,Z`	`MOVC A,@A+PC`
	(Use Z only.)	
From RAM using indirect addressing:		
	`LD   Rn,X`	`MOV A,@R0`
	(Use X, Y, or Z.)	(Use R0 or R1 only.)
From RAM using direct addressing:		
	`LDS  Rn,k`	`MOV A,RAM_addr`
To RAM using indirect addressing mode:		
	`ST   X,Rn`	`MOV @R0,A`
	(Use X, Y, or Z.)	
To RAM using direct addressing mode:		
	`STS  k,X`	`MOV RAM_addr,A`
	(Use X, Y, or Z.)	

APPENDIX F

ASCII CODES

Ctrl	Dec	Hex	Ch	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Ch
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Ch
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Ch
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	⌂

Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	β
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	?	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	≈
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	·
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

APPENDIX G

ASSEMBLERS, DEVELOPMENT RESOURCES, AND SUPPLIERS

This appendix provides various sources for AVR assemblers, compilers, and trainers. In addition, it lists some suppliers for chips and other hardware needs. While these are all established products from well-known companies, neither the author nor the publisher assumes responsibility for any problem that may arise with any of them. You are neither encouraged nor discouraged from purchasing any of the products mentioned; you must make your own judgment in evaluating the products. This list is simply provided as a service to the reader. It also must be noted that the list of products is by no means complete or exhaustive.

The AVR Studio from Atmel
http://www.atmel.com

MicroC from mikroElectronika
http://www.mikroe.com

CodeVision
http://www.hpinfotech.ro

ImageCraft
http://www.imagecraft.com

Micro IDE
http://www.micro-ide.com

Figure G-1. Suppliers of Assemblers and Compilers

AVR assemblers

The AVR assembler is provided by Atmel and other companies. Some of the companies provide shareware versions of their products, which you can download from their websites. However, the size of code for these shareware versions is limited to a few KB. Figure G-1 lists some suppliers of assemblers and compilers.

AVR trainers

There are many companies that produce and market AVR trainers. Figure G-2 provides a list of some of them.

MicroDigitalEd
http://www.MicroDigitalEd.com

Digilent
http://www.digilentinc.com

Atmel
http://www.atmel.com

Figure G-2. Trainer Suppliers

Parts suppliers

Figure G-3 provides a list of suppliers for many electronics parts.

RSR Electronics
Electronix Express
365 Blair Road
Avenel, NJ 07001
Fax: (732) 381-1572
Mail Order: 1-800-972-2225
In New Jersey: (732) 381-8020
http://www.elexp.com

Altex Electronics
11342 IH-35 North
San Antonio, TX 78233
Fax: (210) 637-3264
Mail Order: 1-800-531-5369
http://www.altex.com

Digi-Key
1-800-344-4539 (1-800-DIGI-KEY)
Fax: (218) 681-3380
http://www.digikey.com

Radio Shack
http://www.radioshack.com

JDR Microdevices
1850 South 10th St.
San Jose, CA 95112-4108
Sales 1-800-538-5000
(408) 494-1400
Fax: 1-800-538-5005
Fax: (408) 494-1420
http://www.jdr.com

Mouser Electronics
958 N. Main St.
Mansfield, TX 76063
1-800-346-6873
http://www.mouser.com

Jameco Electronic
1355 Shoreway Road
Belmont, CA 94002-4100
1-800-831-4242
(415) 592-8097
Fax: 1-800-237-6948
Fax: (415) 592-2503
http://www.jameco.com

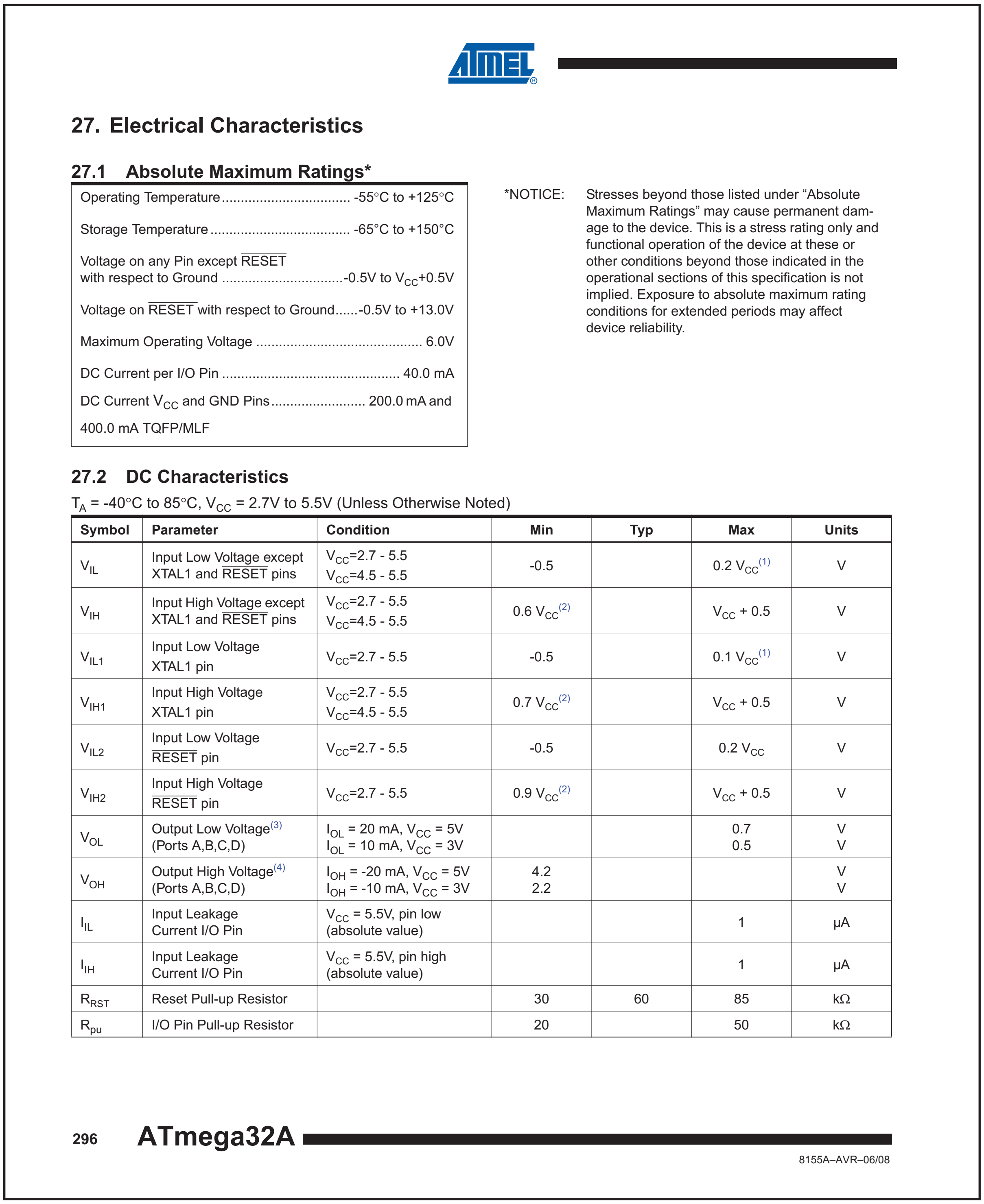
B. G. Micro
P. O. Box 280298
Dallas, TX 75228
1-800-276-2206 (orders only)
(972) 271-5546
Fax: (972) 271-2462
This is an excellent source of LCDs, ICs, keypads, etc.
http://www.bgmicro.com

Tanner Electronics
1100 Valwood Parkway, Suite #100
Carrollton, TX 75006
(972) 242-8702
http://www.tannerelectronics.com

Figure G-3. Electronics Suppliers

APPENDIX H

DATA SHEETS



27. Electrical Characteristics

27.1 Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on any Pin except $\overline{\text{RESET}}$ with respect to Ground	-0.5V to V_{CC}+0.5V
Voltage on $\overline{\text{RESET}}$ with respect to Ground	-0.5V to +13.0V
Maximum Operating Voltage	6.0V
DC Current per I/O Pin	40.0 mA
DC Current V_{CC} and GND Pins	200.0 mA and 400.0 mA TQFP/MLF

*NOTICE: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

27.2 DC Characteristics

T_A = -40°C to 85°C, V_{CC} = 2.7V to 5.5V (Unless Otherwise Noted)

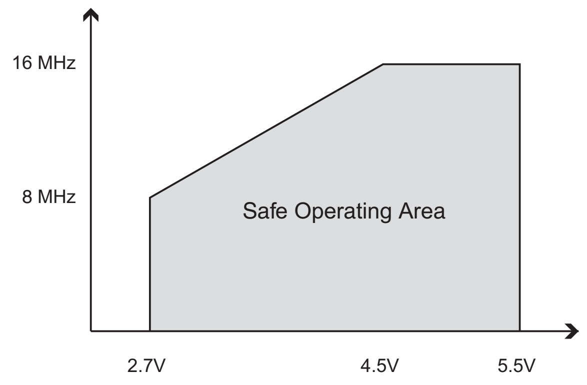
Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{IL}	Input Low Voltage except XTAL1 and $\overline{\text{RESET}}$ pins	V_{CC}=2.7 - 5.5 V_{CC}=4.5 - 5.5	-0.5		0.2 V_{CC}(1)	V
V_{IH}	Input High Voltage except XTAL1 and $\overline{\text{RESET}}$ pins	V_{CC}=2.7 - 5.5 V_{CC}=4.5 - 5.5	0.6 V_{CC}(2)		V_{CC} + 0.5	V
V_{IL1}	Input Low Voltage XTAL1 pin	V_{CC}=2.7 - 5.5	-0.5		0.1 V_{CC}(1)	V
V_{IH1}	Input High Voltage XTAL1 pin	V_{CC}=2.7 - 5.5 V_{CC}=4.5 - 5.5	0.7 V_{CC}(2)		V_{CC} + 0.5	V
V_{IL2}	Input Low Voltage $\overline{\text{RESET}}$ pin	V_{CC}=2.7 - 5.5	-0.5		0.2 V_{CC}	V
V_{IH2}	Input High Voltage $\overline{\text{RESET}}$ pin	V_{CC}=2.7 - 5.5	0.9 V_{CC}(2)		V_{CC} + 0.5	V
V_{OL}	Output Low Voltage(3) (Ports A,B,C,D)	I_{OL} = 20 mA, V_{CC} = 5V I_{OL} = 10 mA, V_{CC} = 3V			0.7 0.5	V V
V_{OH}	Output High Voltage(4) (Ports A,B,C,D)	I_{OH} = -20 mA, V_{CC} = 5V I_{OH} = -10 mA, V_{CC} = 3V	4.2 2.2			V V
I_{IL}	Input Leakage Current I/O Pin	V_{CC} = 5.5V, pin low (absolute value)			1	µA
I_{IH}	Input Leakage Current I/O Pin	V_{CC} = 5.5V, pin high (absolute value)			1	µA
R_{RST}	Reset Pull-up Resistor		30	60	85	kΩ
R_{pu}	I/O Pin Pull-up Resistor		20		50	kΩ

296 **ATmega32A**

8155A–AVR–06/08

27.3 Speed Grades

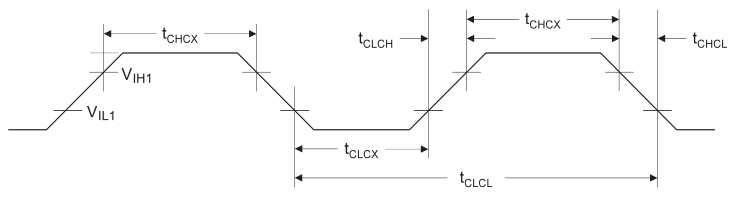
Figure 27-1. Maximum Frequency vs. V_{CC}.



27.4 Clock Characteristics

27.4.1 External Clock Drive Waveforms

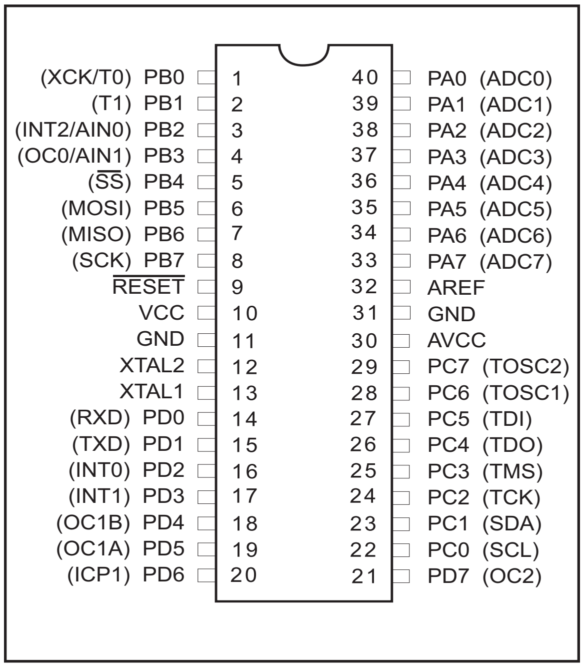
Figure 27-2. External Clock Drive Waveforms



27.4.2 External Clock Drive

Figure 27-3. External Clock Drive

Symbol	Parameter	V_{CC} = 2.7V to 5.5V		V_{CC} = 4.5V to 5.5V		Units
		Min	**Max**	**Min**	**Max**	
$1/t_{CLCL}$	Oscillator Frequency	0	8	0	16	MHz
t_{CLCL}	Clock Period	125		62.5		ns
t_{CHCX}	High Time	50		25		ns
t_{CLCX}	Low Time	50		25		ns



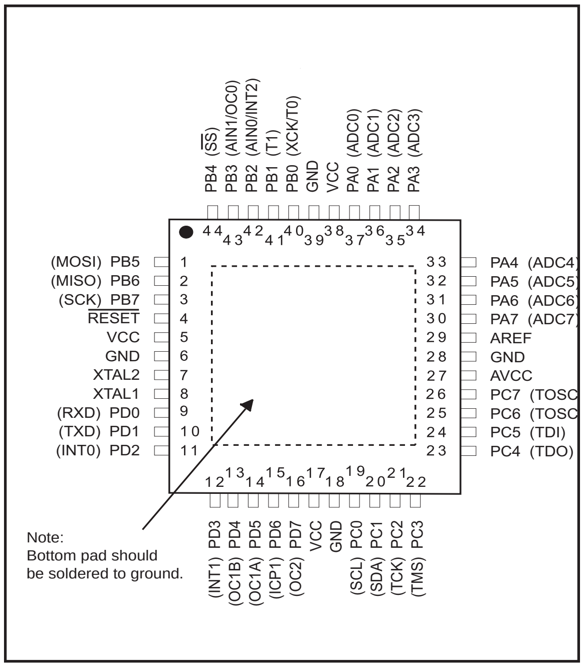


Figure H-1. ATmega16/32 DIP

Figure H-2. ATmega16/32 TQFP

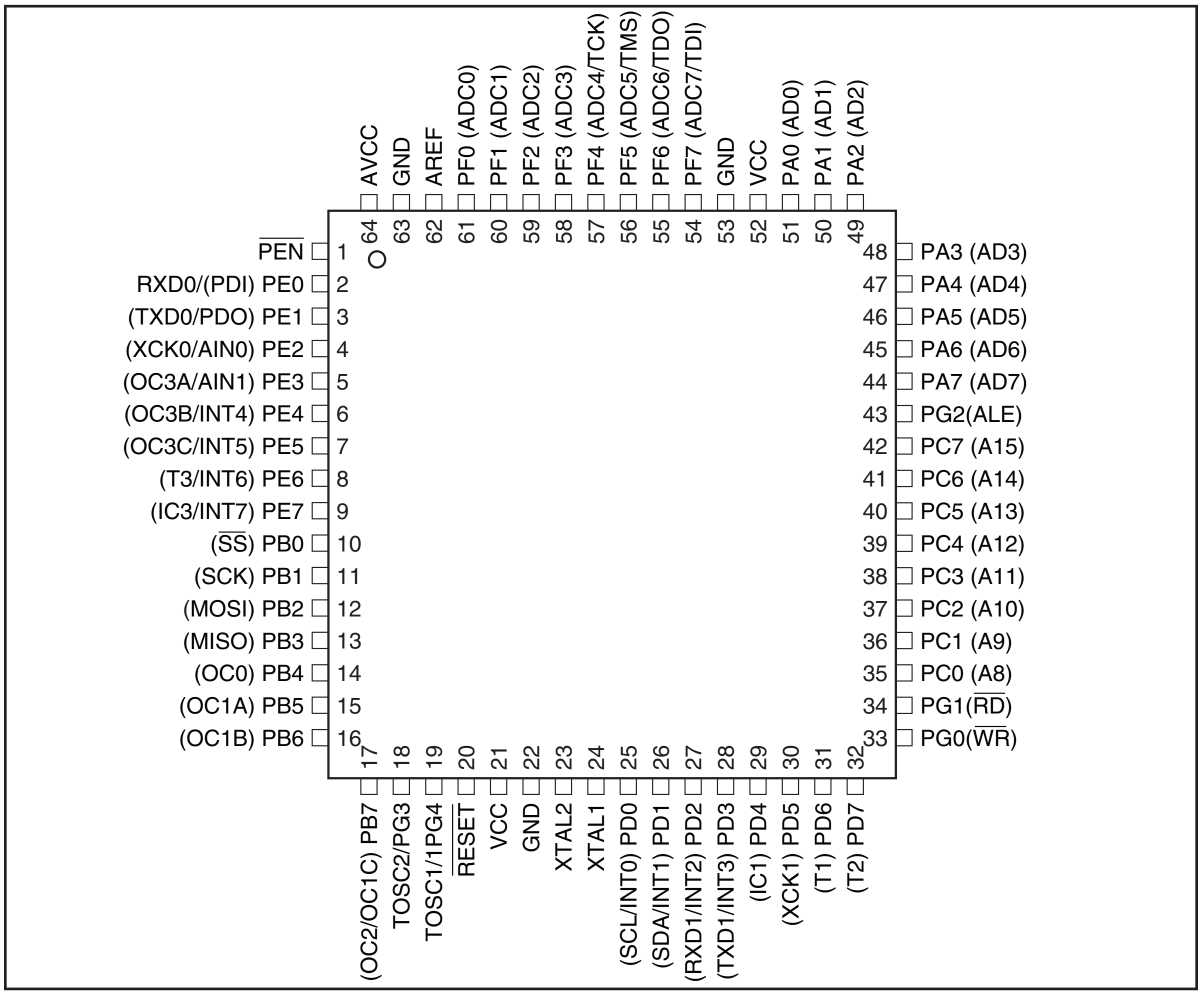


Figure H-3. ATmega 64/128 TQFP

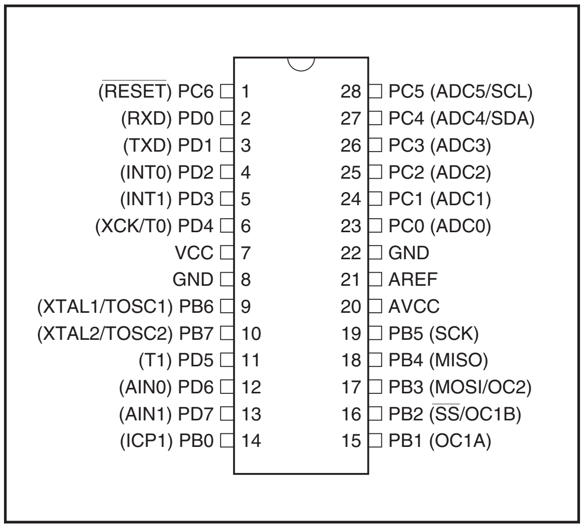


Figure H-4. ATmega8 DIP

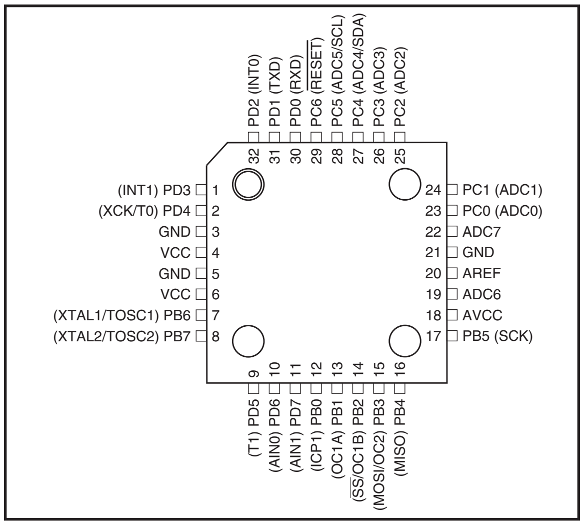


Figure H-5. ATmega8 TQFP

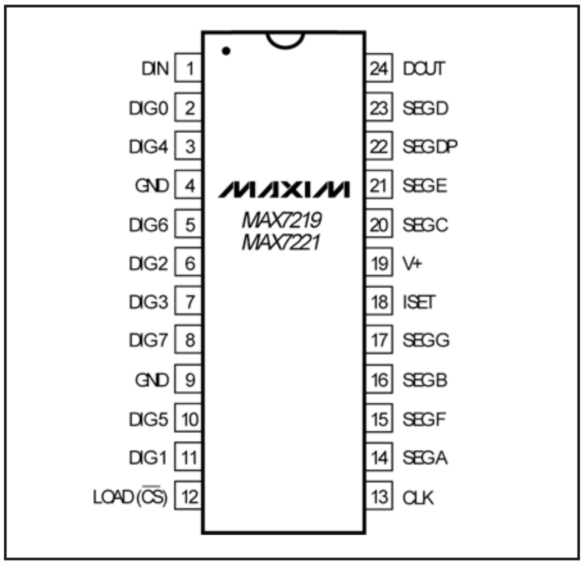


Figure H-6. MAX7221

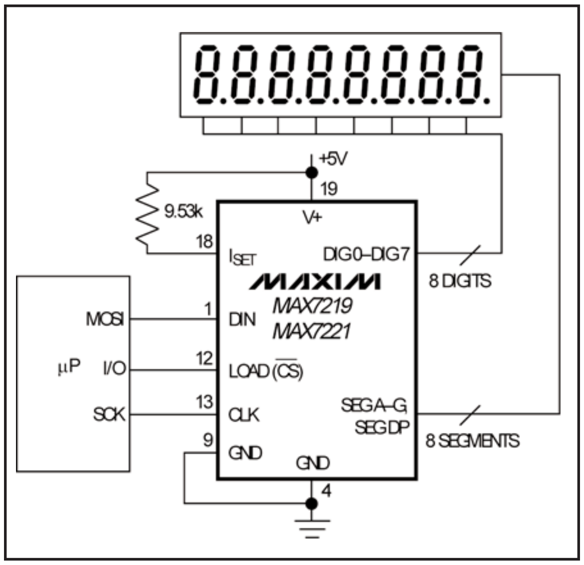


Figure H-7. MAX7221 Connections

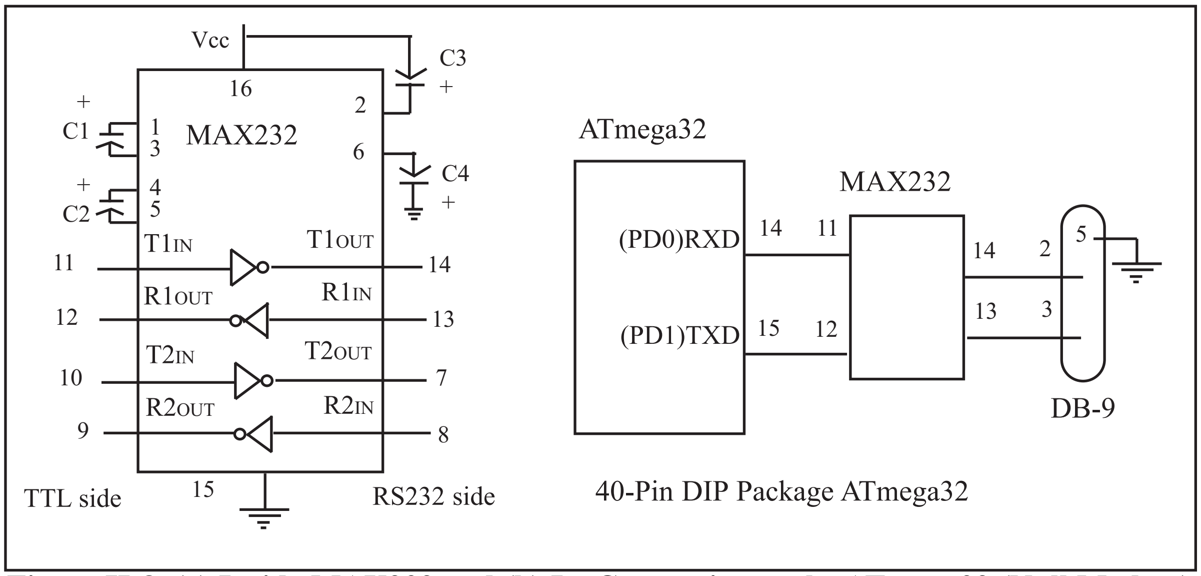


Figure H-8. (a) Inside MAX232 and (b) Its Connection to the ATmega32 (Null Modem)

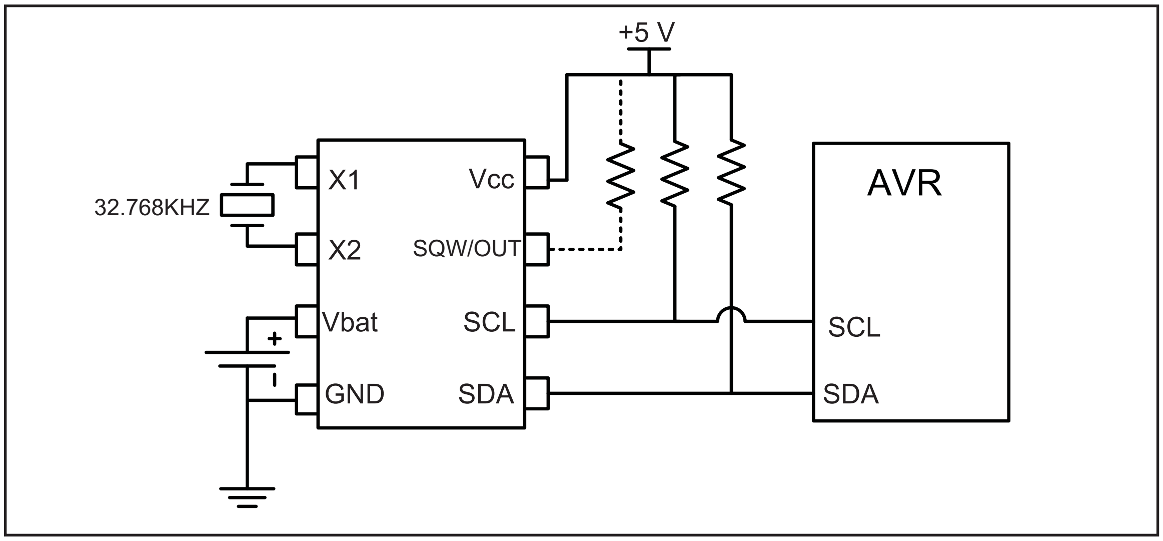


Figure H-9. DS1307 Power Connection Options (Maxim/Dallas Semiconductor)

LCD Pin	Symbol
1	Ground
2	V_{CC}
3	V_{EE}
4	RS
5	R/W
6	E
7	DB0
...	...
14	DB7

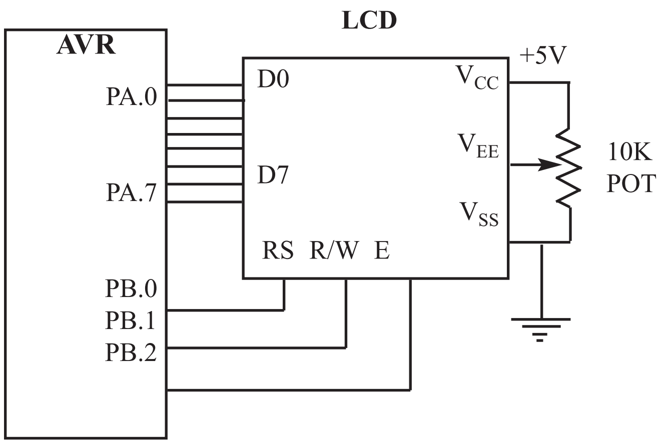


Figure H-10. LCD Connections for 8-bit Data

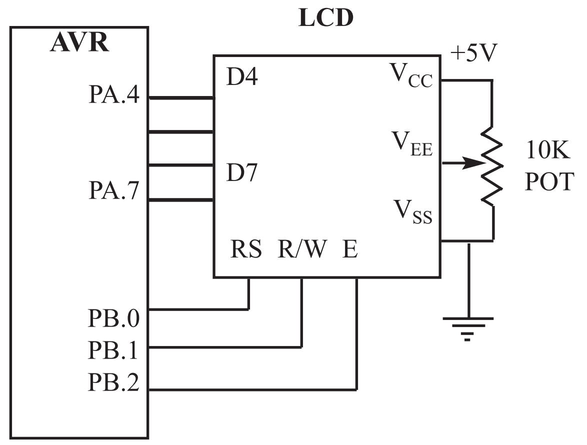


Figure H-11. LCD Connections Using 4-bit Data

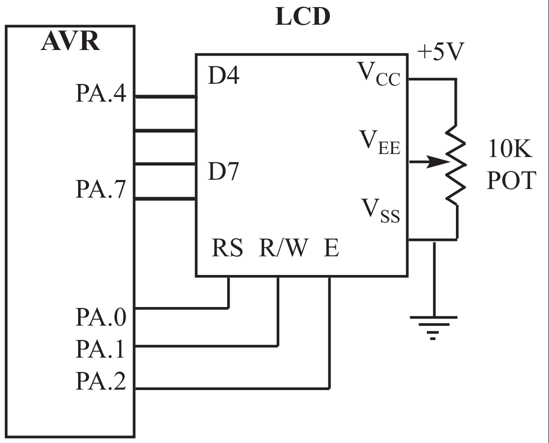


Figure H-12. LCD Connections Using a Single Port