THE AVR MICROCONTROLLER
AND EMBEDDED SYSTEMS

Using Assembly and C

Online Part

Muhammad Ali Mazidi
Sepehr Naimi
Sarmad Naimi
CONTENTS

CHAPTER 8: AVR HARDWARE CONNECTION, HEX FILE, AND FLASH LOADERS
SECTION 8.2: AVR FUSE BITS 631
SECTION 8.3: EXPLAINING THE HEX FILE FOR AVR 637
SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD 642

CHAPTER 18: I2C PROTOCOL AND DS1307 RTC INTERFACING
SECTION 18.5: TWI PROGRAMMING WITH CHECKING STATUS REGISTER 647

APPENDIX A: AVR INSTRUCTIONS EXPLAINED 671
SECTION A.1: INSTRUCTION SUMMARY 672
SECTION A.2: AVR INSTRUCTIONS FORMAT 676
SECTION A.3: AVR REGISTER SUMMARY 708

APPENDIX B: BASICS OF WIRE WRAPPING 709

APPENDIX C: IC INTERFACING AND SYSTEM DESIGN ISSUES 713
SECTION C.1: OVERVIEW OF IC TECHNOLOGY 714
SECTION C.2: AVR I/O PORT STRUCTURE AND INTERFACING 720
SECTION C.3: SYSTEM DESIGN ISSUES 726

APPENDIX D: FLOWCHARTS AND PSEUDOCODE 731

APPENDIX E: AVR PRIMER FOR 8051 PROGRAMMERS 737

APPENDIX F: ASCII CODES 738

APPENDIX G: ASSEMBLERS, DEVELOPMENT RESOURCES, AND SUPPLIERS 740

APPENDIX H: DATA SHEETS 742
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>
<thead>
<tr>
<th>Fuse High Byte</th>
<th>Bit No.</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCDEN</td>
<td>7</td>
<td>Enable OCD</td>
<td>1 (unprogrammed)</td>
</tr>
<tr>
<td>JTAGEN</td>
<td>6</td>
<td>Enable JTAG</td>
<td>0 (programmed)</td>
</tr>
<tr>
<td>SPIEN</td>
<td>5</td>
<td>Enable SPI serial program and data downloading</td>
<td>0 (programmed)</td>
</tr>
<tr>
<td>CKOPT</td>
<td>4</td>
<td>Oscillator options</td>
<td>1 (unprogrammed)</td>
</tr>
<tr>
<td>EESAVE</td>
<td>3</td>
<td>EEPROM memory is preserved through the chip erase</td>
<td>1 (unprogrammed)</td>
</tr>
<tr>
<td>BOOTSZ1</td>
<td>2</td>
<td>Select boot size</td>
<td>0 (programmed)</td>
</tr>
<tr>
<td>BOOTSZ0</td>
<td>1</td>
<td>Select boot size</td>
<td>0 (programmed)</td>
</tr>
<tr>
<td>BOOTRST</td>
<td>0</td>
<td>Select reset vector</td>
<td>1 (unprogrammed)</td>
</tr>
</tbody>
</table>

Table 8-7: Fuse Byte (Low)

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

**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.

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 \( f = \frac{1}{3RC} \). 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.

---

**Table 8-8: Internal RC Oscillator Operation Modes**

<table>
<thead>
<tr>
<th>CKSEL3...0</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1 MHz</td>
</tr>
<tr>
<td>0010</td>
<td>2 MHz</td>
</tr>
<tr>
<td>0011</td>
<td>4 MHz</td>
</tr>
<tr>
<td>0100</td>
<td>8 MHz</td>
</tr>
</tbody>
</table>

**Table 8-9: External RC Oscillator Operation Modes**

<table>
<thead>
<tr>
<th>CKSEL3...0</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>&lt;0.9</td>
</tr>
<tr>
<td>0110</td>
<td>0.9–3.0</td>
</tr>
<tr>
<td>0111</td>
<td>3.0–8.0</td>
</tr>
<tr>
<td>1000</td>
<td>8.0–12.0</td>
</tr>
</tbody>
</table>

---

**Figure 8-4. ATmega32 Clock Sources**

**Figure 8-5 External RC**
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. In Figure 8-6b you see the connection to a 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

<table>
<thead>
<tr>
<th>CKOPT</th>
<th>CKSEL3...1</th>
<th>Frequency (MHz)</th>
<th>C1 and C2 (pF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
<td>0.4–0.9</td>
<td>Not for crystals</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>0.9–3.0</td>
<td>12–22</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>3.0–8.0</td>
<td>12–22</td>
</tr>
<tr>
<td>0</td>
<td>101, 110, 111</td>
<td>More than 1.0</td>
<td>12–22</td>
</tr>
</tbody>
</table>

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

CHAPTER 8: AVR HARDWARE CONNECTION
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
...
```

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.

```c
#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 1 second
        delay_ms(1000);   //delay 1 second
        PORTB = 0xAA;   //delay 1 second
        delay_ms(1000);
    }
    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 oscillator 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

<table>
<thead>
<tr>
<th>Format Name</th>
<th>File Extension</th>
<th>Max. ROM Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended Intel Hex file</td>
<td>.hex</td>
<td>20-bit address</td>
</tr>
<tr>
<td>Motorola S-record</td>
<td>.mot</td>
<td>32-bit address</td>
</tr>
<tr>
<td>Generic</td>
<td>.gen</td>
<td>24-bit address</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Example 8-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>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?</td>
</tr>
<tr>
<td><strong>Solution:</strong></td>
</tr>
<tr>
<td>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):</td>
</tr>
<tr>
<td>0030 (2 bytes segment address) shifted 4 bits to the left --&gt; 00300</td>
</tr>
<tr>
<td>0025 (record address) + 25</td>
</tr>
<tr>
<td>=&gt; (absolute address) 00325</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 8-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the absolute address of the first byte of the second record below?</td>
</tr>
<tr>
<td>:0200000000FC</td>
</tr>
<tr>
<td>:10000000008E00EBF0FE50DBF0FEF07BB05E500553C</td>
</tr>
<tr>
<td><strong>Solution:</strong></td>
</tr>
<tr>
<td>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).</td>
</tr>
<tr>
<td>0000 (segment address) shift 4 bits to the left --&gt; 00000</td>
</tr>
<tr>
<td>+ 0000 (record address)</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>0000000 (absolute address)</td>
</tr>
</tbody>
</table>
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.

Figure 8-7. AVR Flash Memory Contents

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.

<table>
<thead>
<tr>
<th>Flash Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location #0</td>
</tr>
<tr>
<td>Location #1</td>
</tr>
<tr>
<td>Location #2</td>
</tr>
<tr>
<td>Location #3</td>
</tr>
</tbody>
</table>

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
:10000008E00EBF0FE50DBF0EF07BB05E500953C
:100010008BB0E940C00FBCF40E158EC6AEF0000E7
:10002000006A95E1F75A959F74A95B1F7089526
:00000001FF
```

Separating the fields, we get the following:

```
;BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHH CC
;02 0000 02 0000 FC
;10 0000 00 08E00EBF0FE50DBF0EF07BB05E50095 3C
;10 0010 00 08BB0E940C00FBCF40E158EC6AEF0000 E7
;10 0020 00 0006A95E1F75A959F74A95B1F70895 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?
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.
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. $1/16 \text{ MHz} = 62.5 \text{ ns}$
2. $16 \text{ bits} = 2 \text{ bytes}$
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, BODELEVEL
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

- Send START
- Is Status $8? (No)
- Is Status $18? (No)
- Is Status $28? (Yes)
- Send Data
- Want to send more data? (Yes)
- Is Status $28? (No)
- Send STOP
- Do error handling
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.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
<th>TWDR</th>
<th>TWCR</th>
<th>Next Action By TWI module</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8</td>
<td>START condition has been transmitted</td>
<td>TWDR = SLA+W</td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$18</td>
<td>SLA + W transmitted. ACK has been received</td>
<td>TWDR = DATA</td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$20</td>
<td>SLA + W transmitted. NACK has been received</td>
<td>TWCR =(1&lt;&lt;TWEN)</td>
<td>(TWINT)(TWSTO)</td>
<td></td>
</tr>
<tr>
<td>$28</td>
<td>Data byte has been transmitted. ACK has been received.</td>
<td>TWDR = DATA</td>
<td>TWCR =(1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$30</td>
<td>Data transmitted. NACK received.</td>
<td>TWCR =(1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
<td></td>
</tr>
</tbody>
</table>

**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
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.

```c
#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;
}
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-
Send START

Is Status $8? No

Send SLA+R

Is Status $40? No

Is Status $50? No

Send STOP

Want to read more data?

Want to read only one other byte?

Read Data Return ACK

Is Status $50? No

Read Data Return NACK

Is Status $58? Yes

Do error handling

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

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
<th>Your Response</th>
<th>Next Action By TWI module</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8</td>
<td>START condition has been transmitted</td>
<td>TWDR = SLA + R (1) TWCR = (1&lt;&lt;TWEN)</td>
<td>(1&lt;&lt;TWINT)(1&lt;&lt;TWSTA)</td>
</tr>
<tr>
<td>$40</td>
<td>SLA + R has been transmitted. ACK has been received</td>
<td>OR TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$48</td>
<td>SLA + R transmitted. NACK received</td>
<td>OR TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$50</td>
<td>Data byte has been received. ACK has been returned.</td>
<td>OR DATA = TWDR TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$58</td>
<td>Data byte received. NACK ACK returned.</td>
<td>OR DATA = TWDR TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
</tbody>
</table>

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.

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

```
.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
```

CHAPTER 18: I2C PROTOCOL AND DS1307 RTC INTERFACING  656
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.

```c
#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

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

```
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;
}
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 cannot 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
CHAPTER 18: I2C PROTOCOL AND DS1307 RTC INTERFACING

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

1. Wait to be addressed
2. Is TWIF one?
   - Yes
   - No
3. Is Status $A8?
   - Yes
   - No
   - Do error handling
4. Is Status $B8?
   - Yes
   - No
5. Want to send only one other byte?
   - Yes
   - No
6. Send data
5. Wait for ACK
   - Yes
   - No
   - Want to send more data?
   - Yes
   - No
   - Send Data
   - Wait for NACK
   - Yes
   - No
   - Is Status $C8 or $C0?
   - Yes
   - No
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 ($0C8), 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 $0C8, 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-

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
<th>Your Response</th>
<th>Next Action By TWI module</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A8</td>
<td>Own SLA+R received ACK returned</td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(1&lt;&lt;TWIF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWDR = DATA</td>
<td>DATA byte will be transmitted Wait for ACK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$B8</td>
<td>Data has been transmitted ACK received</td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(1&lt;&lt;TWIF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWDR = DATA</td>
<td>DATA byte will be transmitted Wait for ACK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$C0</td>
<td>Data has been transmitted NACK received</td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(1&lt;&lt;TWIF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
<tr>
<td>$C8</td>
<td>Data transmitted ACK received but you wanted NACK (TWEA was 0 in last command)</td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(1&lt;&lt;TWIF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWCR = (1&lt;&lt;TWEN)</td>
<td>(TWINT)</td>
</tr>
</tbody>
</table>

**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.

```assembly
.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 transmitter?
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:
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 instruction if TWINT is 1
RJMP W1; jump to W1 if TWINT is 0
RET

HERE:
RJMP HERE; wait here forever

ERROR:
LDI R21, 0xFF
OUT DDRA, R21; Port A is output
OUT PORTA, R26
RJMP HERE
```

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

```c
#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 18-18: 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.
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.

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 output

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

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

```c
#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)|TWEN); //wait to complete
        return (TWDR);
else //if want to read only one byte
    TWCR = (1<< TWINT)|(1<<TWEN);
while ((TWCR & (1 <<TWINT))==0); //wait to complete
}
//*********************************************************
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
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 a START 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?

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

```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;
}
```
OVERVIEW

In this appendix, we describe each instruction 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

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

<table>
<thead>
<tr>
<th>Mnem.</th>
<th>Oper.</th>
<th>Description</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>RJMP</td>
<td>k</td>
<td>Relative Jump</td>
<td>PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>IJMP</td>
<td></td>
<td>Indirect Jump to (Z)</td>
<td>PC ← Z</td>
<td>None</td>
</tr>
<tr>
<td>JMP</td>
<td>k</td>
<td>Direct Jump</td>
<td>PC ← k</td>
<td>None</td>
</tr>
<tr>
<td>RCALL</td>
<td>k</td>
<td>Relative Subroutine Call</td>
<td>PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>ICALL</td>
<td></td>
<td>Indirect Call to (Z)</td>
<td>PC ← Z</td>
<td>None</td>
</tr>
<tr>
<td>CALL</td>
<td>k</td>
<td>Direct Subroutine Call</td>
<td>PC ← k</td>
<td>None</td>
</tr>
<tr>
<td>RET</td>
<td>k</td>
<td>Subroutine Return</td>
<td>PC ← Stack</td>
<td>None</td>
</tr>
<tr>
<td>RETI</td>
<td></td>
<td>Interrupt Return</td>
<td>PC ← Stack</td>
<td>I</td>
</tr>
<tr>
<td>CPSE</td>
<td>Rd,Rr</td>
<td>Compare, Skip if Equal</td>
<td>if (Rd = Rr) PC ← PC + 2 or 3</td>
<td>None</td>
</tr>
<tr>
<td>CP</td>
<td>Rd,Rr</td>
<td>Compare</td>
<td>Rd − Rr</td>
<td>Z,N,V,C,H</td>
</tr>
<tr>
<td>CPC</td>
<td>Rd,Rr</td>
<td>Compare with Carry</td>
<td>Rd − Rr − C</td>
<td>Z,N,V,C,H</td>
</tr>
<tr>
<td>CPI</td>
<td>Rd,K</td>
<td>Compare Register with Immediate</td>
<td>Rd − K</td>
<td>Z,N,V,C,H</td>
</tr>
<tr>
<td>SBRC</td>
<td>Rr,b</td>
<td>Skip if Bit in Register Cleared</td>
<td>if (Rr(b)=0) PC ← PC + 2 or 3</td>
<td>None</td>
</tr>
<tr>
<td>SBRS</td>
<td>Rr,b</td>
<td>Skip if Bit in Register is Set</td>
<td>if (Rr(b)=1) PC ← PC + 2 or 3</td>
<td>None</td>
</tr>
<tr>
<td>SBIC</td>
<td>P,b</td>
<td>Skip if Bit in I/O Register Cleared</td>
<td>if (P(b)=0) PC ← PC + 2 or 3</td>
<td>None</td>
</tr>
<tr>
<td>SBIS</td>
<td>P,b</td>
<td>Skip if Bit in I/O Register is Set</td>
<td>if (P(b)=1) PC ← PC + 2 or 3</td>
<td>None</td>
</tr>
<tr>
<td>BRBS</td>
<td>s,k</td>
<td>Branch if Status Flag Set</td>
<td>if (SREG(s)=1) then PC←PC+k+1</td>
<td>None</td>
</tr>
<tr>
<td>BRBC</td>
<td>s,k</td>
<td>Branch if Status Flag Cleared</td>
<td>if (SREG(s)=0) then PC←PC+k+1</td>
<td>None</td>
</tr>
<tr>
<td>BREQ</td>
<td>k</td>
<td>Branch if Equal</td>
<td>if (Z = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRNE</td>
<td>k</td>
<td>Branch if Not Equal</td>
<td>if (Z = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRCS</td>
<td>k</td>
<td>Branch if Carry Set</td>
<td>if (C = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRCC</td>
<td>k</td>
<td>Branch if Carry Cleared</td>
<td>if (C = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRSH</td>
<td>k</td>
<td>Branch if Same or Higher</td>
<td>if (C = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRLT</td>
<td>k</td>
<td>Branch if Lower</td>
<td>if (C = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRMI</td>
<td>k</td>
<td>Branch if Minus</td>
<td>if (N = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRPL</td>
<td>k</td>
<td>Branch if Plus</td>
<td>if (N = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRGE</td>
<td>k</td>
<td>Branch if Greater or Equal, Signed</td>
<td>if (N and V= 0) then PC←PC+PC+k+1</td>
<td>None</td>
</tr>
<tr>
<td>BRLT</td>
<td>k</td>
<td>Branch if Less Than Zero, Signed</td>
<td>if (N and V= 1) then PC←PC+PC+k+1</td>
<td>None</td>
</tr>
<tr>
<td>BRHS</td>
<td>k</td>
<td>Branch if Half Carry Flag Set</td>
<td>if (H = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRHC</td>
<td>k</td>
<td>Branch if Half Carry Flag Cleared</td>
<td>if (H = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRTS</td>
<td>k</td>
<td>Branch if T Flag Set</td>
<td>if (T = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRTC</td>
<td>k</td>
<td>Branch if T Flag Cleared</td>
<td>if (T = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRVS</td>
<td>k</td>
<td>Branch if Overflow Flag is Set</td>
<td>if (V = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRVC</td>
<td>k</td>
<td>Branch if Overflow Flag is Cleared</td>
<td>if (V = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRIE</td>
<td>k</td>
<td>Branch if Interrupt Enabled</td>
<td>if (I = 1) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>BRID</td>
<td>k</td>
<td>Branch if Interrupt Disabled</td>
<td>if (I = 0) then PC ← PC + k + 1</td>
<td>None</td>
</tr>
<tr>
<td>Mnem.</td>
<td>Operan.</td>
<td>Description</td>
<td>Operation</td>
<td>Flags</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-------------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>SBI</td>
<td>P, b</td>
<td>Set Bit in I/O Register</td>
<td>I/O(P, b) ← 1</td>
<td>None</td>
</tr>
<tr>
<td>CBI</td>
<td>P, b</td>
<td>Clear Bit in I/O Register</td>
<td>I/O(P, b) ← 0</td>
<td>None</td>
</tr>
<tr>
<td>LSL</td>
<td>Rd</td>
<td>Logical Shift Left</td>
<td>RD(n+1) ← RD(n), RD(0) ← 0</td>
<td>Z,C,N,V</td>
</tr>
<tr>
<td>LSR</td>
<td>Rd</td>
<td>Logical Shift Right</td>
<td>RD(n) ← RD(n+1), RD(7) ← 0</td>
<td>Z,C,N,V</td>
</tr>
<tr>
<td>ROL</td>
<td>Rd</td>
<td>Rotate Left Through Carry</td>
<td>RD(7) ← C, RD(n+1) ← RD(n), C ← RD(7)</td>
<td>Z,C,N,V</td>
</tr>
<tr>
<td>ROR</td>
<td>Rd</td>
<td>Rotate Right Through Carry</td>
<td>RD(7) ← C, RD(n) ← RD(n+1), C ← RD(0)</td>
<td>Z,C,N,V</td>
</tr>
<tr>
<td>ASR</td>
<td>Rd</td>
<td>Arithmetic Shift Right</td>
<td>RD(n) ← RD(n+1), n = 0..6</td>
<td>Z,C,N,V</td>
</tr>
<tr>
<td>SWAP</td>
<td>Rd</td>
<td>Swap Nibbles</td>
<td>RD(3..0) ← RD(7..4), RD(7..4) ← RD(3..0)</td>
<td>None</td>
</tr>
<tr>
<td>BSET</td>
<td>s</td>
<td>Flag Set</td>
<td>SREG(s) ← 1</td>
<td>SREG(s)</td>
</tr>
<tr>
<td>BCLR</td>
<td>s</td>
<td>Flag Clear</td>
<td>SREG(s) ← 0</td>
<td>SREG(s)</td>
</tr>
<tr>
<td>BST</td>
<td>Rr, b</td>
<td>Bit Store from Register to T</td>
<td>T ← Rr(b)</td>
<td>T</td>
</tr>
<tr>
<td>BLD</td>
<td>Rd, b</td>
<td>Bit Load from T to Register</td>
<td>Rd(b) ← T</td>
<td>None</td>
</tr>
<tr>
<td>SEC</td>
<td></td>
<td>Set Carry</td>
<td>C ← 1</td>
<td>C</td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td>Clear Carry</td>
<td>C ← 0</td>
<td>C</td>
</tr>
<tr>
<td>SEN</td>
<td></td>
<td>Set Negative Flag</td>
<td>N ← 1</td>
<td>N</td>
</tr>
<tr>
<td>CLN</td>
<td></td>
<td>Clear Negative Flag</td>
<td>N ← 0</td>
<td>N</td>
</tr>
<tr>
<td>SEZ</td>
<td></td>
<td>Set Zero Flag</td>
<td>Z ← 1</td>
<td>Z</td>
</tr>
<tr>
<td>CLZ</td>
<td></td>
<td>Clear Zero Flag</td>
<td>Z ← 0</td>
<td>Z</td>
</tr>
<tr>
<td>SEI</td>
<td></td>
<td>Global Interrupt Enable</td>
<td>I ← 1</td>
<td>I</td>
</tr>
<tr>
<td>CLI</td>
<td></td>
<td>Global Interrupt Disable</td>
<td>I ← 0</td>
<td>I</td>
</tr>
<tr>
<td>SES</td>
<td></td>
<td>Set Signed Test Flag</td>
<td>S ← 1</td>
<td>S</td>
</tr>
<tr>
<td>CLS</td>
<td></td>
<td>Clear Signed Test Flag</td>
<td>S ← 0</td>
<td>S</td>
</tr>
<tr>
<td>SEV</td>
<td></td>
<td>Set Two’s Complement Overflow</td>
<td>V ← 1</td>
<td>V</td>
</tr>
<tr>
<td>CLV</td>
<td></td>
<td>Clear Two’s Complement Overflow</td>
<td>V ← 0</td>
<td>V</td>
</tr>
<tr>
<td>SET</td>
<td></td>
<td>Set T in SREG</td>
<td>T ← 1</td>
<td>T</td>
</tr>
<tr>
<td>CLT</td>
<td></td>
<td>Clear T in SREG</td>
<td>T ← 0</td>
<td>T</td>
</tr>
<tr>
<td>SEH</td>
<td></td>
<td>Set Half Carry Flag in SREG</td>
<td>H ← 1</td>
<td>H</td>
</tr>
<tr>
<td>CLH</td>
<td></td>
<td>Clear Half Carry Flag in SREG</td>
<td>H ← 0</td>
<td>H</td>
</tr>
</tbody>
</table>
### ARITHMETIC AND LOGIC INSTRUCTIONS

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

### MCU CONTROL INSTRUCTIONS

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Operands</th>
<th>Description</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td></td>
<td>No Operation</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>SLEEP</td>
<td></td>
<td>Sleep</td>
<td>(see specific descr. for Sleep function)</td>
<td>None</td>
</tr>
<tr>
<td>WDR</td>
<td></td>
<td>Watchdog Reset</td>
<td>(see specific descr. for WDR/timer)</td>
<td>None</td>
</tr>
<tr>
<td>BREAK</td>
<td></td>
<td>Break</td>
<td>For On-Chip Debug Only</td>
<td>None</td>
</tr>
</tbody>
</table>

APPENDIX A: AVR INSTRUCTIONS EXPLAINED 675
### SECTION A.2: AVR INSTRUCTIONS FORMAT

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Format</th>
<th>Flags</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADC</strong> Rd, Rr</td>
<td>Adds two registers and the contents of the C flag and places the result in the destination register Rd.</td>
<td>$0 \leq d \leq 31, 0 \leq r \leq 31$</td>
<td>H, S, V, N, Z, C</td>
<td>1</td>
</tr>
<tr>
<td><strong>ADD</strong> Rd, Rr</td>
<td>Adds two registers without the C flag and places the result in the destination register Rd.</td>
<td>$0 \leq d \leq 31, 0 \leq r \leq 31$</td>
<td>H, S, V, N, Z, C</td>
<td>1</td>
</tr>
<tr>
<td><strong>ADIW</strong> Rd+1:Rd, K</td>
<td>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.</td>
<td>$d \in {24,26,28,30}, 0 \leq K \leq 63$</td>
<td>S, V, N, Z, C</td>
<td>2</td>
</tr>
<tr>
<td><strong>AND</strong> Rd, Rr</td>
<td>Performs the logical AND between the contents of register Rd and register Rr and places the result in the destination register Rd.</td>
<td>$0 \leq d \leq 31, 0 \leq r \leq 31$</td>
<td>S, V, N, Z</td>
<td>1</td>
</tr>
<tr>
<td><strong>ANDI</strong> Rd, K</td>
<td>Performs the logical AND between the contents of register Rd and a constant and places the result in the destination register Rd.</td>
<td>$16 \leq d \leq 31, 0 \leq K \leq 255$</td>
<td>S, V, N, Z</td>
<td>1</td>
</tr>
</tbody>
</table>

---

Example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add r2,r0</code></td>
<td>Add low byte</td>
</tr>
<tr>
<td><code>adc r3,r1</code></td>
<td>Add with carry high byte</td>
</tr>
<tr>
<td><code>add r1,r2</code></td>
<td>Add r2 to r1 (r1=r1+r2)</td>
</tr>
<tr>
<td><code>add r28,r28</code></td>
<td>Add r28 to itself (r28=r28+r28)</td>
</tr>
<tr>
<td><code>adiw r25:24,1</code></td>
<td>Add 1 to r25:r24</td>
</tr>
<tr>
<td><code>adiw ZH:ZL,63</code></td>
<td>Add 63 to the Z-pointer (r31:r30)</td>
</tr>
<tr>
<td><code>and r2,r3</code></td>
<td>Bitwise and r2 and r3, result in r2</td>
</tr>
<tr>
<td><code>ldi r16,1</code></td>
<td>Set bitmask 0000 0001 in r16</td>
</tr>
<tr>
<td><code>and r2,r16</code></td>
<td>Isolate bit 0 in r2</td>
</tr>
</tbody>
</table>
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: 1 or 2

Example:

```
cpi r20,5  ; Compare r20 to the value 5
brb c 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.
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)
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRCC k</td>
<td>Branch if Carry Cleared</td>
<td>$-64 \leq k \leq +63$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRCS k</td>
<td>Branch if Carry Set</td>
<td>$-64 \leq k \leq +63$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BREAK</td>
<td>Break</td>
<td></td>
</tr>
</tbody>
</table>
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)

\(-64 \leq k \leq +63\)  ; If Rd < Rr \((C = 1)\) then \(PC \rightarrow PC + k + 1\), else \(PC \rightarrow PC + 1\)

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)

\(-64 \leq k \leq +63\)  ; If Rd < Rr \((N \oplus V = 1)\) then \(PC \rightarrow PC + k + 1\), else \(PC \rightarrow PC + 1\)

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

\(-64 \leq k \leq +63\)  ; If N=1 then \(PC \rightarrow PC + k + 1\), else \(PC \rightarrow PC + 1\)

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

\(-64 \leq k \leq +63\)  ; If Rd \(\neq\) Rr \((Z = 0)\) then \(PC \rightarrow PC + k + 1\), else \(PC \rightarrow PC + 1\)

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)

\[\text{BRPL } k\] ; Branch if Plus
\[\text{–64} \leq k \leq +63\] ; If \(N = 0\) then \(\text{PC} \leftarrow \text{PC} + k + 1\), else \(\text{PC} \leftarrow \text{PC} + 1\)

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

Flags: --- Cycles: 1 or 2

Example:

\[\text{subi } r26,$50\] ; Subtract $50 from r26
\[\text{brpl positive}\] ; Branch if r26 positive
... positive: nop ; Branch destination (do nothing)

\[\text{BRSH } k\] ; Branch if Same or Higher (Unsigned)
\[\text{–64} \leq k \leq +63\] ; If \(R_d \geq R_r (C = 0)\) then \(\text{PC} \leftarrow \text{PC} + k + 1\), else \(\text{PC} \leftarrow \text{PC} + 1\)

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:

\[\text{subi } r19,4\] ; Subtract 4 from r19
\[\text{brsh highsm}\] ; Branch if r19 \(\geq 4\) (unsigned)
... highsm: nop ; Branch destination (do nothing)

\[\text{BRTC } k\] ; Branch if the T Flag is Cleared
\[\text{–64} \leq k \leq +63\] ; If \(T = 0\) then \(\text{PC} \leftarrow \text{PC} + k + 1\), else \(\text{PC} \leftarrow \text{PC} + 1\)

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

Flags: --- Cycles: 1 or 2

Example:

\[\text{bst r3,5}\] ; Store bit 5 of r3 in T flag
\[\text{brtc tclear}\] ; Branch if this bit was cleared
... tclear: nop ; Branch destination (do nothing)

APPENDIX A: AVR INSTRUCTIONS EXPLAINED
### BRTS k

| 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

| 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
...
nover: nop ; Branch destination (do nothing)
```

### BRVS k

| 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

| 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

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

---

682  The AVR Microcontroller & Embedded Systems (Mazidi & Naimi)
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:

```assembly
mov r16,r0 ;Copy r0 to r16
call check ;Call subroutine
nop ;Continue (do nothing)
... 
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:

```assembly
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:

```assembly
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:

```assembly
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:

```assembly
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:

```assembly
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:

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

CLR Rd ; Clear Register
0 ≤ d ≤ 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:

```assembly
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:

```assembly
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:

```assembly
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 ≤ d ≤ 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 ≤ d ≤ 31, 0 ≤ r ≤ 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 ≤ d ≤ 31, 0 ≤ r ≤ 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:
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 logicalExclusive 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
This instruction performs 8-bit × 8-bit → 16-bit unsigned multiplication and shifts the result one bit left.

<table>
<thead>
<tr>
<th>Rd</th>
<th>Multiplier</th>
<th>×</th>
<th>R1</th>
<th>Product High</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

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:

```
fms 16x16_32:
clr r2
fmuls r23, r21 ;((signed)ah *(signed)bh) << 1
movw r19:r18, r1:r0
fml r22, r20 ;(al * bl) << 1
adc r18, r2
movwr 17:r16, r1:r0
fmuls r23, r20 ;((signed)ah * bl) << 1
sbc r19, r2
add r17, r0
adc r18, r1
adc r19, r2
fmuls 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 ≤ d ≤ 23, 16 ≤ r ≤ 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.

<table>
<thead>
<tr>
<th>Rd</th>
<th>×</th>
<th>Rr</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

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:

```assembly
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 ≤ d ≤ 23, 16 ≤ r ≤ 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:

```
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 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
```

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:
```assembly
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 ≤ d ≤ 31, 0 ≤ A ≤ 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:
```assembly
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 ≤ d ≤ 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:
```assembly
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 ≤ 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
(iv) 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.
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.

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>Operation:</th>
<th>Comment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) LD Rd, Z</td>
<td>Rd ← (Z)</td>
<td>Z: Unchanged</td>
</tr>
<tr>
<td>(ii) LD Rd, Z+</td>
<td>Rd ← (Z) Z ← Z + 1</td>
<td>Z: Postincrement</td>
</tr>
<tr>
<td>(iii) LD Rd, –Z</td>
<td>Z ← Z – 1 Rd ← (Z)</td>
<td>Z: Predecrement</td>
</tr>
<tr>
<td>(iiii) LDD Rd, Z + q</td>
<td>Rd ← (Z + q)</td>
<td>Z: Unchanged, q: Displacement</td>
</tr>
</tbody>
</table>

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: 

(i) LPM
   \( R0 \leftarrow (Z) \)  
   Z: Unchanged, R0 implied Rd

(ii) LPM Rd, Z
    \( Rd \leftarrow (Z) \)  
    Z: Unchanged

(iii) LPM Rd, Z+
     \( Rd \leftarrow (Z), Z \leftarrow Z + 1 \)  
     Z: Postincremented

Example:

```assembly
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:

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

---

**LSR Rd**  
*Logical Shift Right*

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:

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

---

**MOV Rd,Rr**  
*Copy Register*

0 ≤ d ≤ 31, 0 ≤ r ≤ 31

\( Rd \leftarrow Rr \)

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: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**

This instruction performs 8-bit × 8-bit → 16-bit unsigned 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**

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

**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**  
\(16 \leq d \leq 31, 0 \leq K \leq 255\)  
*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:
```assembly
ori r16,$F0 ; Set high nibble of r16
ori r17,1 ; Set bit 0 of r17
```

**OUT A,Rr**  
\(0 \leq r \leq 31, 0 \leq A \leq 63\)  
*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:
```assembly
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**  
\(0 \leq d \leq 31\)  
*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:
```assembly
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**  
\(0 \leq d \leq 31\)  
*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:
```assembly
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

\(-2K \leq k < 2K\); 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
```

698 The AVR Microcontroller & Embedded Systems (Mazidi & Naimi)
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:
ls1 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
brcs zeroenc1 ;Branch if carry cleared
asr r17 ;Divide r17:r16 by two
ror r16 ;r17:r16 is a signed two-byte integer
brcs zeroenc2 ;Branch if carry cleared
...
zeroenc1: nop ;Branch destination (do nothing)
...
zeroenc2: nop ;Branch destination (do nothing)
SBC Rd,Rr ; Subtract with Carry
\[ 0 \leq d \leq 31, \ 0 \leq r \leq 31 \]
\[ ; \ Rd \leftarrow Rd - Rr - C \]

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:
\[
\text{sub } r2, r0 ; \text{Subtract low byte} \\
\text{sbc } r3, r1 ; \text{Subtract with carry high byte}
\]

SBCI Rd,K ; Subtract Immediate with Carry
\[ 0 \leq d \leq 31, \ 0 \leq r \leq 31 \]
\[ ; \ Rd \leftarrow Rd - K - C \]

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:
\[
\text{subi } r16, \$23 ; \text{Subtract low byte} \\
\text{sbci } r17, \$4F ; \text{Subtract with carry high byte}
\]

SBI A,b ; Set Bit in I/O Register
\[ 0 \leq A \leq 31, \ 0 \leq b \leq 7 \]
\[ ; \ I/O(A,b) \leftarrow 1 \]

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

Flags: ---.  Cycles: 2

Example:
\[
\text{out } \$1E, r0 ; \text{Write EEPROM address} \\
\text{sbi } \$1C, 0 ; \text{Set read bit in EECR} \\
\text{in } r1, \$1D ; \text{Read EEPROM data}
\]

SBIC A,b ; Skip if Bit in I/O Register is Cleared
\[ 0 \leq d \leq 31, \ 0 \leq r \leq 31 \]
\[ ; \text{If } I/O(A,b) = 0 \text{ then } PC \leftarrow PC + 2 \text{ (or 3) else } PC \leftarrow PC + 1 \]

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:
\[
\text{e2wait: sbic } \$1C, 1 ; \text{Skip next inst. if EEWE cleared} \\
\text{rjmp e2wait ; EEPROM write not finished} \\
\text{nop ; Continue (do nothing)}
\]

SBIS A,b ; Skip if Bit in I/O Register is Set
\[ 0 \leq d \leq 31, \ 0 \leq r \leq 31 \]
\[ ; \text{If } I/O(A,b) = 1 \text{ then } PC \leftarrow PC + 2 \text{ (or 3) else } PC \leftarrow PC + 1 \]

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:
\[
\text{waitset: sbis } \$10, 0 ; \text{Skip next inst. if bit 0 in Port D set} \\
\text{rjmp waitset ; Bit not set} \\
\text{nop ; Continue (do nothing)}
\]
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.
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
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEH</td>
<td>; Set Half Carry Flag</td>
<td>H ← 1</td>
</tr>
<tr>
<td>SEI</td>
<td>; Set Global Interrupt Flag</td>
<td>I ← 1</td>
</tr>
<tr>
<td>SEN</td>
<td>; Set Negative Flag</td>
<td>N ← 1</td>
</tr>
<tr>
<td>SER Rd</td>
<td>; Set all Bits in Register</td>
<td>16 ≤ d ≤ 31; Rd ← $FF</td>
</tr>
<tr>
<td>SES</td>
<td>; Set Signed Flag</td>
<td>S ← 1</td>
</tr>
<tr>
<td>SET</td>
<td>; Set T Flag</td>
<td>T ← 1</td>
</tr>
</tbody>
</table>

Sets the Half Carry (H) in SREG (Status Register).
Example: seh ; Set Half Carry flag

Sets the Global Interrupt flag (I) in SREG (Status Register). The instruction following SEI will be executed before any pending interrupts.
Example: sei ; Set global interrupt enable

Sets the Negative flag (N) in SREG (Status Register).
Example: add r2,r19 ; Add r19 to r2

Loads $FF directly to register Rd.
Example: ser r17 ; Set r17

Sets the Signed flag (S) in SREG (Status Register).
Example: add r2,r19 ; Add r19 to r2

Sets the T flag in SREG (Status Register).
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. 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) SPM</td>
<td>(RAMPZ:Z) ← $ffff</td>
<td>Erase program memory page</td>
</tr>
<tr>
<td>(ii) SPM</td>
<td>(RAMPZ:Z) ← R1:R0</td>
<td>Write program memory word</td>
</tr>
<tr>
<td>(iii) SPM</td>
<td>(RAMPZ:Z) ← R1:R0</td>
<td>Write temporary page buffer</td>
</tr>
<tr>
<td>(iv) SPM</td>
<td>(RAMPZ:Z) ← TEMP</td>
<td>Write temporary page buffer to program memory</td>
</tr>
<tr>
<td>(v) SPM</td>
<td>BLBITS ← R1:R0</td>
<td>Set Boot Loader Lock bits</td>
</tr>
</tbody>
</table>
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)

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 is added to the entire 24-bit address on such devices.
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
(iii) 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
(iii) STD Z + q, Rr  (Z + q) ← Rr  Z: Unchanged,

Example:
```assembly
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:
```assembly
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:
```assembly
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:            ; Branch destination (do nothing)
```

---

**SWAP Rd**

; Swap Nibbles

\[0 \leq d \leq 31\]

\[R(7:4) \leftarrow Rd(3:0), R(3:0) \leftarrow Rd(7:4)\]

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

\[0 \leq d \leq 31\]

\[Rd \leftarrow Rd \land Rd\]

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 \leftarrow 1, N, Z

Cycles: 1

Example:

```
tst r0          ; Test r0
breq zero       ; Branch if r0=0
...              
zero:            ; 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3F$</td>
<td>SREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3E$</td>
<td>SPH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3D$</td>
<td>SPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3C$</td>
<td>OCR0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3B$</td>
<td>GICR</td>
<td>INT1</td>
<td>INT0</td>
<td>INT2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3A$</td>
<td>GIFR</td>
<td>INT1</td>
<td>INT0</td>
<td>INT2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$39$</td>
<td>TIMSK</td>
<td>OCIE2</td>
<td>TOIE2</td>
<td>TCIE1</td>
<td>OCIE1A</td>
<td>OCIE1B</td>
<td>TOIE1</td>
<td>OCIE0</td>
<td>TOIE0</td>
</tr>
<tr>
<td>$38$</td>
<td>TIFR</td>
<td>OCIF2</td>
<td>TOV2</td>
<td>ICF1</td>
<td>OCIF1A</td>
<td>OCIF1B</td>
<td>TOV1</td>
<td>OCFO</td>
<td>TOV0</td>
</tr>
<tr>
<td>$37$</td>
<td>SPIMCR</td>
<td>SPMIE</td>
<td>RWWSRE</td>
<td>BLBSET</td>
<td>PGWRT</td>
<td>PGR9</td>
<td>SPME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$36$</td>
<td>TWCR</td>
<td>TWINT</td>
<td>TWEA</td>
<td>TWSTA</td>
<td>TWSTO</td>
<td>TWCC</td>
<td>TWEN</td>
<td></td>
<td>TWIE</td>
</tr>
<tr>
<td>$35$</td>
<td>MCUCR</td>
<td>SE</td>
<td>SM2</td>
<td>SM1</td>
<td>SM0</td>
<td>ISCl1</td>
<td>ISCl0</td>
<td>ISCl0</td>
<td>ISCl0</td>
</tr>
<tr>
<td>$34$</td>
<td>MCUCSR</td>
<td>JTD</td>
<td>ISC2</td>
<td></td>
<td>JTRF</td>
<td>WDRF</td>
<td>BORF</td>
<td>EXTTR</td>
<td>PORF</td>
</tr>
<tr>
<td>$33$</td>
<td>TCCR0</td>
<td>FOC0</td>
<td>WGM00</td>
<td>COM01</td>
<td>COM00</td>
<td>WGM01</td>
<td>CS02</td>
<td>CS01</td>
<td>CS00</td>
</tr>
<tr>
<td>$32$</td>
<td>TCNT0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$31$</td>
<td>OSCCAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$30$</td>
<td>SFIOR</td>
<td>ADTS2</td>
<td>ADTS1</td>
<td>ADTS0</td>
<td></td>
<td></td>
<td>ACME</td>
<td>PUD</td>
<td>PSR2</td>
</tr>
<tr>
<td>$2F$</td>
<td>TCCR1A</td>
<td>COMA1A</td>
<td>COMA0</td>
<td>COMA1B</td>
<td>COMA0B</td>
<td>FOC1A</td>
<td>FOC1B</td>
<td>WGM11</td>
<td>WGM10</td>
</tr>
<tr>
<td>$2E$</td>
<td>TCCR1B</td>
<td>ICIC1</td>
<td>ICES1</td>
<td></td>
<td>WGM13</td>
<td>WGM12</td>
<td>CS12</td>
<td>CS11</td>
<td>CS10</td>
</tr>
<tr>
<td>$2D$</td>
<td>TCTN1H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2C$</td>
<td>TCTN1L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2B$</td>
<td>OCR1AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2A$</td>
<td>OCR1AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$29$</td>
<td>OCR1BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$28$</td>
<td>OCR1BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$27$</td>
<td>ICR1H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$26$</td>
<td>ICR1L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$25$</td>
<td>TCCR2</td>
<td>FOC2</td>
<td>WGM20</td>
<td>COM21</td>
<td>COM20</td>
<td>WGM21</td>
<td>CS22</td>
<td>CS21</td>
<td>CS20</td>
</tr>
<tr>
<td>$24$</td>
<td>TCTN2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$23$</td>
<td>OCR2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$22$</td>
<td>ASSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$21$</td>
<td>WDTCR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$20$</td>
<td>UBRRH</td>
<td>URSR</td>
<td>URSR</td>
<td>UMCSEL</td>
<td>UPM1</td>
<td>UPM0</td>
<td>USBS</td>
<td>UCSZ1</td>
<td>UCZ2</td>
</tr>
<tr>
<td>$1F$</td>
<td>EEARH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1E$</td>
<td>EEARL</td>
<td>EEARH</td>
<td>EEARH</td>
<td>EEARH</td>
<td>EEARH</td>
<td>EEARH</td>
<td>EEARH</td>
<td>EEARH</td>
<td>EEARH</td>
</tr>
<tr>
<td>$1D$</td>
<td>EEDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1C$</td>
<td>EECR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1B$</td>
<td>PORTA</td>
<td>PORTA7</td>
<td>PORTA6</td>
<td>PORTA5</td>
<td>PORTA4</td>
<td>PORTA3</td>
<td>PORTA2</td>
<td>PORTA1</td>
<td>PORTA0</td>
</tr>
<tr>
<td>$1A$</td>
<td>DDRA</td>
<td>DDA7</td>
<td>DDA6</td>
<td>DDA5</td>
<td>DDA4</td>
<td>DDA3</td>
<td>DDA2</td>
<td>DDA1</td>
<td>DDA0</td>
</tr>
<tr>
<td>$19$</td>
<td>PINA</td>
<td>PINA7</td>
<td>PINA6</td>
<td>PINA5</td>
<td>PINA4</td>
<td>PINA3</td>
<td>PINA2</td>
<td>PINA1</td>
<td>PINA0</td>
</tr>
<tr>
<td>$18$</td>
<td>PORTB</td>
<td>PORTB7</td>
<td>PORTB6</td>
<td>PORTB5</td>
<td>PORTB4</td>
<td>PORTB3</td>
<td>PORTB2</td>
<td>PORTB1</td>
<td>PORTB0</td>
</tr>
<tr>
<td>$17$</td>
<td>DDRB</td>
<td>DDB7</td>
<td>DDB6</td>
<td>DDB5</td>
<td>DDB4</td>
<td>DDB3</td>
<td>DDB2</td>
<td>DDB1</td>
<td>DDB0</td>
</tr>
<tr>
<td>$16$</td>
<td>PINB</td>
<td>PINB7</td>
<td>PINB6</td>
<td>PINB5</td>
<td>PINB4</td>
<td>PINB3</td>
<td>PINB2</td>
<td>PINB1</td>
<td>PINB0</td>
</tr>
<tr>
<td>$15$</td>
<td>PORTC</td>
<td>PORTC7</td>
<td>PORTC6</td>
<td>PORTC5</td>
<td>PORTC4</td>
<td>PORTC3</td>
<td>PORTC2</td>
<td>PORTC1</td>
<td>PORTC0</td>
</tr>
<tr>
<td>$14$</td>
<td>DDCR</td>
<td>DDC7</td>
<td>DDC6</td>
<td>DDC5</td>
<td>DDC4</td>
<td>DDC3</td>
<td>DDC2</td>
<td>DDC1</td>
<td>DDC0</td>
</tr>
<tr>
<td>$13$</td>
<td>PINC</td>
<td>PINC7</td>
<td>PINC6</td>
<td>PINC5</td>
<td>PINC4</td>
<td>PINC3</td>
<td>PINC2</td>
<td>PINC1</td>
<td>PINC0</td>
</tr>
<tr>
<td>$12$</td>
<td>PORTD</td>
<td>PORTD7</td>
<td>PORTD6</td>
<td>PORTD5</td>
<td>PORTD4</td>
<td>PORTD3</td>
<td>PORTD2</td>
<td>PORTD1</td>
<td>PORTD0</td>
</tr>
<tr>
<td>$11$</td>
<td>DDRD</td>
<td>DD07</td>
<td>DD06</td>
<td>DD05</td>
<td>DD04</td>
<td>DD03</td>
<td>DD02</td>
<td>DD01</td>
<td>DD00</td>
</tr>
<tr>
<td>$10$</td>
<td>PIND</td>
<td>PIND7</td>
<td>PIND6</td>
<td>PIND5</td>
<td>PIND4</td>
<td>PIND3</td>
<td>PIND2</td>
<td>PIND1</td>
<td>PIND0</td>
</tr>
<tr>
<td>$0F$</td>
<td>SPDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0E$</td>
<td>SPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0D$</td>
<td>SPCR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0C$</td>
<td>UDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0B$</td>
<td>UCSRA</td>
<td>RXC</td>
<td>TXC</td>
<td>UDRE</td>
<td>FE</td>
<td>DOR</td>
<td>PE</td>
<td>U2X</td>
<td>MPCM</td>
</tr>
<tr>
<td>$0A$</td>
<td>UCSRB</td>
<td>RXCIE</td>
<td>TXCIE</td>
<td>UDRIE</td>
<td>RXEN</td>
<td>TXEN</td>
<td>UCSS2</td>
<td>RXB8</td>
<td>TXB8</td>
</tr>
<tr>
<td>$09$</td>
<td>UBRRH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$08$</td>
<td>ACSCR</td>
<td>ACO</td>
<td>ACO</td>
<td>ACO</td>
<td>ACI</td>
<td>ACIE</td>
<td>ACIC</td>
<td>ACIS1</td>
<td>ACIS0</td>
</tr>
<tr>
<td>$07$</td>
<td>ADMUX</td>
<td>REFS1</td>
<td>REFS0</td>
<td>ADLAR</td>
<td>MUX4</td>
<td>MUX3</td>
<td>MUX2</td>
<td>MUX1</td>
<td>MUX0</td>
</tr>
<tr>
<td>$06$</td>
<td>ADCSRA</td>
<td>ADEN</td>
<td>ADCC</td>
<td>ADATE</td>
<td>ADS1</td>
<td>ADIE</td>
<td>ADPS2</td>
<td>ADPS1</td>
<td>ADPS0</td>
</tr>
<tr>
<td>$05$</td>
<td>ADCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$04$</td>
<td>ADCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$03$</td>
<td>TWDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$02$</td>
<td>TWSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$01$</td>
<td>TWAR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$00$</td>
<td>TWBR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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. Several 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 con-
nections. 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 ac-
complished 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 con-
tinuity. 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 specifi-
cally 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 volt-
age 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 $\mu$F electrolytic capacitor
and a 0.1 $\mu$F monolithic capacitor are connected from $V_{cc}$ to ground, in par-
allel 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 $\mu$F tantalum
capacitor. Remember that the long lead is the positive one.
16. To filter the transient current, use a 0.1 $\mu$F monolithic capacitor for each IC.
Place the 0.1 $\mu$F 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)]
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.

![Bipolar NPN Transistor vs. NMOS Transistor](image-url)

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 VCC 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](image)

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](image)

**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.
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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>STD TTL</th>
<th>LSTTL</th>
<th>ALSTTL</th>
<th>HCMOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{CC}$</td>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
</tr>
<tr>
<td>$V_{IH}$</td>
<td>2.0 V</td>
<td>2.0 V</td>
<td>2.0 V</td>
<td>3.15 V</td>
</tr>
<tr>
<td>$V_{IL}$</td>
<td>0.8 V</td>
<td>0.8 V</td>
<td>0.8 V</td>
<td>1.1 V</td>
</tr>
<tr>
<td>$V_{OH}$</td>
<td>2.4 V</td>
<td>2.7 V</td>
<td>2.7 V</td>
<td>3.7 V</td>
</tr>
<tr>
<td>$V_{OL}$</td>
<td>0.4 V</td>
<td>0.5 V</td>
<td>0.4 V</td>
<td>0.4 V</td>
</tr>
<tr>
<td>$I_{IL}$</td>
<td>$-1.6$ mA</td>
<td>$-0.36$ mA</td>
<td>$-0.2$ mA</td>
<td>$-1$ $\mu$A</td>
</tr>
<tr>
<td>$I_{IH}$</td>
<td>40 $\mu$A</td>
<td>20 $\mu$A</td>
<td>20 $\mu$A</td>
<td>1 $\mu$A</td>
</tr>
<tr>
<td>$I_{OL}$</td>
<td>16 mA</td>
<td>8 mA</td>
<td>4 mA</td>
<td>4 mA</td>
</tr>
<tr>
<td>$I_{OH}$</td>
<td>$-400$ $\mu$A</td>
<td>$-400$ $\mu$A</td>
<td>$-400$ $\mu$A</td>
<td>4 mA</td>
</tr>
<tr>
<td>Propagation delay</td>
<td>10 ns</td>
<td>9.5 ns</td>
<td>4 ns</td>
<td>9 ns</td>
</tr>
<tr>
<td>Static power dissipation</td>
<td>10 mW</td>
<td>2 mW</td>
<td>1 mW</td>
<td>0.0025 nW</td>
</tr>
<tr>
<td>at f = 100 kHz</td>
<td>10 mW</td>
<td>2 mW</td>
<td>1 mW</td>
<td>0.17 mW</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>Family</th>
<th>Year</th>
<th>Suppliers</th>
<th>Tech Base</th>
<th>I/O Level</th>
<th>Speed (ns)</th>
<th>Static Current</th>
<th>IOH/IOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACQ</td>
<td>1989</td>
<td>2</td>
<td>CMOS</td>
<td>CMOS/CMOS</td>
<td>6.0</td>
<td>80 μA</td>
<td>-24/24 mA</td>
</tr>
<tr>
<td>ACTQ</td>
<td>1989</td>
<td>2</td>
<td>CMOS</td>
<td>TTL/CMOS</td>
<td>7.5</td>
<td>80 μA</td>
<td>-24/24 mA</td>
</tr>
<tr>
<td>FCTx</td>
<td>1987</td>
<td>3</td>
<td>CMOS</td>
<td>TTL/CMOS</td>
<td>4.1–4.8</td>
<td>1.5 mA</td>
<td>-15/64 mA</td>
</tr>
<tr>
<td>FCTxT</td>
<td>1990</td>
<td>2</td>
<td>CMOS</td>
<td>TTL/TTL</td>
<td>4.1–4.8</td>
<td>1.5 mA</td>
<td>-15/64 mA</td>
</tr>
<tr>
<td>FASTr</td>
<td>1990</td>
<td>1</td>
<td>Bipolar</td>
<td>TTL/TTL</td>
<td>3.9</td>
<td>50 mA</td>
<td>-15/64 mA</td>
</tr>
<tr>
<td>BCT</td>
<td>1987</td>
<td>2</td>
<td>BICMOS</td>
<td>TTL/TTL</td>
<td>5.5</td>
<td>10 mA</td>
<td>-15/64 mA</td>
</tr>
</tbody>
</table>


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 VCC is being embraced by many designers. The lowering of VCC 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.

**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.
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:

\[
\text{fan-out (of LOW)} = \frac{I_{OL}}{I_{IL}} \quad \text{fan-out (of HIGH)} = \frac{I_{OH}}{I_{IH}}
\]

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](image)

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 $I_{IL} = 1.6 \text{ mA}$ and $I_{IH} = 40 \mu\text{A}$. Table C-1 shows $I_{OH} = 400 \mu\text{A}$ and $I_{OL} = 8 \text{ mA}$ for the LS family. Therefore, we have

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

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.
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

<table>
<thead>
<tr>
<th>Buffer</th>
<th>IOH (mA)</th>
<th>IOL (mA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>74LS244</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>74LS245</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

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 discuss 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-
The DDRx register controls the output buffer and the pull-up resistor. 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.

The 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.

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.
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.

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

<table>
<thead>
<tr>
<th>Pin</th>
<th>Fan-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOL</td>
<td>20 mA</td>
</tr>
<tr>
<td>IOH</td>
<td>−20 mA</td>
</tr>
<tr>
<td>IIL</td>
<td>−1 μA</td>
</tr>
<tr>
<td>IIH</td>
<td>1 μA</td>
</tr>
</tbody>
</table>

*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:

\[
Q = CV
\]
\[
\frac{Q}{T} = \frac{CV}{T}
\]

\[\text{since } F = \frac{1}{T} \quad \text{and } \quad I = \frac{Q}{T} \]

\[I = CVF \]

\[\text{now } \quad P = VI = CV^2F \]

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, \( \frac{di}{dt} \), is also increased, resulting in an increase in the inductance voltage \( L \frac{di}{dt} \) 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
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 $\mu$F or 0.1 $\mu$F 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 [$V = L \frac{di}{dt}$].
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 fluc-
tuation of $V_{CC}$ on the board where all the ICs are mounted. To eliminate this, a rela-
tively 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 num-
ber of ICs on the board and the amount of current drawn by each IC, but it is com-
mon to have a single 22 $\mu$F to 47 $\mu$F capacitor for each of the 16 devices, placed
between the $V_{CC}$ and ground lines.

**Crosstalk**

Crosstalk is due to mutual inductance. See Figure C-16. Previously, we discussed self-
inductance, which is inherent in a piece of con-
derator. 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](image)

<table>
<thead>
<tr>
<th>Logic Family</th>
<th>Line Length (in.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>25</td>
</tr>
<tr>
<td>S, AS</td>
<td>11</td>
</tr>
<tr>
<td>F, ACT</td>
<td>8</td>
</tr>
<tr>
<td>AS, ECL</td>
<td>6</td>
</tr>
<tr>
<td>FCT, FCTA</td>
<td>5</td>
</tr>
</tbody>
</table>

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

---

**Figure D-3. IF THEN ELSE Pseudocode versus Flowchart**

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

---

**Figure D-4. IF THEN Pseudocode versus Flowchart**

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

---
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.

Figure D-5. REPEAT UNTIL Pseudocode versus Flowchart

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

<table>
<thead>
<tr>
<th><strong>AVR</strong></th>
<th><strong>8051</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit registers:</td>
<td>32 general-purpose registers (R0 to R31)</td>
</tr>
<tr>
<td>16-bit (data pointer):</td>
<td>X, Y, Z</td>
</tr>
<tr>
<td>Program Counter:</td>
<td>PC (up to 22-bit)</td>
</tr>
</tbody>
</table>

### Input:

- `IN Rn, PINx`
- `(Use R0, R1, ..., R31.)`

### Output:

- `OUT PORTx, Rn`
- `(Use X, Y, or Z.)`

### Loop:

- `DEC Rn`
- `BRNE TARGET`
- `(Using R0-R7)`

### Stack pointer:

- `SP (16-bit)`
- `SP (8-bit)`

### Data movement:

- **From the code segment:**
  - `LPM Rn, Z`
  - `(Use Z only.)`
  - `MOVC A, @A+PC`

- **From RAM using indirect addressing:**
  - `LD Rn, X`
  - `(Use X, Y, or Z.)`
  - `MOV A, @R0`
- `(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`
  - `(Use X, Y, or Z.)`
  - `MOV @R0, A`

- **To RAM using direct addressing mode:**
  - `STS k, X`
  - `(Use X, Y, or Z.)`
  - `MOV RAM_addr, A`
# APPENDIX F

## ASCII CODES

<table>
<thead>
<tr>
<th>Ctrl</th>
<th>Dec</th>
<th>Hex</th>
<th>Ch</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^E</code></td>
<td>0</td>
<td>00</td>
<td>NUL</td>
<td></td>
</tr>
<tr>
<td><code>^A</code></td>
<td>1</td>
<td>01</td>
<td>SOH</td>
<td></td>
</tr>
<tr>
<td><code>^B</code></td>
<td>2</td>
<td>02</td>
<td>STX</td>
<td></td>
</tr>
<tr>
<td><code>^C</code></td>
<td>3</td>
<td>03</td>
<td>ETX</td>
<td></td>
</tr>
<tr>
<td><code>^D</code></td>
<td>4</td>
<td>04</td>
<td>EOT</td>
<td></td>
</tr>
<tr>
<td><code>^E</code></td>
<td>5</td>
<td>05</td>
<td>ENQ</td>
<td></td>
</tr>
<tr>
<td><code>^F</code></td>
<td>6</td>
<td>06</td>
<td>ACK</td>
<td></td>
</tr>
<tr>
<td><code>^G</code></td>
<td>7</td>
<td>07</td>
<td>BEL</td>
<td></td>
</tr>
<tr>
<td><code>^H</code></td>
<td>8</td>
<td>08</td>
<td>BS</td>
<td></td>
</tr>
<tr>
<td><code>^I</code></td>
<td>9</td>
<td>09</td>
<td>HT</td>
<td></td>
</tr>
<tr>
<td><code>^J</code></td>
<td>10</td>
<td>0A</td>
<td>LF</td>
<td></td>
</tr>
<tr>
<td><code>^K</code></td>
<td>11</td>
<td>0B</td>
<td>VT</td>
<td></td>
</tr>
<tr>
<td><code>^L</code></td>
<td>12</td>
<td>0C</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td><code>^M</code></td>
<td>13</td>
<td>0D</td>
<td>CR</td>
<td></td>
</tr>
<tr>
<td><code>^N</code></td>
<td>14</td>
<td>0E</td>
<td>SO</td>
<td></td>
</tr>
<tr>
<td><code>^O</code></td>
<td>15</td>
<td>0F</td>
<td>SI</td>
<td></td>
</tr>
<tr>
<td><code>^P</code></td>
<td>16</td>
<td>10</td>
<td>DLE</td>
<td></td>
</tr>
<tr>
<td><code>^Q</code></td>
<td>17</td>
<td>11</td>
<td>DC1</td>
<td></td>
</tr>
<tr>
<td><code>^R</code></td>
<td>18</td>
<td>12</td>
<td>DC2</td>
<td></td>
</tr>
<tr>
<td><code>^S</code></td>
<td>19</td>
<td>13</td>
<td>DC3</td>
<td></td>
</tr>
<tr>
<td><code>^T</code></td>
<td>20</td>
<td>14</td>
<td>DC4</td>
<td></td>
</tr>
<tr>
<td><code>^U</code></td>
<td>21</td>
<td>15</td>
<td>NAK</td>
<td></td>
</tr>
<tr>
<td><code>^V</code></td>
<td>22</td>
<td>16</td>
<td>SYN</td>
<td></td>
</tr>
<tr>
<td><code>^W</code></td>
<td>23</td>
<td>17</td>
<td>ETB</td>
<td></td>
</tr>
<tr>
<td><code>^X</code></td>
<td>24</td>
<td>18</td>
<td>CAN</td>
<td></td>
</tr>
<tr>
<td><code>^Y</code></td>
<td>25</td>
<td>19</td>
<td>EM</td>
<td></td>
</tr>
<tr>
<td><code>^Z</code></td>
<td>26</td>
<td>1A</td>
<td>SUB</td>
<td></td>
</tr>
<tr>
<td>`^['</td>
<td>27</td>
<td>1B</td>
<td>ESC</td>
<td></td>
</tr>
<tr>
<td><code>^\</code></td>
<td>28</td>
<td>1C</td>
<td>FS</td>
<td></td>
</tr>
<tr>
<td><code>^]</code></td>
<td>29</td>
<td>1D</td>
<td>GS</td>
<td></td>
</tr>
<tr>
<td><code>^^</code></td>
<td>30</td>
<td>1E</td>
<td>RS</td>
<td></td>
</tr>
<tr>
<td><code>^_</code></td>
<td>31</td>
<td>1F</td>
<td>US</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Ch</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>60</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>61</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>62</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>79</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>7A</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>7B</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>7D</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>7E</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>Ch</td>
<td>Dec</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>----</td>
<td>-----</td>
</tr>
<tr>
<td>118</td>
<td>70</td>
<td>Ç</td>
<td>160</td>
</tr>
<tr>
<td>119</td>
<td>71</td>
<td>Ü</td>
<td>161</td>
</tr>
<tr>
<td>120</td>
<td>72</td>
<td>å</td>
<td>162</td>
</tr>
<tr>
<td>121</td>
<td>73</td>
<td>å</td>
<td>163</td>
</tr>
<tr>
<td>122</td>
<td>74</td>
<td>å</td>
<td>164</td>
</tr>
<tr>
<td>123</td>
<td>75</td>
<td>å</td>
<td>165</td>
</tr>
<tr>
<td>124</td>
<td>76</td>
<td>å</td>
<td>166</td>
</tr>
<tr>
<td>125</td>
<td>77</td>
<td>ç</td>
<td>167</td>
</tr>
<tr>
<td>126</td>
<td>78</td>
<td>õ</td>
<td>168</td>
</tr>
<tr>
<td>127</td>
<td>79</td>
<td>å</td>
<td>169</td>
</tr>
<tr>
<td>128</td>
<td>7A</td>
<td>å</td>
<td>170</td>
</tr>
<tr>
<td>129</td>
<td>7B</td>
<td>å</td>
<td>171</td>
</tr>
<tr>
<td>130</td>
<td>7C</td>
<td>å</td>
<td>172</td>
</tr>
<tr>
<td>131</td>
<td>7D</td>
<td>å</td>
<td>173</td>
</tr>
<tr>
<td>132</td>
<td>7E</td>
<td>å</td>
<td>174</td>
</tr>
<tr>
<td>133</td>
<td>7F</td>
<td>Å</td>
<td>175</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>å</td>
<td>176</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>å</td>
<td>177</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>å</td>
<td>178</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>å</td>
<td>179</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>å</td>
<td>180</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>å</td>
<td>181</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>å</td>
<td>182</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>å</td>
<td>183</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>å</td>
<td>184</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>å</td>
<td>185</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>å</td>
<td>186</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>å</td>
<td>187</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>å</td>
<td>188</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>å</td>
<td>189</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>å</td>
<td>190</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>å</td>
<td>191</td>
</tr>
</tbody>
</table>
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.

**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.
## Parts suppliers

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

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Address</th>
<th>Contact Details</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSR Electronics</td>
<td>365 Blair Road, Avenel, NJ 07001</td>
<td>Fax: (732) 381-1572, Mail Order: 1-800-972-2225, In New Jersey: (732) 381-8020</td>
<td><a href="http://www.elexp.com">http://www.elexp.com</a></td>
</tr>
<tr>
<td>Electronix Express</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>365 Blair Road</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avenel, NJ 07001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fax: (732) 381-1572</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mail Order: 1-800-972-2225</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In New Jersey: (732) 381-8020</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.elexp.com">http://www.elexp.com</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Altex Electronics</td>
<td>11342 IH-35 North, San Antonio, TX 78233</td>
<td>Fax: (210) 637-3264, Mail Order: 1-800-531-5369</td>
<td><a href="http://www.altex.com">http://www.altex.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digi-Key</td>
<td>1-800-344-4539</td>
<td>1-800-276-2206 (orders only)</td>
<td><a href="http://www.digikey.com">http://www.digikey.com</a></td>
</tr>
<tr>
<td></td>
<td>(1-800-DIGI-KEY)</td>
<td>(972) 271-5546</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fax: (972) 271-2462</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio Shack</td>
<td></td>
<td></td>
<td><a href="http://www.radioshack.com">http://www.radioshack.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDR Microdevices</td>
<td>1850 South 10th St., San Jose, CA 95112-4108</td>
<td>Fax: (408) 494-1400, Fax: (408) 494-1420</td>
<td><a href="http://www.jdr.com">http://www.jdr.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mouser Electronics</td>
<td>958 N. Main St., Mansfield, TX 76063</td>
<td>1-800-346-6873</td>
<td><a href="http://www.mouser.com">http://www.mouser.com</a></td>
</tr>
<tr>
<td>Jameco Electronic</td>
<td>1355 Shoreway Road, Belmont, CA 94002-4100</td>
<td>Fax: (415) 592-2503</td>
<td><a href="http://www.jameco.com">http://www.jameco.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B. G. Micro</td>
<td>P. O. Box 280298, Dallas, TX 75228</td>
<td>1-800-276-2206 (orders only)</td>
<td><a href="http://www.bgmicro.com">http://www.bgmicro.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(972) 271-5546</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fax: (972) 271-2462</td>
<td></td>
</tr>
<tr>
<td>Tanner Electronics</td>
<td>1100 Valwood Parkway, Suite #100, Carrollton, TX 75006</td>
<td>(972) 242-8702</td>
<td><a href="http://www.tannerelectronics.com">http://www.tannerelectronics.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure G-3. Electronics Suppliers
APPENDIX H

DATA SHEETS

27. Electrical Characteristics

27.1 Absolute Maximum Ratings*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Condition</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIL</td>
<td>Input Low Voltage except XTAL1 and RESET pins</td>
<td>VCC=2.7 - 5.5</td>
<td>-0.5</td>
<td>0.2 VCC</td>
<td>V</td>
</tr>
<tr>
<td>VIH</td>
<td>Input High Voltage except XTAL1 and RESET pins</td>
<td>VCC=2.7 - 5.5</td>
<td>0.6 VCC</td>
<td>Vcc + 0.5</td>
<td>V</td>
</tr>
<tr>
<td>VIL1</td>
<td>Input Low Voltage XTAL1 pin</td>
<td>VCC=2.7 - 5.5</td>
<td>-0.5</td>
<td>0.1 VCC</td>
<td>V</td>
</tr>
<tr>
<td>VIH1</td>
<td>Input High Voltage XTAL1 pin</td>
<td>VCC=2.7 - 5.5</td>
<td>0.7 VCC</td>
<td>Vcc + 0.5</td>
<td>V</td>
</tr>
<tr>
<td>VIL2</td>
<td>Input Low Voltage RESET pin</td>
<td>VCC=2.7 - 5.5</td>
<td>-0.5</td>
<td>0.2 VCC</td>
<td>V</td>
</tr>
<tr>
<td>VIH2</td>
<td>Input High Voltage RESET pin</td>
<td>VCC=2.7 - 5.5</td>
<td>0.9 VCC</td>
<td>Vcc + 0.5</td>
<td>V</td>
</tr>
<tr>
<td>VIL3</td>
<td>Output Low Voltage (Ports A,B,C,D)</td>
<td>IOL = 20 mA, VCC = 5V</td>
<td>0.7</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOL = 10 mA, VCC = 3V</td>
<td>0.5</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VOH</td>
<td>Output High Voltage (Ports A,B,C,D)</td>
<td>IOH = -20 mA, VCC = 5V</td>
<td>4.2</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOH = -10 mA, VCC = 3V</td>
<td>2.2</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>IIL</td>
<td>Input Leakage Current I/O Pin</td>
<td>VCC = 5.5V, pin low (absolute value)</td>
<td>1</td>
<td>μA</td>
<td></td>
</tr>
<tr>
<td>IH</td>
<td>Input Leakage Current I/O Pin</td>
<td>VCC = 5.5V, pin high (absolute value)</td>
<td>1</td>
<td>μA</td>
<td></td>
</tr>
<tr>
<td>RRESET</td>
<td>Reset Pull-up Resistor</td>
<td>30</td>
<td>60</td>
<td>85</td>
<td>kΩ</td>
</tr>
<tr>
<td>RPU</td>
<td>I/O Pin Pull-up Resistor</td>
<td>20</td>
<td>50</td>
<td>85</td>
<td>kΩ</td>
</tr>
</tbody>
</table>

*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)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Condition</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{IL}$</td>
<td>Input Low Voltage except XTAL1 and RESET pins</td>
<td>$V_{CC}=2.7-5.5$</td>
<td>-0.5</td>
<td>0.2 $V_{CC}$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IH}$</td>
<td>Input High Voltage except XTAL1 and RESET pins</td>
<td>$V_{CC}=2.7-5.5$</td>
<td>0.6 $V_{CC}$</td>
<td>$V_{CC}+0.5$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IL1}$</td>
<td>Input Low Voltage XTAL1 pin</td>
<td>$V_{CC}=2.7-5.5$</td>
<td>-0.5</td>
<td>0.1 $V_{CC}$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IH1}$</td>
<td>Input High Voltage XTAL1 pin</td>
<td>$V_{CC}=2.7-5.5$</td>
<td>0.7 $V_{CC}$</td>
<td>$V_{CC}+0.5$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IL2}$</td>
<td>Input Low Voltage RESET pin</td>
<td>$V_{CC}=2.7-5.5$</td>
<td>-0.5</td>
<td>0.2 $V_{CC}$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IH2}$</td>
<td>Input High Voltage RESET pin</td>
<td>$V_{CC}=2.7-5.5$</td>
<td>0.9 $V_{CC}$</td>
<td>$V_{CC}+0.5$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IL3}$</td>
<td>Output Low Voltage (Ports A,B,C,D)</td>
<td>$I_{OL}=20mA, V_{CC}=5V$</td>
<td>0.7</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$I_{OL}=10mA, V_{CC}=3V$</td>
<td>0.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_{OH}$</td>
<td>Output High Voltage (Ports A,B,C,D)</td>
<td>$I_{OH}=-20mA, V_{CC}=5V$</td>
<td>4.2</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$I_{OH}=-10mA, V_{CC}=3V$</td>
<td>2.2</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_{IL}$</td>
<td>Input Leakage Current I/O Pin</td>
<td>$V_{CC}=5.5V$, pin low (absolute value)</td>
<td>1</td>
<td>μA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_{IH}$</td>
<td>Input Leakage Current I/O Pin</td>
<td>$V_{CC}=5.5V$, pin high (absolute value)</td>
<td>1</td>
<td>μA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_{RESET}$</td>
<td>Reset Pull-up Resistor</td>
<td>30</td>
<td>60</td>
<td>85</td>
<td>kΩ</td>
<td></td>
</tr>
<tr>
<td>$R_{PU}$</td>
<td>I/O Pin Pull-up Resistor</td>
<td>20</td>
<td>50</td>
<td>85</td>
<td>kΩ</td>
<td></td>
</tr>
</tbody>
</table>
27.3 Speed Grades

![Figure 27-1. Maximum Frequency vs. VCC.](image)

27.4 Clock Characteristics

27.4.1 External Clock Drive Waveforms

![Figure 27-2. External Clock Drive Waveforms](image)

27.4.2 External Clock Drive

![Figure 27-3. External Clock Drive](image)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>$V_{CC} = 2.7\text{V} \text{ to } 5.5\text{V}$</th>
<th>$V_{CC} = 4.5\text{V} \text{ to } 5.5\text{V}$</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/τ_{CLCL}</td>
<td>Oscillator Frequency</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>τ_{CLCL}</td>
<td>Clock Period</td>
<td>125</td>
<td>62.5</td>
<td>ns</td>
</tr>
<tr>
<td>τ_{CHCX}</td>
<td>High Time</td>
<td>50</td>
<td>25</td>
<td>ns</td>
</tr>
<tr>
<td>τ_{CLCX}</td>
<td>Low Time</td>
<td>50</td>
<td>25</td>
<td>ns</td>
</tr>
</tbody>
</table>
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)

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