SECTION 8.2: AVR FUSE BITS

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

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

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

Table 8-6: Fuse Byte (High)

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

Table 8-7: Fuse Byte (Low)

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

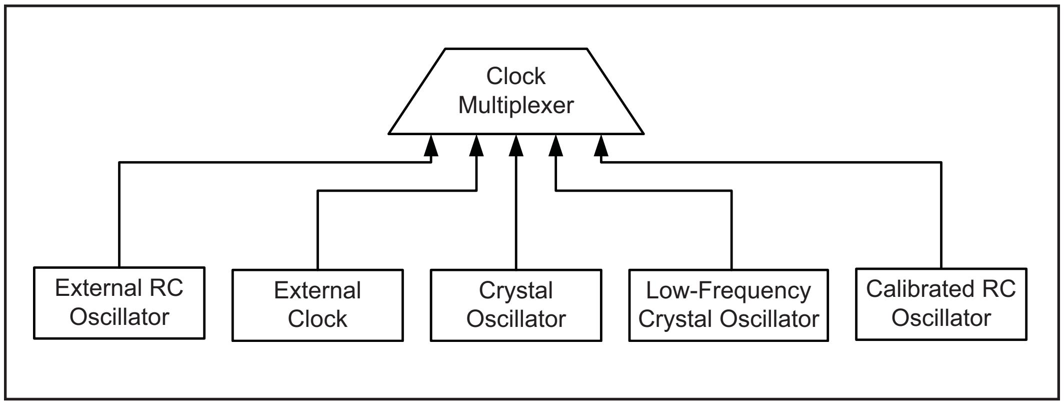


Figure 8-4. ATmega32 Clock Sources

Fuse bits and oscillator clock source

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

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.

Table 8-8: Internal RC Oscillator Operation Modes

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

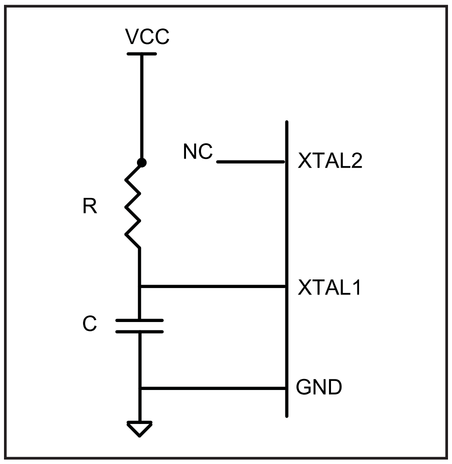


Figure 8-5 External RC

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

Table 8-9: External RC Oscillator Operation Modes

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

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

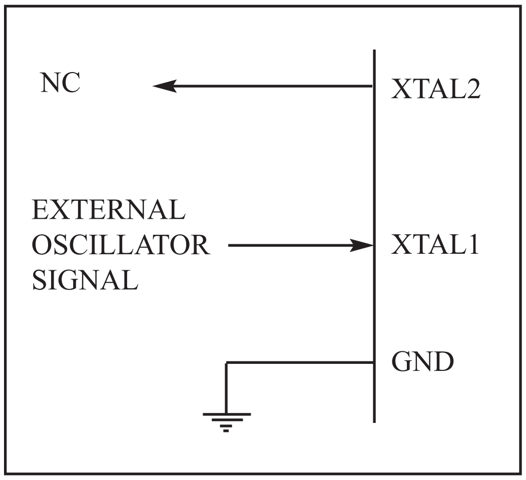


Figure 8-6a. XTAL1 Connection to an External Clock Source

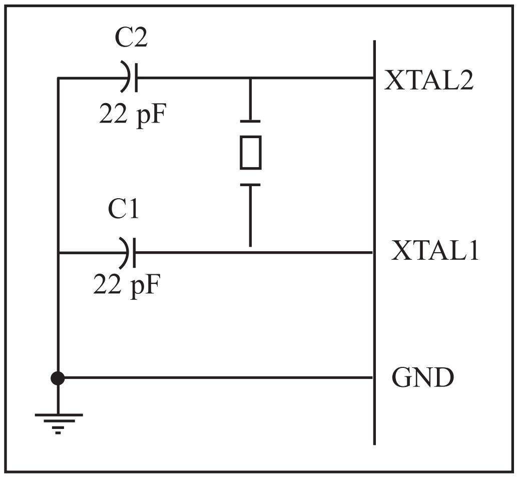


Figure 8-6b. XTAL1–XTAL2 Connection to Crystal Oscillator

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

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

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

Example 8-1

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

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

Fuse bits and reset delay

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

Brown-out detector

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

A good rule of thumb

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

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

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

Putting it all together

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

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

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

...                     ;place your code here
```

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

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

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

      DEC   R20
      BRNE  L1
      RET
```

Program 8-1: Toggling Port B in Assembly

Toggle program in C

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

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

void delay_ms(int d);

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

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

Program 8-2: Toggling Port B in C

Review Questions

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

SECTION 8.3: EXPLAINING THE HEX FILE FOR AVR

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

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

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

Analyzing the Intel Hex file

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

```
:BBAAAATTHHHHH.......HHHHCC
```

The following describes each part:

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

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

Example 8-2

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

Solution:

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

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

Example 8-3

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

```
:020000020000FC
:10000000008E00EBF0FE50DBF0FEF07BB05E500953C
```

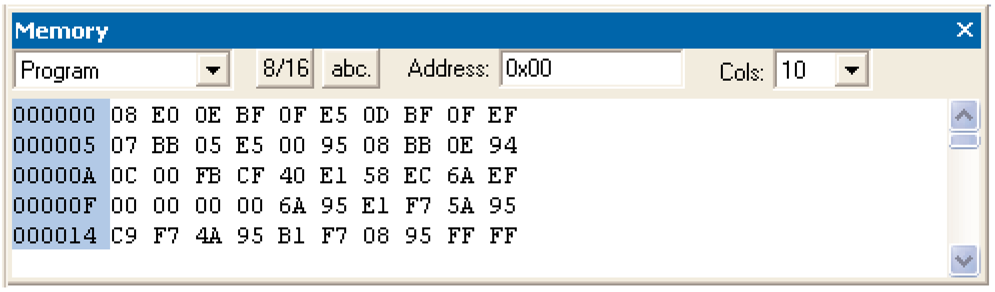
Solution:

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

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

Analyzing the bytes in the Flash memory vs. list file

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



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

Figure 8-7. AVR Flash Memory Contents

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

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

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

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

Flash Memory

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

Figure 8-9. AVR Flash Memory Locations

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

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

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

Separating the fields, we get the following:

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

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

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

Example 8-4

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

Solution:

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

Example 8-5

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

Solution:

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

Example 8-6

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

Solution:

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

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

Review Questions

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

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

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

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

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

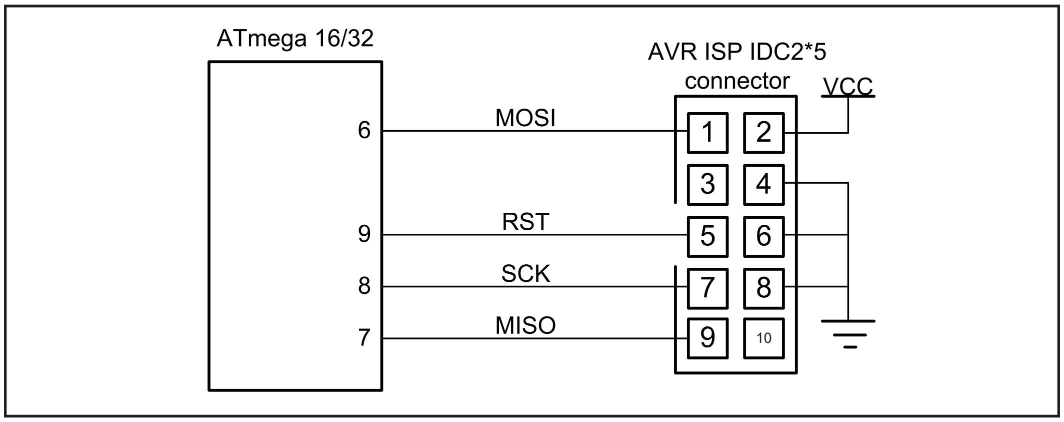


Figure 8-11. ISP 10-pin Connections (See www.Atmel.com for 6-pin version)

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

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

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

AVR trainers

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

Review Questions

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

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

SUMMARY

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

PROBLEMS

SECTION 8.2: AVR FUSE BITS

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

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

SECTION 8.3: EXPLAINING THE INTEL HEX FILE FOR AVR

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

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

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

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

ANSWERS TO REVIEW QUESTIONS

SECTION 8.2: AVR FUSE BITS

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

SECTION 8.3: EXPLAINING THE INTEL HEX FILE FOR AVR

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

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

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