6

HARDWARE AND SOFTWARE INTERFACING WITH THE AVR

This chapter deals with actually putting the AVR processor to some use. It shows how to connect the AVR processor to many I/O devices such as switches, LEDs, displays, ADCs, DACs, motors, etc. To be able to do anything useful with a microcontroller, it needs the combination of appropriate hardware and suitable driver software. So the hardware and software for an embedded application for which the AVR processor could be used are tightly linked, and both of these aspects of a complete system design need to be considered together.

So let's get started and build our first supersimple circuit that will light up an LED. Trivial as it may seem, it nevertheless provides a lot of confidence to a beginner.

6.1 A Beginner's Circuit

If you are new to AVR processors, you probably want to build a simple circuit and run a program that does something. Nothing better than lighting up a LED. The circuit presented here and the code that runs on the processor does just that.

There are three aspects to this simple starting step:

1. Build the hardware on a general-purpose PCB.
2. Write the accompanying code and assemble it on a PC.
3. Program the AT90S1200 processor and plug it into your PCB.

You also need a +5-V power supply or at least three 1.5-V cells arranged in series to get about 4.5 V, which is suitable for running this circuit.

Figure 6.1 illustrates the circuit diagram. The circuit is not fancy at all. After you put the programmed chip into the socket and power the circuit, the LED should glow. Now press the switch connected to the reset pin and the LED should be turned off. Release the switch and the LED should glow again. This indicates that the program is running and it is the program that is lighting up the LED.

The circuit operates at 4 MHz using the external crystal. If you have a AT90S1200A part, then the oscillator components are not required and you can omit the crystal and the 22-pF capacitors. The AT90S1200A has the internal RC oscillator clock enabled, and the processor then runs at about 1 MHz at +5-V supply voltage. The clock speed is not critical in this particular case.

The following program is also available on the CD in the code directory as file led-light.asm.

```
;ledlight.asm
;A beginner's program
;lights up an LED on pin PORTB0
;LED is arranged to sink current into the PORTB0 pin
;assembled using Atmel's avrasm assembler.
;the following .inc file should be placed in the same directory as
;this assembly program
.include "1200def.inc"
.cseg
.org 0
```

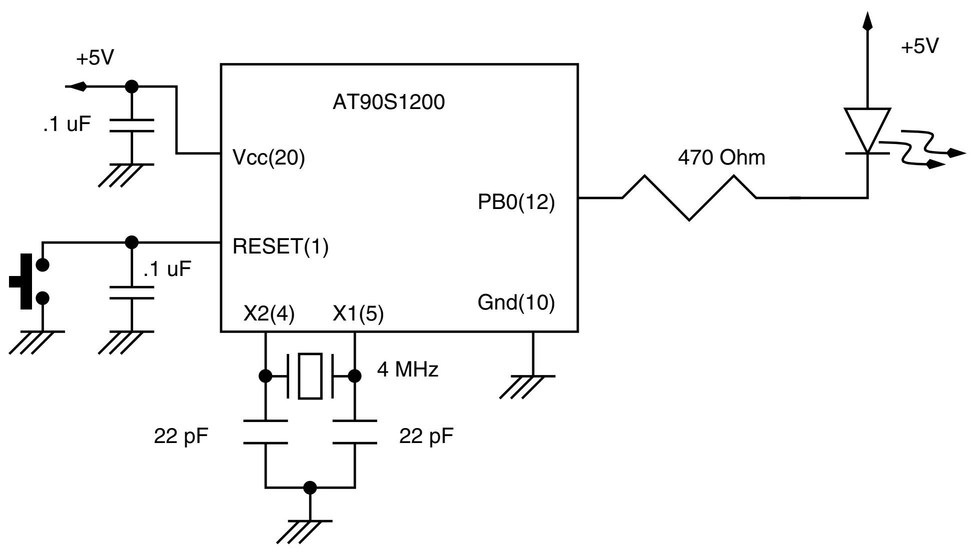


Figure 6.1 A simple introductory circuit to light an LED.

```
        rjmp    RESET              ;Reset Handle
        rjmp    RESET
        rjmp    RESET
RESET:   ldi r16, 0b11111111       ;load register r16 with all 1's
         out DDRB, r16             ;configure PORT B for all outputs
loopit:   ldi r16, 0               ;load register r16 with all 0's
         out PORTB, r16            ;output the contents of r16
                                   ;on PORTB
                                   ;Thus PORTB0 pin is at logic '0'
                                   ;as well as all the other PORTB
                                   ;pins. This enables the current
                                   ;through the LED to flow into
                                   ;PORTB0 pin and the LED lights up
         rjmp loopit
```

6.2 Lights and Switches

Now that we have built a simple beginner's circuit, let's add some input components to the circuit. The simplest input device is a switch. Figure 6.2 illustrates the circuit. The output devices, namely the LEDs, are connected to the PORTB pins, and the input devices, the switches, are connected to the PORTD pins. This keeps our code quite simple. Also, the LEDs and the switches are arranged in a logically symmetrical order and in our code, we map each switch to an LED. To keep code simple, let's map switch on PORTD0 pin to the LED on PORTB0 and so on.

What we want to do is to simply record the state of the switches and copy the state to the corresponding LED. So if we press a switch, thereby putting a logic "0" on the

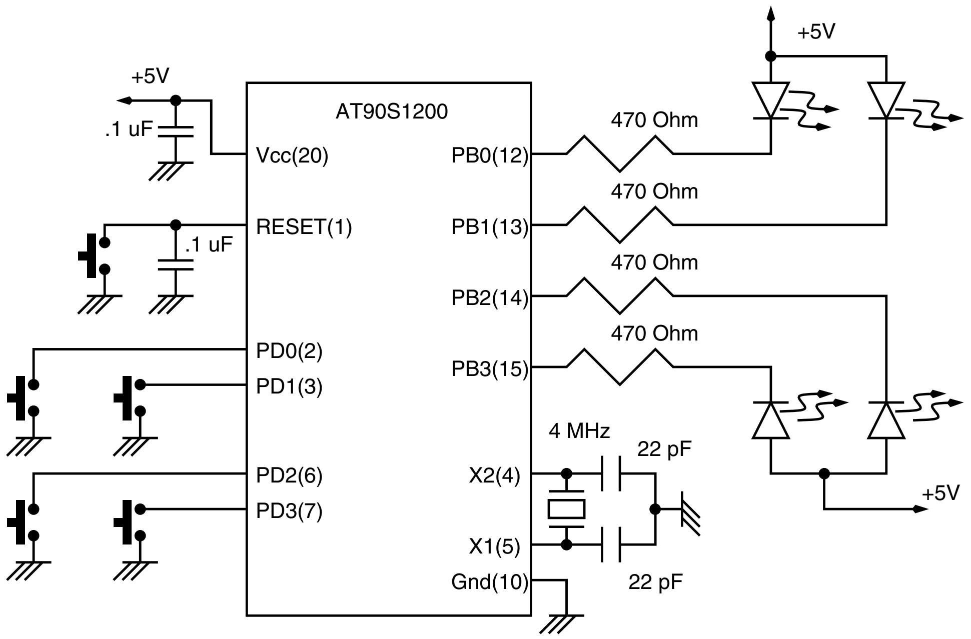


Figure 6.2 Controlling LEDs with switches.

corresponding PORTD pin, we output logic "0" on the matching LED on the PORTB pin. Thus a LED will glow if you press the corresponding switch, and when you release the switch, the LED will stop glowing.

Each port on the AVR processor has three I/O registers associated with it. These registers are called Data Direction register, Output Latch register, and input buffer. These are referred to as DDRx, PORTx, and PINx respectively. So, for portb, these I/O registers are called DDRB, PORTB, and PINB. To output data onto a PORTB pin, you write to the PORTB, and to read data from a PORTB pin, you read the PINB buffer.

The following program is also available on the CD in the code directory as file ledswich.asm.

```
;ledswich.asm
;4 LEDs on PORTB, 4 switches on PORTD
;PORTD0 SWITCH ——-——> PORTB0 LED
;PORTD1 SWITCH ——-——> PORTB1 LED
;PORTD2 SWITCH ——-——> PORTB2 LED
;PORTD3 SWITCH ——-——> PORTB3 LED
;Press one or more switches and corresponding LEDs will lightup
;assembled using Atmel's avrasm assembler.
;the following .inc file should be placed in the same directory as
;this assembly program
.include "1200def.inc"
.cseg
.org 0
       rjmp   RESET             ;Reset Handle
       rjmp   RESET
       rjmp   RESET
RESET:  ldi r16, 0b11111111     ;load register r16 with all 1's
       out DDRB, r16            ;configure PORT B for all outputs
       ldi r16, 0b00000000      ;load register r16 with all 0's
       out DDRD, r16            ;configure PORTD for all inputs
loopit:  in r16, PIND           ;read the state of the pin on PORTD
                                ;into r16 register
       out PORTB, r16           ;and copy it to PORTB
       rjmp loopit
```

The above piece of code shows how to read a switch and light up an LED. However, the switch interfacing is not proper. Typically, a switch, being a mechanical device, doesn't make a clean contact when it is pressed or released.

Figure 6.3 illustrates the signal bounce when a mechanical switch is released. Similar bounce occurs when a switch is pressed. The bounce can last for several milliseconds as illustrated in the figure. Comparatively, the processor executes instructions much faster, up to 1000 times faster or even more. Given such a disparity, if a program were to read a switch and decide to take some action if it is pressed, then even for a single-switch press, it will end up taking the action many, many times. One cure for this problem is to use external damping components such as an RC delay circuit. A better, cost-saving method, which is more elegant, is to provide the damping in software. This software damping scheme is called debouncing the switch.

The way the switch debouncing is performed is as follows: The processor reads the switch input pin, and when it detects a change of logic from "1" to "0" (for the switch configuration as illustrated in Figure 6.2), it knows that the switch has been pressed. It then calls a delay routine, which is of the order of a few milliseconds, say 20 ms (which is the

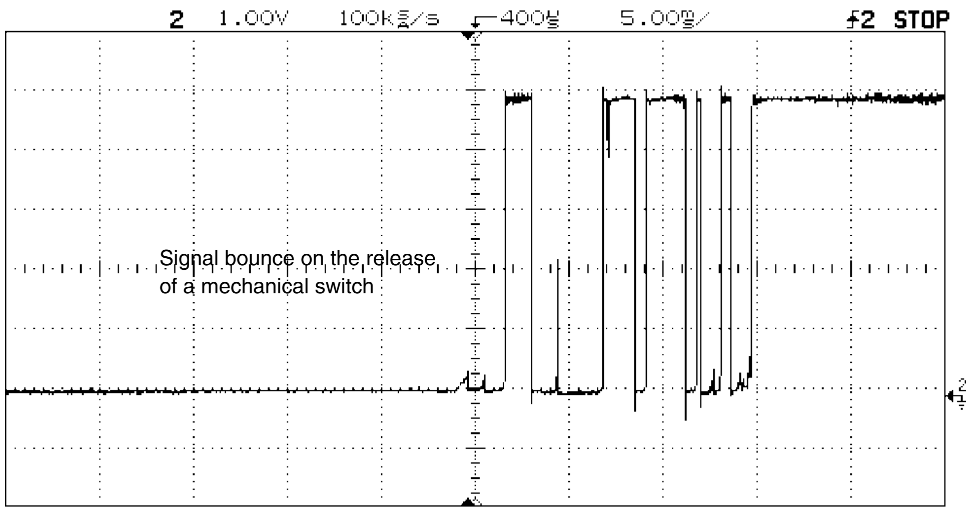


Figure 6.3 Signal bounce on a mechanical switch when it is released.

time for which the signal bounce of the switch remains). After this period, the logic on the switch has stabilized to “0”. The processor then reads the switch input again to ensure that it is still pressed. The processor then enters a software loop and monitors the switch input pin till the switch is released again. The release of the switch is characterized by the logic at the pin changing from logic “0” to logic “1”. After detecting this logic change, the processor again calls a delay routine to timeout the signal bounce on the switch and again checks if the switch has stabilized to a logic value “1”. If so, the program concludes that the particular switch was pressed and released and then can take any action as necessary.

Now, before we write a piece of code to read a switch in the way just described, we need to understand how subroutines are written and called in AVR processors.

6.3 Stack Operation in AVR Processors

Subroutine calls are interruptions in the normal sequential flow of the program. To call a subroutine, the address of the subroutine is loaded into the program counter. The processor then starts executing the code resident at this address and onwards. After the subroutine has finished, the program execution must resume from the point where it was suspended in the calling program. To do that, the processor must remember the address of the program memory from where the execution has to resume. This address is stored in a *stack*. A stack is a special storage area that is used to store return addresses. However, the stack is also used for passing parameters to a subroutine, if required, and to return results to the calling program. Typically, the stack is implemented in RAM and is accessed with a special register called Stack Pointer. Stack Pointer is an address register, and it indicates the address of the RAM memory location of the stack.

In the AVR processors, the stack is implemented in two different ways. For those processors that do not have any SRAM, such as the AT90S1200, the processor has a hardware stack.

The hardware stack is three levels deep, meaning that it can store three return addresses. Thus, at any time, only three nested subroutine calls can be made. The hardware stack is only used by the processor to store return addresses. It cannot be used by the program to pass any parameters to the subroutine, as there is no push or pop instruction to access data on the stack. This may seem like a problem, especially for storing the processor state during an interrupt execution. Since the interrupt occurs asynchronously, the state of the SREG register, which has all the flags, can get changed due to instruction execution within the interrupt subroutine. One way out of this is to store the value of the SREG register into another designated register at the beginning of the interrupt subroutine, and while returning from the subroutine, to restore the value from the designated register back into the SREG register as illustrated below.

```
ISR: mov R0, SREG            ;start of interrupt subroutine
                             ;copy SREG value into R0
;Interrupt subroutine code
; ....
; ....
   mov SREG, R0              ;restore R0 value back into SREG
   reti                      ;return from subroutine
```

On the other hand, for those processors that have on-chip SRAM, the processor implements a stack in the SRAM. The stack can be initialized anywhere in this SRAM area. To initialize the stack, the stack pointer is loaded with the address of the SRAM memory, and after this is done, the stack can be accessed by the push and pop instruction. The stack gets used when a subroutine is called or when an interrupt occurs.

The stack grows from a larger memory address into the lower address. Thus, when some data is pushed, data is stored at the current stack pointer address, and then the stack pointer is decremented. Similarly, when the data is popped from the stack, the stack pointer is first incremented and then the data is copied from the stack to the destination register.

Let's now use this information about calling subroutines and improve our lights-and-switches system so that it will now wait for a switch to be pressed, and after a switch is pressed, it will light up the corresponding LED and wait for another switch. If two switches are pressed, then for the one which is pressed earlier, the LED corresponding to that switch will be lit.

The following program is also available on the CD in the code directory as file newswich.asm.

```
;newswich.asm
;4 LEDs on PORTB, 4 switches on PORTD
;PORTD0 SWITCH ——-——> PORTB0 LED
;PORTD1 SWITCH ——-——> PORTB1 LED
;PORTD2 SWITCH ——-——> PORTB2 LED
;PORTD3 SWITCH ——-——> PORTB3 LED
;Press a switch and corresponding to the LED will light up
;press another switch and the first LED will go off and
;the LED corresponding to the new switch will light up
```

```
;assembled using Atmel's avrasm assembler.
;the following .inc file should be placed in the same directory as
;this assembly program
.include "1200def.inc"
.cseg
.org 0
       rjmp   RESET            ;Reset Handle
       rjmp   RESET
       rjmp   RESET
RESET:  ldi r16, 0b11111111     ;load register r16 with all 1's
       out DDRB, r16            ;configure PORT B for all outputs
       ldi r16, 0b00000000      ;load register r16 with all 0's
       out DDRD, r16            ;configure PORTD for all inputs
       ldi r16, 255             ;all LEDs off
       out PORTB, r16
loopit:  rcall get_switch       ;call the subroutine to
                                ;determine which switch is pressed.
                                ;the subroutine returns the result
                                ;in register r17
      out PORTB, r17            ;output the value on PORTB
      rjmp loopit               ;get more
;—————————————————————————***********——————————————————————————-
;GET_SWITCH: Subroutine to determine which switch is pressed.
;switch on      return value in r17
;  PD0         0b11111110
;  PD1         0b11111101
;  PD2         0b11111011
;  PD3         0b11110111
;registers destroyed: r18, r19
;subroutines called: delay20ms
;—————————————————————————***********——————————————————————————-
get_switch:
      in r18, PIND              ;read PIND buffer
      andi r18, $0F             ;
      cpi r18, $0F              ;if no switch is pressed
                                ;then loop back till pressed
      breq get_switch
      cpi r18, 0b00001110       ;check is SW0 is pressed
      brne not_0                ;if not check more
its_0:  rjmp next_step
not_0:  cpi r18, 0b00001101     ;check is SW1 is pressed
       brne not_1               ;if not check more
its_1:  rjmp next_step
not_1:  cpi r18, 0b00001011     ;check is SW2 is pressed
       brne not_2               ;if not check more
its_2:  rjmp next_step
not_2:   cpi r18, 0b00000111    ;check is SW3 is pressed
       brne get_switch          ;if not some problem, so go back
next_step:
       rcall delay20ms          ;call a debounce delay routine
waitfor_rel:                    ;now wait for the switch to be
       in r19, PIND             ;be released
       andi r19, $0F            ;when the switch is released, all
       cpi r19, $0F             ;PIND0-3 bits will be '1'
       brne waitfor_rel
       rcall delay20ms          ;OK, the switch is released
                                ;debounce it
       mov r17, r18             ;put the switch code in r17
       ori r17, $F0
       ret                      ;and return
```

```
;—————————————***********————————————————-
;DELAY20MS: A 20ms delay subroutine
;Crystal Frequency is 4MHz
;registers destroyed: r21, r20
;—————————————***********————————————————-
delay20ms:
       ldi r21, 31
outer_loop:
       ldi r20, 255
inner_loop:
       nop
       nop
       nop
       nop
       nop
       nop
       nop
       dec r20
       brne inner_loop
       dec r21
       brne outer_loop
       ret
```

6.4 Implementing Combinational Logic

In a previous chapter we mentioned how a controller can be used to implement a simple combinational logic equation as illustrated in Figure 1.3 in Chapter 1.

The figure shows four inputs connected to the PB0, PB1, PB2, and PB3 pins. The output of the circuit is on pin PB4. The following program will implement the logic equation:

```
Output = ((/A * B) + (/B * A)) * (C * /D)
```

The following program (also available on the CD as combi.asm) is only to illustrate how the AVR can be used to implement combinational logic. The program is not optimized. For example, the required output needs an XOR between two inputs, which the AVR can perform. However, I have chosen to implement the XOR using NOT, AND, and OR instructions.

```
;combi.asm
.include "1200def.inc"
.def A=r16
.def Abar=r17
.def B=r18
.def Bbar=r19
.def C=r20
.def Dbar=r21
.def temp=r22
.cseg
.org 0
rjmp RESET                  ;reset handle
RESET:    ldi temp, 0b00001111;
```

```
          out DDRB, temp      ;PB0-3 are inputs
                              ;PB4-7 are outputs
loop_here:  in temp, PINB      ;read PORTB pins
          mov A, temp
          mov Abar, temp
          com Abar            ;invert A
          mov B, temp
          mov Bbar, temp
          com Bbar            ;invert B
          mov C, temp
          mov Dbar, temp
          com Dbar            ;invert D
          andi A, 1           ;isolate the bit for A
          andi Abar, 1        ;isolate the bit for Abar
          lsr B               ;get input B to position bit0
          lsr Bbar
          lsr C               ;get input C to position bit0
          lsr C
          lsr Dbar            ;get input D to position bit0
          lsr Dbar
          lsr Dbar
          andi B, 1
          andi Bbar, 1
          andi C, 1
          andi Dbar, 1
          and A, Bbar         ;A = A * Bbar
          and B, Abar         ;B = Abar * B
          and C, Dbar         ;C = C * Dbar
          or A, B             ;A = (A * Bbar) + (Abar * B)
          and A, C            ;A = ((A * Bbar) + (Abar * B))* (C  * Dbar)
          and A, 1
          cpi A, 1
          breq Its1
          cbi PORTB, 4        ;no its 0, so reset PB4
          rjmp loop_here
Its1:      sbi PORTB, 4
          rjmp loop_here
```

6.5 Connecting the AVR to the PC Serial Port

Now that we have written and tested a couple of programs, it is time to connect the AVR processor to the PC. The simplest port to connect to, on the PC, is the RS-232 serial port. You may want to read the operation of the RS-232 port in detail, presented in a later chapter.

Many of the AVR processors are equipped with a built-in serial port. On the entry-level processors such as the AT90S1200, one can create a software-driven serial port.

This section presents both of these methods. Of course, it is very easy to use the built-in serial port of the AVR processor with only a few instructions. The processor takes care of serializing and shifting out the data on the output pin and assembling the incoming data into a byte. The user needs to set the serial port parameters such as the baud rate (which indicates the bits per second), the number of bits in a transmission, number of stop bits, and parity bit. The processor can generate most of the standard and popular baud rates with a suitable clock frequency.

The serial port of the AVR cannot be connected to the PC serial port rightway. The RS-232 signals are bipolar and in the range of +12 V and −12 V, while the AVR can only handle TTL-level signals (if powered from a +5-V supply). Also, the data as appears on the RS-232 line is inverted. That is to say that when the PC wants to send a logic “0”, the voltage on the RS-232 line is +12 V, and when the PC wants to send out logic “1”, the line voltage is −12 V. So some sort of RS-232 line driver and receiver that converts the RS-232 signal levels to TTL, and vice versa, is needed. Also, performing the signal inversion is needed.

A very popular RS-232 line driver and receiver that I have extensively used is MAX232 from Maxim, as well as the pin-compatible ADM232 from Analog Devices. The circuit schematic for the RS-232 interface is illustrated in Figure 6.19, appearing later in this chapter.

The following piece of code shows how to set up the built-in serial port (called UART in the AVR datasheets) of the AVR processors. The following program is also available on the CD in the code directory as file uartdrv.asm.

The code is executed on the circuit illustrated in Figure 6.4. I have chosen to use the AT90S8515 for this exercise. The 8515 is connected to the PC serial port through a MAX232 level translator chip. On Windows or DOS, run any terminal emulation program and set the baud rate to 9600, 8 data bits, 1 stop bit, and no parity format. Now type any key; the 8515 will light up the ASCII code of the key on the eight LEDs and also increment the code and transmit it back. So if you press “A”, it will send back “B” and so on.

```
;uartdrv.asm
;
.include "8515def.inc"
.def rtemp=r17              ;temporary register
.def rreg=r18               ;register for receiving data
.def treg=r19               ;register for transmitting data
.equ baudrate=$33           ;baud rate of 9600 bps for a clock fre-
                            quency of 8 Mhz
.equ RXC=7                  ;UART receive complete flag( 7th bit of
                            USR register)
.equ UDRE=5                 ;UART data register empty flag(5th bit of
                            USR register)
.cseg
.org 0
rjmp RESET                  ;reset handle
rjmp RESET
rjmp RESET
RESET:  ldi r16, low(RAMEND) ;initialize stackpointer
out SPL,r16
ldi r16, HIGH(RAMEND)
out SPH,r16
ldi r16,255                 ;initialize port B for output
out DDRB,r16
rcall init_uart             ;initialize 8515 for transmit and receive
up: rcall rxcomp            ;receive a byte of data
     mov treg, rreg
     com rreg
out portb,rreg              ;output the data on port B
inc treg                    ;increment the received byte
rcall txcomp                ;transmit the byte
  rjmp up
;*****************************************************
;INIT_UART: Initialize the UART for 9600 bits per second
;        8 data bits, 1 stop bit, no parity
;*****************************************************
```

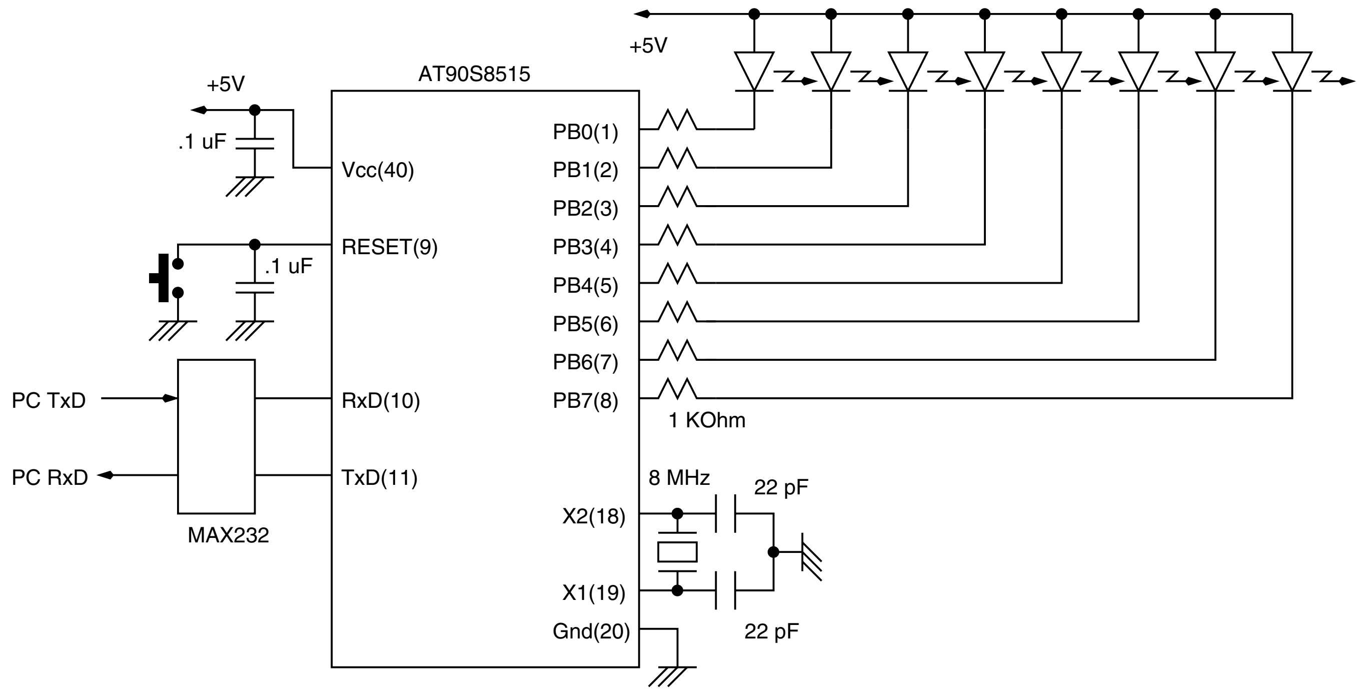


Figure 6.4 **Connecting AT90S8515 to a PC serial port. Other components that go with MAX232 are not illustrated.**

```
init_uart:
      ldi rtemp, baudrate    ;set baud rate
      out UBRR,rtemp
      ldi rtemp, $18         ;initialize UART control register
      out UCR, rtemp
      ret
;*****************************************************
;RXCOMP: Receive a byte from the serial port
;polls the RXC flag in the UART Status Reg (USR)
;if '1', then data is read from the UART Data Register (UDR)
;*****************************************************
rxcomp: sbis USR,RXC         ;poll to check if char received
      rjmp rxcomp
      in rreg,UDR            ;put received data in rreg
      ret
;*****************************************************
;TXCOMP: Transmit a byte from the serial port
;polls to see if the UDRE flag is '1'. If '1' then
;a byte is written to the UDR to be transmitted.
;*****************************************************
txcomp: sbis USR,UDRE        ;poll to check end of transmission
     rjmp txcomp
     out UDR, treg
     ret
```

For those processors that do not have a built-in UART, we describe a software-driven serial port. A software-driven serial port can only be half duplex, meaning that either the serial data can be received or it can be transmitted. A hardware UART, on the other hand, can be full duplex, as the data transmission and reception are being handled by hardware registers that do not need any program intervention in the actual bit-shifting process.

Figure 6.5 illustrates the timing involved in a RS-232 transmission. To receive a serial bit stream, the program must monitor the signal (the TTL signal as illustrated in the figure). The idle state of the serial TTL signal is "1". As soon as a low-going transition is detected, it denotes the beginning of the Start bit and the start of a transmission. The program just monitors the signal again at T/2 time later, which is denoted as the 0th sample. T denotes the bit time. For a 2400-bps speed, the bit time T = 1 / 2400, which is about 416 us. Thus after ensuring that the signal is still "0", the program then just samples the TTL signal at each T time interval after the 0th sample at sample points denoted by 1, 2, 3, etc. The program just records the logic at these sample intervals and shifts the recorded logic values in a register. At the end of eight sample points, the data byte is ready.

Serial data transmission is easy compared to receiving it. The program just generates a start bit for T time units and then shifts out the data to be transmitted, each bit lasting T time units. To get the timing intervals, the AVR processor can use the Timer0 timer, which is available in all the AVR processors.

The software-driven data transmission and reception routine that is included on the CD in fact interfaces directly to the RS-232 port without using any MAX232 type of line converters. The level conversion from RS-232 level to TTL is performed with a few resistors and diodes. The signal inversion is performed in software. For transmission, the TTL data can be directly put on a RS-232 line, and the PC will receive it correctly (the data must be inverted in logic, though).

A working example of this approach is illustrated in a later chapter, and the circuit diagram is illustrated in Figure 17.7. Serial driver (bit-banging method) test code for Figure 17.7 is available in the code directory in file ser_drv.asm.

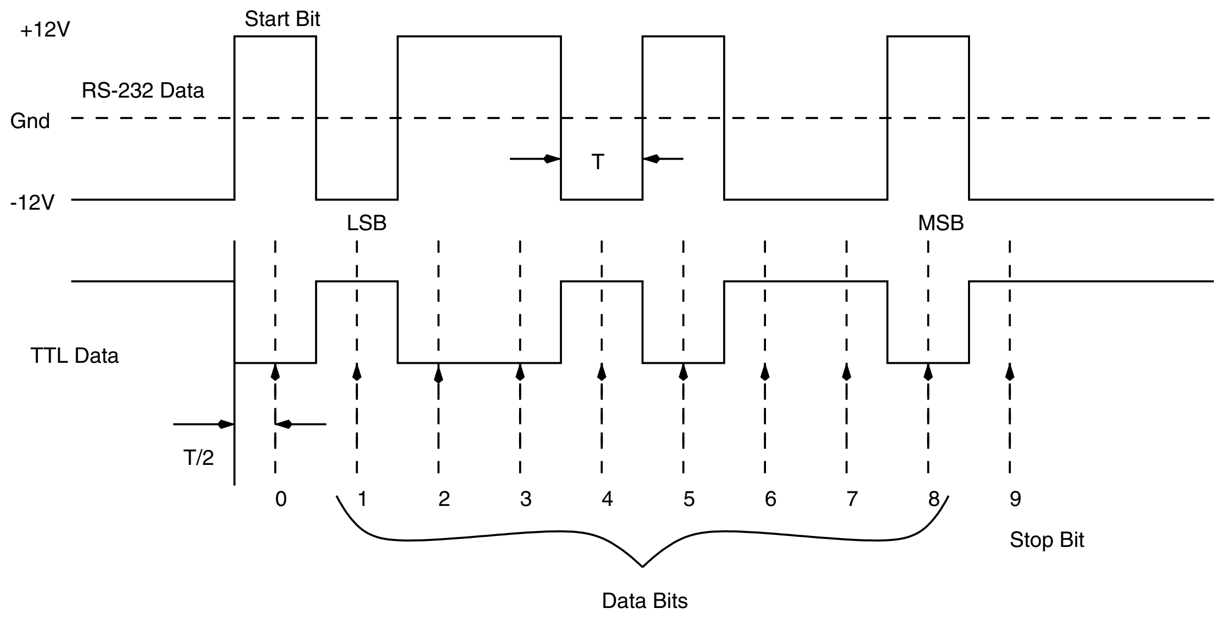


Figure 6.5 **Timing the RS-232 signal. The first bit is the Start bit and the last bit is the Stop bit.**

6.6 Expanding I/O

The AVR processors are available many different pinouts, with different I/O resources, depending upon the number of pins in the particular processor. In some cases, you may feel the need for additional I/O pins than are available. There are many ways to expand the number of I/O pins with the help of shift registers or port expanders with a SPI or I2C interface. This section discusses means of expanding I/O.

The primary requirement is that the I/O expansion scheme should have some serial format so as to take up minimum I/O pins on the processors. Serial Shift registers are great for such applications. Usually, these shift registers are of the serial-in and parallel-out or parallel-in and serial-out format, which suits our requirement. There are many bidirectional I/O expansion ICs with a 2-wire I2C interface available that are, of course, the best in terms of minimum pin usage.

6.6.1 I/O EXPANSION USING SHIFT REGISTER

Figure 6.6 illustrates the scheme for a an 8-bit digital input port using an 8-bit parallel-in, serial-out shift register. This expansion scheme requires 3 I/O pins, and for the cost of 3 I/O pins, you get 8 input-only pins. The 74165 has 5 control lines: serial-in to cascade multiple shift registers, Qout, which is the shift register output, Clock Inhibit to disable clocking of the shift register, Shift/Load*, that is used to capture the input data and shift it out through the Qout pin, and the Clock input pin.

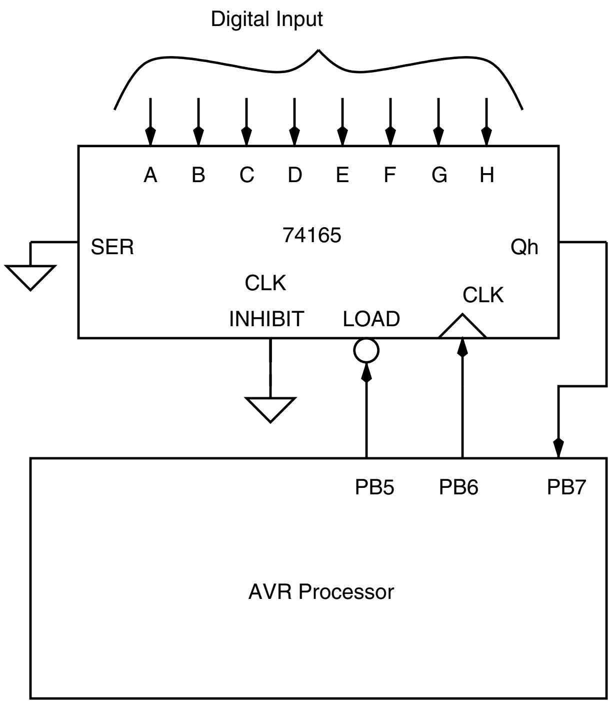


Figure 6.6 **Eight-bit digital input port using a parallel-in serial-out shift register.**

For an 8-bit input port, we need just one 74165, and so the serial-in pin is connected to ground. The clock inhibit pin is also grounded so the clock input is always enabled. The Qout pin is connected to the PORTB7 pin for reading in the shift register data, the Clock signal pin is connected to the PORTB6 pin, and the Shift/Load* pin is connected to the PORTB5 pin.

To read a byte of input data from this expansion port, the Shift/Load* pin is reset to "0" momentarily and then set to "1". This captures the input data in an internal register in the shift register. After this, the Clock signal is pulsed and for each pulse, the PORTB7 pin is read and a bit is shifted out in an internal register. After eight such clock pulses and shifts, the entire byte from the 74165 shift register is read into the AVR processor.

Similarly, Figure 6.7 illustrates an 8-bit output only port. The circuit operates similar to the input port expansion scheme, except that the PORTB7 pin is used to output data to the output shift register CD4094. Eight bits of data are shifted into CD4094, and after eight shifts, the strobe signal for the output stage latch of the CD4094 is set to "1" to transfer the shift register data to the output pins. When the data is being shifted into the shift register, the strobe signal is held at logic "0".

6.6.2 IIC EXPANDERS

In addition to the shift register method of expanding the I/O capacity of an AVR processor, there exists another method to expand I/O capacity. The idea is to use IIC bus-based I/O expander ICs. Manufacturers have perceived the need for increasing the I/O and have designed chips for the purpose. Philips, who is the developer of the IIC bus has designed

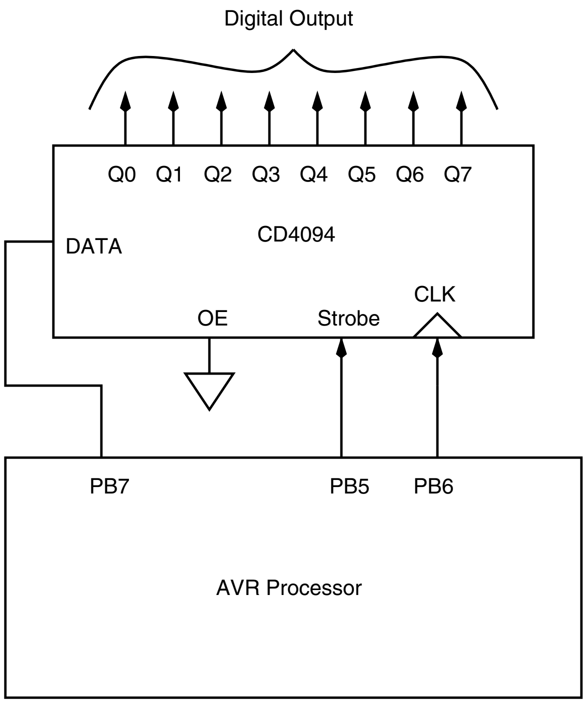


Figure 6.7 **Eight-bit digital output port using a serial-in parallel-out shift register.**

many IIC I/O expanders. Figure 6.8 illustrates the block diagram of just such an I/O expander. It offers one 8-bit bidirectional port. Up to eight such ICs can be hooked on the same IIC bus to achieve more I/O capability.

Figure 6.9 illustrates how the PCF8574 I/O expander IC can be connected to the AVR processor. The INT* output is connected to the INT0 input of the AVR so that by sending an interrupt signal on this line, the remote I/O can inform the microcontroller if there is incoming data on its ports without having to communicate via the I2C-bus. This means that the PCF8574 can remain a simple slave device.

6.7 Interfacing Analog-to-Digital Converters

An analog-to-digital converter (ADC) is a device that converts analog voltage to a digital number. An ADC is used to digitize analog signals. A signal varying with time is sampled at discrete time intervals, and a number representing the amplitude of the signal at the instant is recorded. This is illustrated in Figure 6.10. The code output is on the Y axis and the time is on the X axis. The code output has eight levels, and these can be encoded with three bits. So the encoded binary number ranges from 000 to 111.

There are many types of ADC techniques, and we will not go into those details. I will mention the type of ADC when we consider a particular chip. For now, let’s see how the AVR processor can be used to encode an external analog signal.

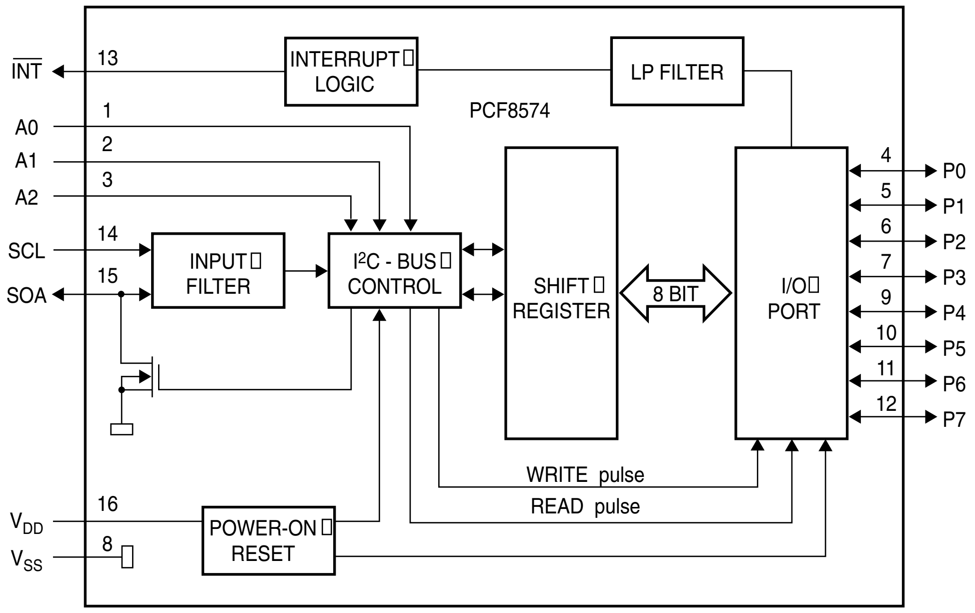


Figure 6.8 **Eight-bit bidirectional digital I/O port expander.**

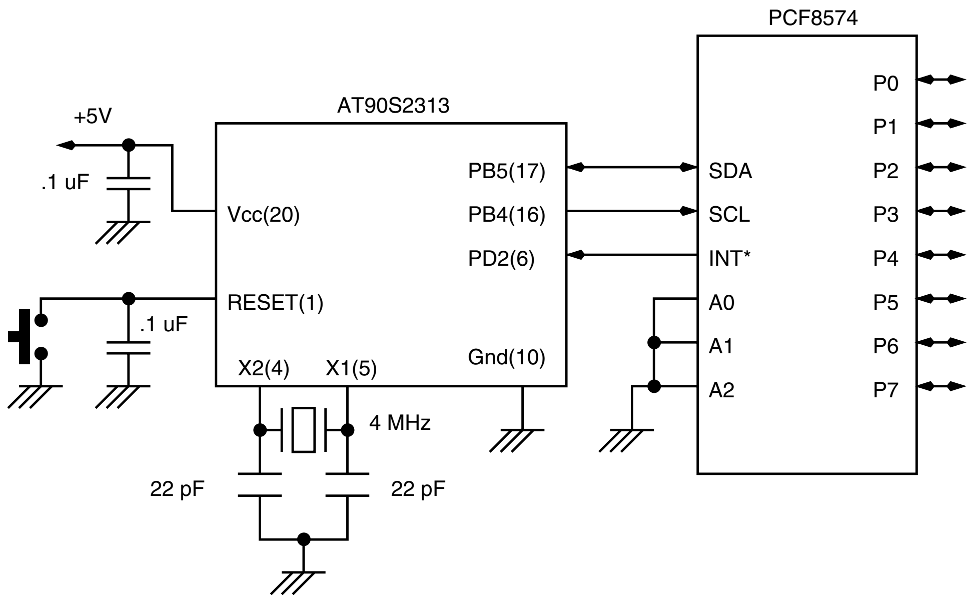


Figure 6.9 AVR interface to PCF8574.

6.7.1 AD CONVERSION USING THE ON-CHIP COMPARATOR

The simplest ADC can be built with the AVR processor by using the on-chip analog comparator together with either a timer or even a counter. (See Figure 6.11.) Figure 6.12 illustrates a rather crude ADC using the on-chip analog comparator. The comparator compares the voltages on the +ve input Ain0 and the −ve input Ain1 and if the Ain0 voltage is greater (i.e., more positive) than the Ain1 voltage, the output of the comparator ACO is set to "1". The ACO output is directly readable as a bit in the ACSR register.

The simple ADC in Figure 6.12 works as follows. To begin with, the PB0 pin is set to logic "0". This discharges any charge on the capacitor. Then the PB0 is pin programmed as an input with no pull-up resistors, and either a software counter or Timer0 (or Timer1) is triggered to start counting. The capacitor starts charging to +5 V through the resistor R1. When the voltage on the capacitor becomes more than input voltage on the Ain1 pin, the comparator output switches "1". When this is detected by the program, which is polling in a loop to detect the change of state of the comparator to "1", the software counter (or the Timer0) is stopped and the accumulated count is proportional to the input voltage on Ain-pin. The larger the voltage, on Ain-, the capacitor will have to charge to a voltage higher than that, and that will take more time, which means the internal counter will be clocked for more time, accumulating a larger count. There is only one glitch (and a rather undesirable one) in this approach, and that is the voltage on the capacitor does not increase linearly but exponentially. So the accumulated count is not linearly proportional to the input voltage. However, by restricting the input to a small range of say 0 to 2.5 V, a fairly linear region of the RC charging curve would be used. A normalized plot of the difference between the count generated by the RC charging method and the true count is provided by the plot illustrated in Figure 6.13. The input range of voltage is restricted between 0 and 2.5 V.

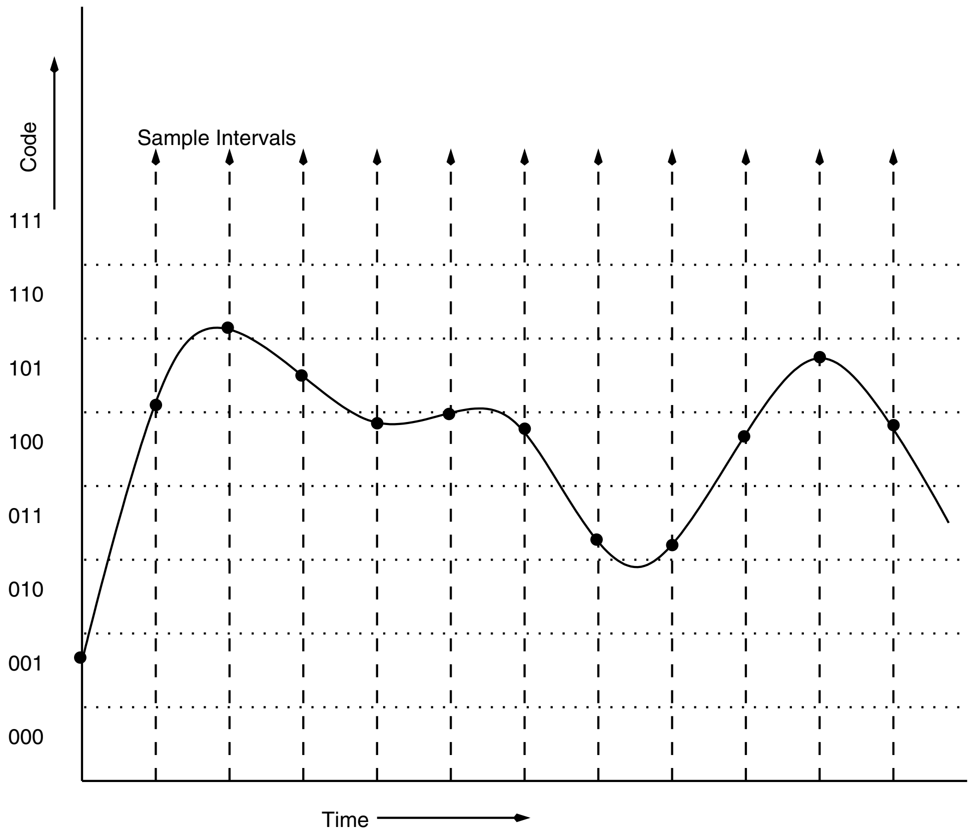


Figure 6.10 An analog signal being sampled and encoded by an ADC. The number output of the ADC is on the Y axis and the time is on the X axis.

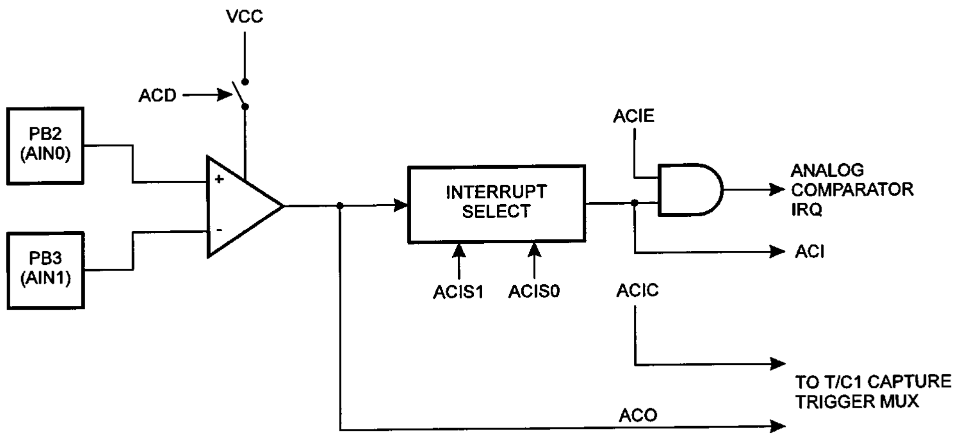


Figure 6.11 Analog comparator block diagram.

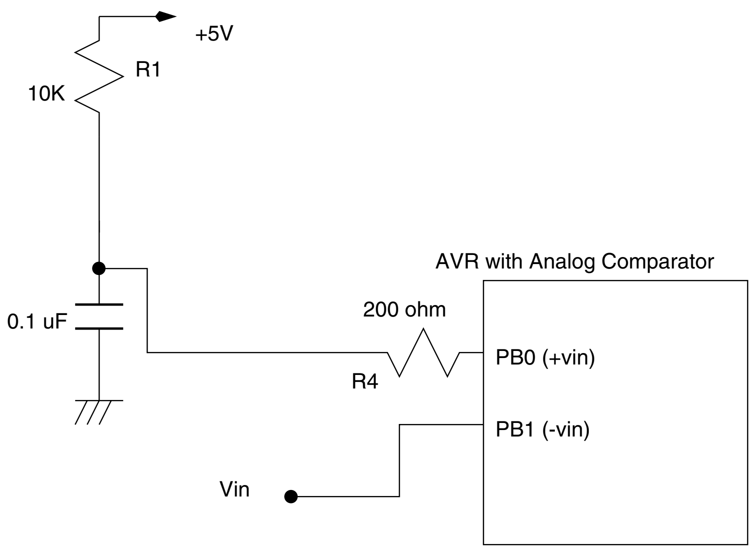


Figure 6.12 **Block diagram for a crude analog-to-digital converter using the on-chip comparator on an AVR processor.**

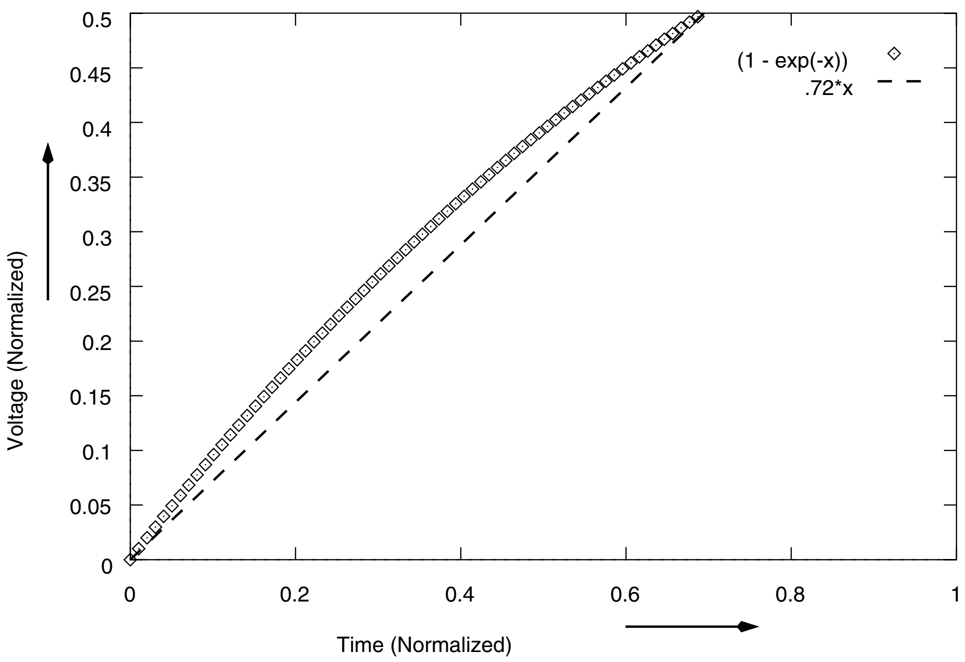


Figure 6.13 **A linear and an exponential plot for a small input range. This plot gives an idea of the amount of nonlinearity between the count accumulated using the simple RC charging scheme and the ideal count.**

For the values illustrated, the RC time constant is 1 ms, and so the capacitor will charge to 2.5 V in about 720 μs. Thus the worst-case conversion time by the scheme is 720 μs.

The nonlinearity in the RC charging can be removed with the help of a scheme as illustrated in Figure 6.14, where the resistor R1 is replaced with a transistor current source. A constant current source will charge the capacitor linearly, and so the accumulated count will be linearly proportional to the input-applied voltage on the Ain-pin.

With the values illustrated in Figure 6.14, the current sourced by the PNP transistor is about 80 μA. The charging of a capacitor with a constant current source is expressed with the equation:

$$dv/dt = I/C$$

Therefore, the time to charge a capacitor from 0 to 5 V is

$$T = (C * 5) /I$$

Plugging in the values, T = 6:25 ms. Thus the worst-case conversion time with this scheme is 6.25 ms. This can be modified by changing the current provided by the current source.

The conversion time is long enough to extract a 10-bit or even 12-bit resolution—i.e., a 10- or 12-bit counter (either software or with the help of Timer0 or Timer1) can be filled up easily during the conversion time.

The resistor R4 is used to limit the capacitor discharge current to within safe values. Without this resistor, the PB0 input would get damaged from the capacitor discharging a large transient pulse into the PB0 pin.

One of the possible applications of the improved ADC is as a temperature recorder. Figure 6.15 illustrates the circuit. LM335 is a temperature sensor that provides a voltage output proportional to the ambient temperature. The voltage generated by the sensor is 10 mV/K. This temperature operates from −40°C to +100°C temperature. So at room tem-

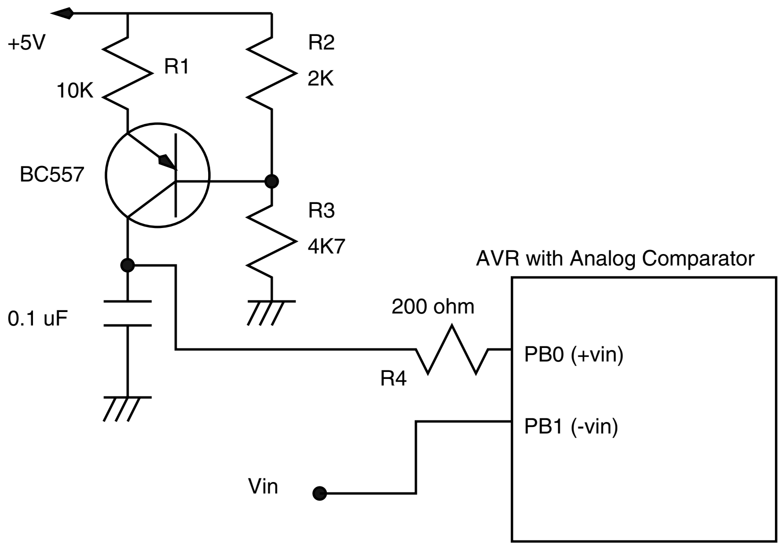


Figure 6.14 **Block diagram for an improved analog-to-digital converter using the on-chip comparator on an AVR processor.**

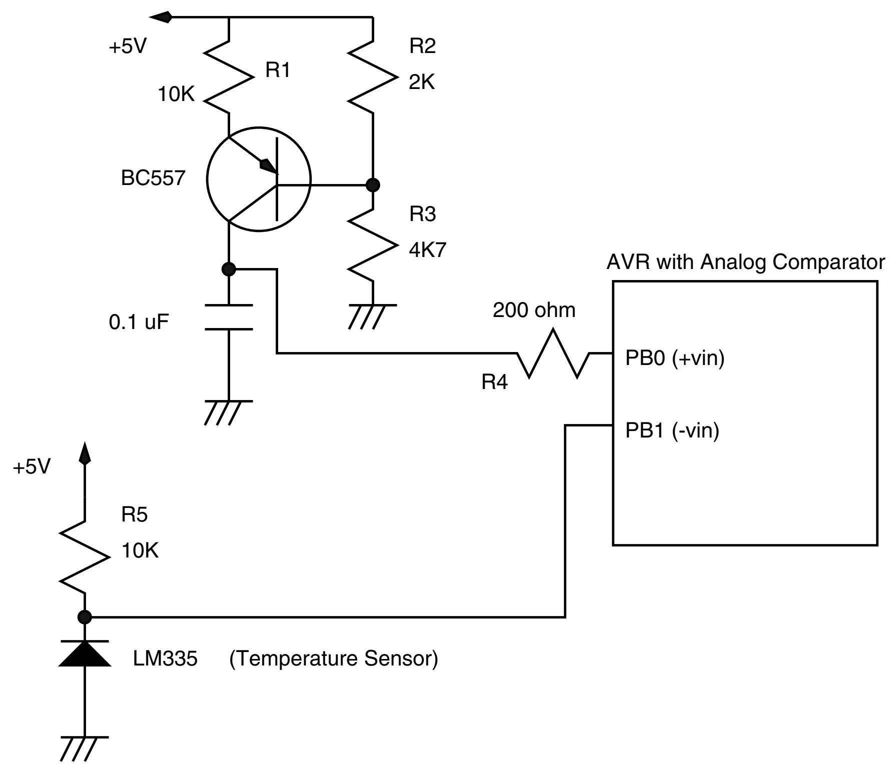


Figure 6.15 **Block diagram for a temperature sensor interface to the comparator-based ADC.**

perature (20°C, which is 393 K), the voltage would be 3.93 V. A simple AT90S1200 can be used for this application and the data can be either stored in the internal EEPROM or transmitted to a PC using the software-driven serial link as described in a previous section.

6.7.2 MAX186

MAX186 is a 12-bit, 8-channel serial ADC system with built-in voltage reference, internal sample and hold amplifier, and multiplexer. The ADC offers various modes of operation such as single-ended conversion, differential conversion, sleep mode, etc. The maximum current conversion is 2 mA and 100 μA during low power modes (sleep mode).

MAX186 is a complete ADC system combining an 8-channel analog multiplexer, sample and hold amplifier, serial data transfer interface and voltage reference, and a 12-bit resolution successive approximation converter. All of these features are packed into a 20-pin DIP package (other packaging styles are also offered). The IC consumes extremely low power and offers power-down modes and high conversion rates. The power-down modes can be invoked in software as well as hardware. The IC can operate from a single +5-V as well as ±5-V power supply.

The analog inputs to the ADC can be configured via software to accept either unipolar or bipolar voltages. The inputs can also be configured to operate as single-ended inputs or differential inputs. The ADC has an internal voltage reference source of 4.096 V, but the user can choose not to use this reference and supply an external voltage between +2.50 V and +5.0 V. This gives the user the advantage of adjusting the span of the ADC according to the need—e.g., if the input analog voltage is expected to be in the range of 0 to +3.0 V, then choosing a reference voltage of 3.0 V will provide the user with the entire ADC input range with a better resolution.

This ADC is an extremely fast device. It can convert at up to 133000 samples per second at the fastest serial clock frequency. This ADC is best suited for devices that can generate fast serial-controlled clocks—e.g., DSPs and microcontrollers such as the AVR. Figure 6.16 illustrates the block diagram of the ADC and the various associated signals. The description of the ADC signals is listed in Table 6.1.

6.7.3 MAX186 DATA CONVERSION AND READOUT

While the many details of this very fine ADC can be had from the IC manufacturer data sheets, our intention here is to see how we can connect this device to the AVR processor to begin with and how a conversion can be initiated and the result read out into the AVR processor.

To initiate a conversion, the ADC must be supplied with a control byte. The control byte is input into the ADC through the Din signal input. To clock the control byte, either an internally or externally generated clock signal (on SCLK pin) could be used. To keep the hardware small and simple, it is necessary to use the external clock mode. The format of the control byte is illustrated in Figure 6.17.

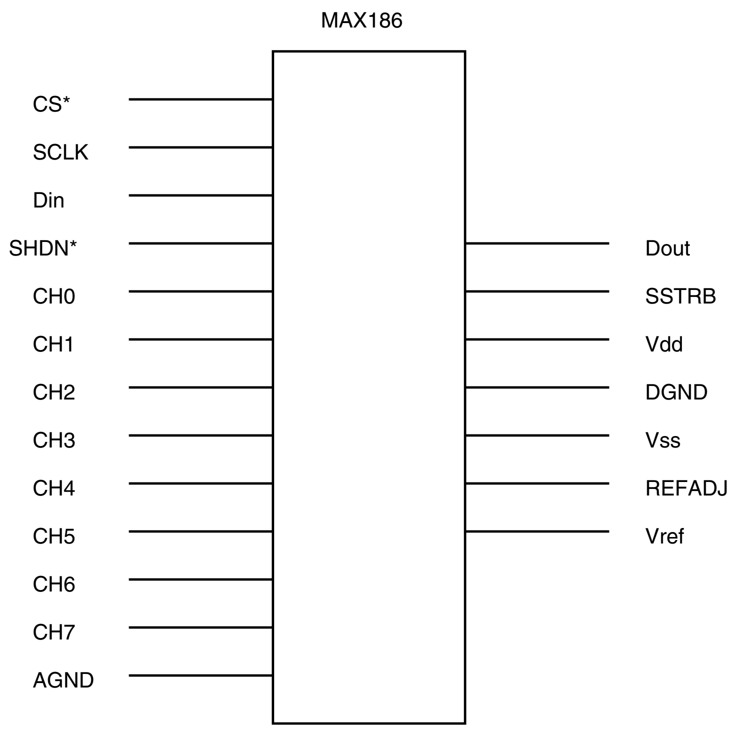


Figure 6.16 **Block diagram of MAX186 ADC.**

TABLE 6-1 ADC MAX186 SIGNALS AND THEIR FUNCTIONS

SIGNAL NAME	FUNCTION
CS*	Active low-chip select input
SCLK	Serial clock input. Clocks data in and out of the ADC. In the external clock mode, the duty cycle must be 45% to 55%.
Din	Serial data input. Data is clocked at the rising edge of SCLK.
SHDN*	Three-level shutdown input. A low input puts the ADC in low-power mode and conversions are stopped. A high input puts the reference buffer amplifier in internal compensation mode. A floating input puts it in external compensation mode.
CH0-CH7	Analog inputs.
AGND	Analog ground and input for single-ended conversions.
Dout	Serial data output. Data is clocked out at the falling edge of SCLK.
SSTRB	Serial strobe output. In external clock mode, it pulses high for one clock period before the MSB decision.
DGND	Digital ground.
Vdd	Positive supply voltage. +5 volts ±5%.
Vss	Negative supply voltage. −5 volts ±5% or AGND.
REFADJ	Input to the reference buffer amplifier.
Vref	Reference voltage for AD conversion. Also output of the reference buffer amplifier (+4.096 volts). Also, input for an external precision reference voltage source.

To clock the control byte into the ADC, the CS* pin is pulled low and a rising edge on SCLK clocks a bit into Din. The control byte format requires that the first bit to be shifted in should be “1”. This defines the beginning of the control byte. Until this start bit is clocked in, any number of “0” can be clocked in by the SCLK signal without any effect.

The control byte must be 1XXXXXX11 (binary). Xs denote the bits required for channel and conversion mode selection. The two least-significant bits are set to ‘1’ and “1” to select the external clock mode option.

Figure 6.17 illustrates the control byte format. The control byte value for starting a conversion on channel 0 of the ADC, in unipolar, single-ended conversion mode using external clock, is 10001111 (binary) or 8F hex.

Let’s now consider the timing diagram in Figure 6.18, which illustrates the conversion and readout process on ADC channel 0.

The timing diagram illustrates five traces, namely CS*, the chip select signal; SCLK, the serial clock required for programming the ADC and the subsequent readout; the Din, which carries the programming information (the control byte); the SSTRB, which the ADC generates to indicate the beginning of the readout process; and Dout, the actual data output from the ADC, which is the conversion result.

The data on signal Din is clocked into the ADC at the rising edge of the SCLK signal.

MSB							LSB
START	SEL2	SEL1	SEL0	UNI/BP*	SGL/DF*	PD1	PD0

START: The first logic '1' bit after CS* goes low defines the start of the Control byte

SEL2, SEL1, SEL0: These 3 bits select which of the 8 channels will be used for conversion

UNI/BP*: 1=Unipolar; input can range between 0 to +Vref;
0=Bipolar; input can range between +Vref/2 to -Vref/2

SGL/DF*: 1=single ended; 0=Differential

PD1, PD0: Defines clock & power down modes.

0	0	: Full power down mode
0	1	: Fast power down mode
1	0	: Internal clock mode
1	1	: External clock mode

Figure 6.17 **MAX186 control byte format.**

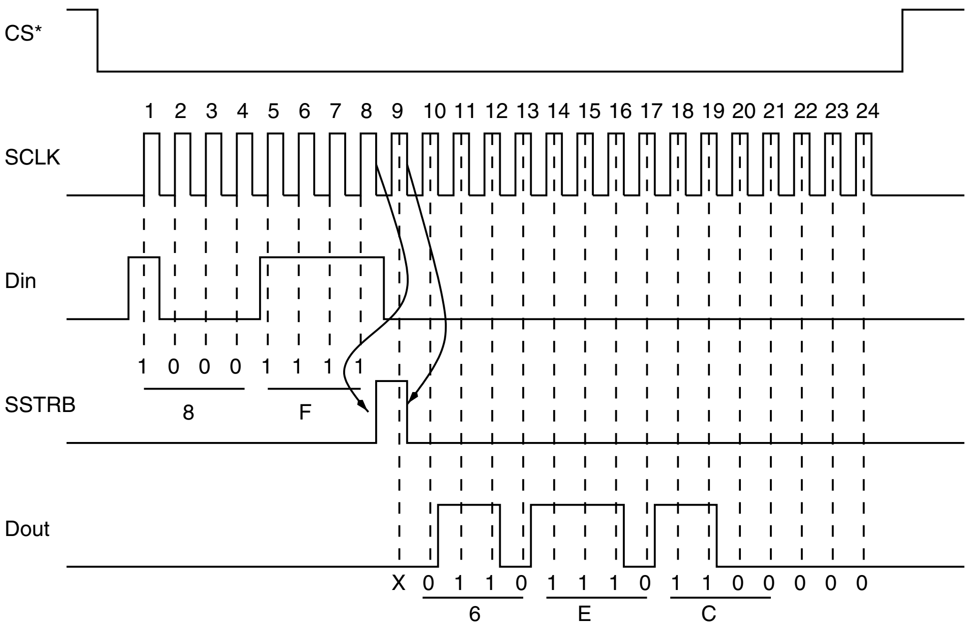


Figure 6.18 **Timing diagram of a typical MAX186 conversion process as recorded on a logic analyzer.**

The first bit that is clocked in is D7. To begin the conversion, D7 needs to be set to "1", as can also be seen from the value of the control byte that we calculated. So Din is set to "1" and the first SCLK rising edge is applied to the ADC. The SCLK is then taken low.

Thereafter, the Din is set to each of the subsequent bits of the control byte before applying the SCLK. At the end of 8 SCLK pulses, the Din bit is not required and is set to "0". At the falling edge of the 8th SCLK pulse, the ADC sets the SSTRB bit to "1". At the falling edge of the 9th SCLK bit, SSTRB is taken to "0".

At the falling edge of the 9th SCLK signal, the ADC outputs data on the Dout signal, one bit for each of the next 15 falling edges of the SCLK signal. The data on the 9th pulse is "0" and the actual conversion result is effective after the 10th falling edge to the 21st rising edge. Thereafter, for the next 3 edges, the ADC outputs "0"s.

For a controller circuit such as the AVR, with minimal parts, to initiate conversion and readout the result would need three output bits and one input bit. The output bits would be needed to generate the Din and SCLK signal and the input bit to read the Dout signal from the ADC.

Figure 6.19 illustrates the circuit with a MAX186 ADC, an AT90S2313 processor, and a MAX232 RS-232 level translator. The circuit is connected to the PC serial port. The AVR processor waits for a command from the PC and then initiates conversion on the MAX186 and sends out the data back to the PC serial port.

The program is available on the CD. The code for this project is available in the code directory in the file MX186_ex.asm.

6.7.4 MAX110/MAX111

MAX111/MAX110 is a serial 14-bit, dual-channel ADC from Maxim. MAX111-/MAX110 ADC uses an internal autocalibration technique to achieve 14-bit resolution without any external component. The ADC offers two channels of ADC conversion and operates with 650μA current, thus making it ideal for portable, battery-operated data acquisition operations.

MAX111 operates from a single +5-V power supply and converts differential signals in the range of ±1.5 V or differential signals in the range of 0 to 1.5 V.

MAX111 can operate from an external as well as internal oversampling clock that is used for the ADC conversion. To start a conversion, digital data is shifted into the MAX111 serial register after pulling the CS low. CS can only be pulled low when BUSY is inactive. MAX111 has a fully static serial I/O shift register which can be read at any serial clock (SCLK) rates from DC to 2 MHz. Input data to the ADC is clocked in at the rising edge of the SCLK and the output data from the ADC (conversion result) is clocked out at SCLK falling edge and should be read on SCLK rising edge.

The data clocked into the ADC determines the ADC operation, which could be to initiate a new conversion, calibrate the ADC, perform offset null, change ADC channel, change oversampling clock divider ratio, etc.

The format of this control word is as follows:

```
bit #  15     14     13     12     11     10     9     8
     No-op    NU     NU   CONV4  CONV3  CONV2  CONV1  DV4
bit #   7     6     5     4     3     2     1     0
       DV2    NU     NU    CHS    CAL    NUL   PDX     PD
```

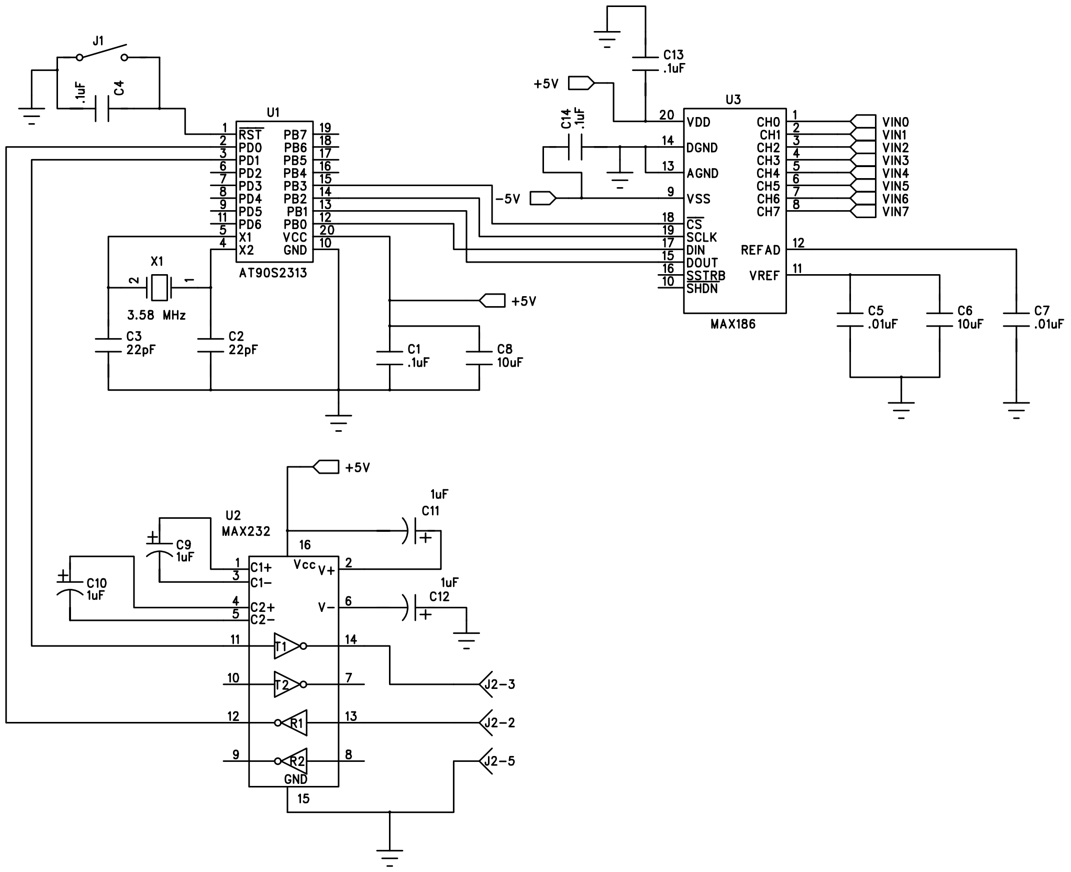


Figure 6.19 Circuit schematic for an AT90S2313 processor interface to the MAX186 ADC.

BIT NAME	FUNCTION
No-OP	If this bit is 1, the remaining 15 bits are transferred to the control register and a new conversion begins when CS* returns high.
NU	Not Used, should be set low.
CONV1-4	Conversion time control bits.
DV4-2	Oversampling clock ration control bits.
CHS	Input channel select, logic 1 selects channel 2, low selects channel 1.
CAL	Gain Calibration bit. A high bit selects gain calibration mode.
NUL	Internal Offset Null bit. Logic high selects this mode.
PDX	Oscillator power down bit, selected with logic high.
PD	Analog power down bit selected with logic high.

Figure 6.20 illustrates an AVR processor interfaced to the MAX111 ADC.

The AVR controller monitors the status of BUSY* signal, which indicates if the ADC is busy with a conversion. A “0” on this pin indicates that the ADC is still converting. The program reads the status of BUSY* on the PORTD2 pin. When the program finds BUSY* at logic “1”, it pulls the CS* signal of the ADC low to start a new conversion process.

It then generates 16 clock pulses on the PORTD5 pin connected to the SCLK signal pin of the ADC. Synchronized to these pulses, the program generates a serial bit stream on pin PORTD4 connected to the Din pin of the ADC. This bit stream contains the control

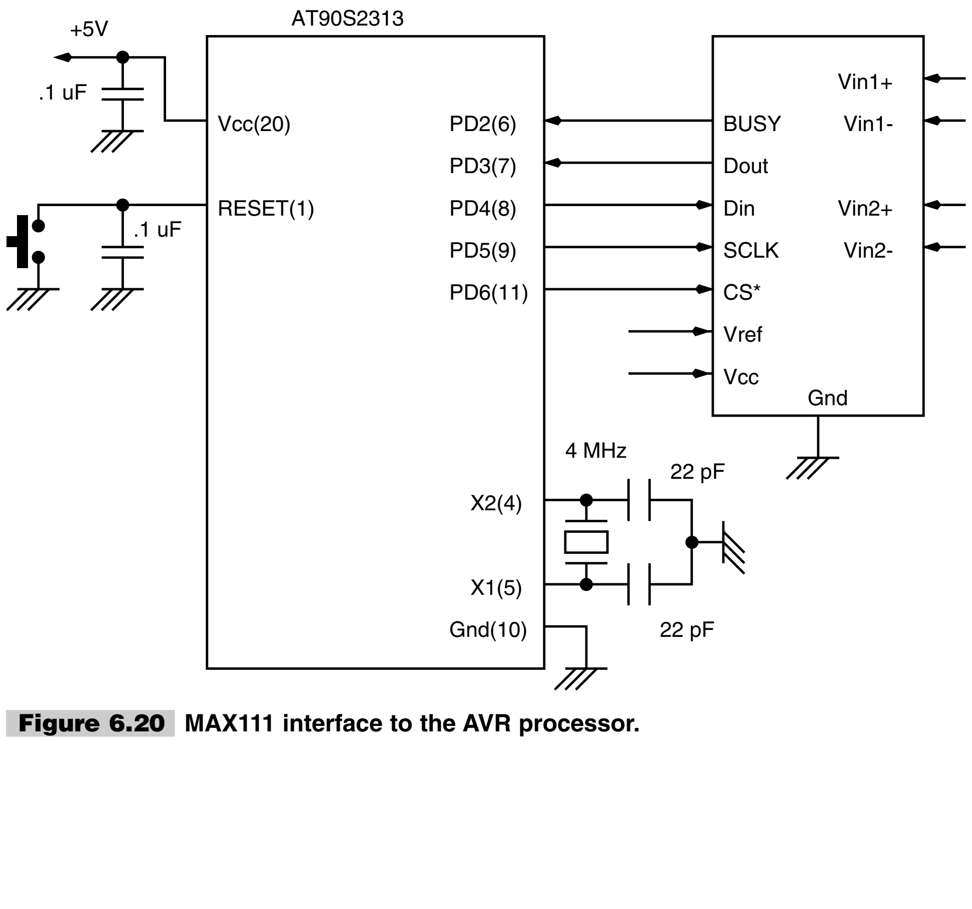


Figure 6.20 MAX111 interface to the AVR processor.

word with the format described previously. Output data from the ADC is clocked out on Dout pin on the falling edges of the SCLK pulses. The program reads this data on the PORTD3 pin. The CS* signal connected to the PORTD6 pin is pulled up after the 16 clock pulses are generated.

The ADC pulls its BUSY* signal low while the conversion is in progress. The conversion time depends upon the XCLK frequency and the format of the control word. In this circuit, the internal RC oscillator is used for the conversion clock. The converted data is clocked out in the next round of the clocking sequence by the ADC.

Figure 6.21 illustrates the timing diagram of a typical conversion and readout sequence recorded on a logic analyzer. A suitable data conversion and readout driver code is included in a later project chapter. The driver program is in “C”.

6.8 Interfacing Digital-to-Analog Converters

Digital-to-Analog Converters (DACs) are devices that function exactly opposite to the ADCs. DACs convert digital data to analog voltage (or current). Functionally, the DAC has *n* digital input lines and 1 output line that provides analog voltage or current. The analog output is proportional to the weighted sum of the digital inputs.

6.8.1 USING PWM FOR A DAC

Pulse Width Modulation (PWM) technique can be used easily to create a DAC, especially since many of the members of the AVR processor family are equipped with on-chip PWM.

In PWM, a digital signal of fixed frequency is generated. The pulse width of the signal is changed according to the requirement. Ideally, it should be possible to vary the width to any arbitrary value. However, with a counter-based PWM, the change can be only as much as the resolution of the counter. A PWM implemented using an 8-bit counter can only change the pulse width by 0.4% approximately (1 bit in 255). By employing a low-pass filter at the output of a PWM wave, the average value of the signal is extracted. The average value of a digital signal is equal to the duty cycle of the waveform.

Figure 6.22 illustrates a 2-bit PWM signal. Figure 6.23 illustrates a DAC using the built-in PWM generator on output PORTB3 pin and an external RC filter. The RC filter has a bandwidth of about 16 Hz for the values illustrated. The Timer1 can be clocked at the system clock frequency of 4 MHz as illustrated, and for an 8-bit PWM, the PWM frequency will be about 7800 Hz. The RC filter will cleanly filter out the high-frequency components, and a clean DC value will be produced.

6.8.2 R-2R LADDER DAC

Using only two different values of resistors, it is possible to build a simple R-2R ladder DAC of reasonable linearity. Figure 6.24 illustrates a R-2R ladder DAC connected to the PORTB of the AT90S1200. To use the DAC, the following code can be used.

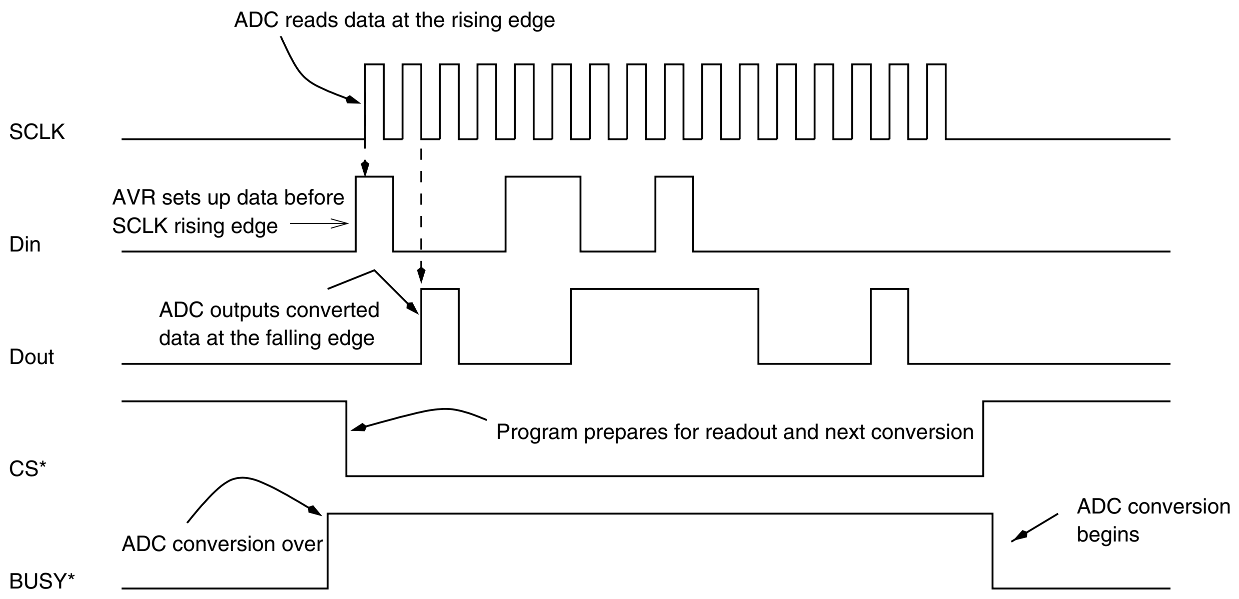

ADC reads data at the rising edge
SCLK
AVR sets up data before SCLK rising edge
Din
ADC outputs converted data at the falling edge
Dout
Program prepares for readout and next conversion
CS*
ADC conversion over
ADC conversion begins
BUSY*

Figure 6.21 **Timing diagram of the conversion and readout process of the MAX111.**

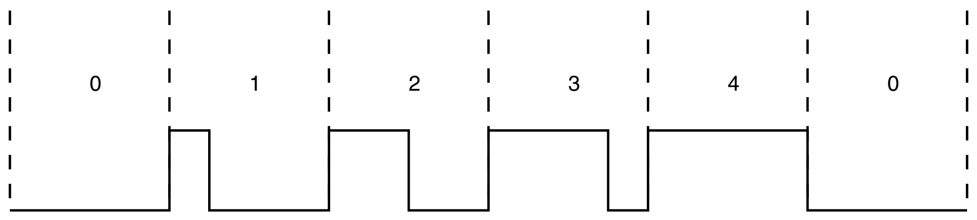


Figure 6.22 A continuously varying PWM signal. The average value of the signal changes by 25% in each period.

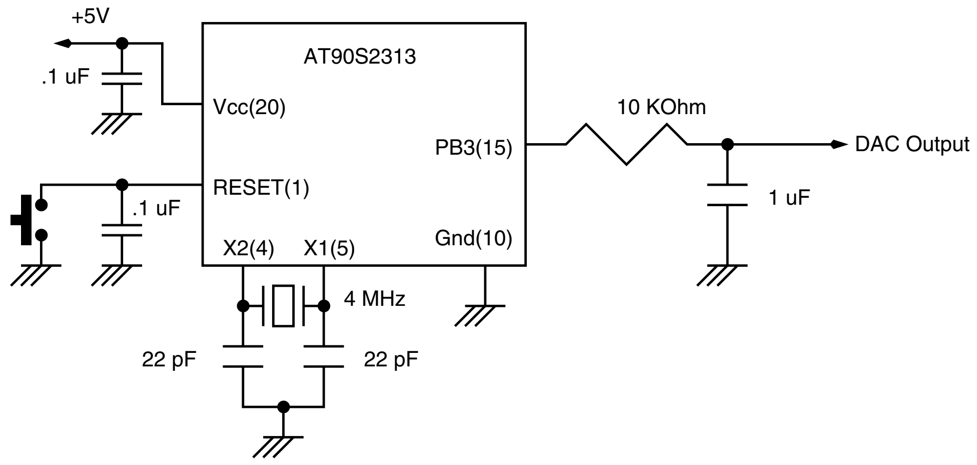


Figure 6.23 PWM DAC using an AT90S2313 and an output RC filter.

```
.include "1200def.inc"
.def DACVALUE=r17                ;Register with the DAC value
.def temp    =r18
init_portb:  ldi temp, $FF       ;Initialize PORTB as output
          out DDRB, temp
load_dac:    out PORTB, DACVALUE  ;output value to the DAC
```

6.8.3 MAX521 DAC

MAX521 is a voltage output DAC and has a simple 2-wire digital interface. These 2 wires can be connected to more MAX521s (total up to 4). The IC operates from a single +5-V supply. Even with a +5-V supply, the outputs of the DACs can swing from 0 to +5 V. The IC has 5 reference voltage inputs that have a range that can be set to anywhere between 0 to +5 V.

Figure 6.25 illustrates the block diagram of the IC. Table 6.2 lists the signals of the MAX521 DAC IC. The MAX521 has five reference inputs. The first four DACs each have independent reference inputs, and the last four share a common reference voltage input.

The digital interface allows the IC to communicate to the host at a maximum of 400 Kbps. The input of the DACs have a dual data buffer. One of the buffer outputs drives the DACs while the other can be loaded with a new input. All the DACs can be set to a new value independently or simultaneously. The IC can also be programmed to a low-power mode, during which time the supply current is reduced to 4 μA only. The power-on reset circuit inside the IC sets all of the DAC outputs to 0 V when power is initially applied.

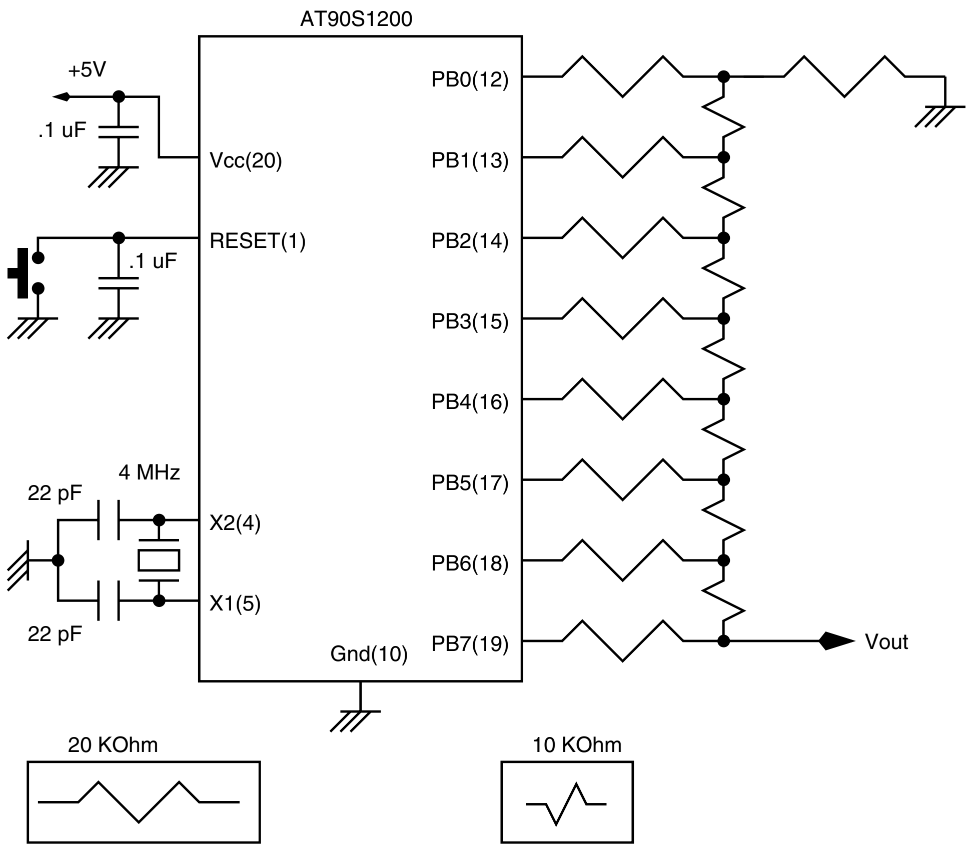


Figure 6.24 **R-2R ladder DAC implementation with an AVR controller.**

The output of an 8-bit DAC is

$$\text{Vout} = \text{Vref (input/256)}$$

where input is an 8-bit number and Vref is the reference voltage for the channel.

6.8.4 DATA TRANSFER TO A MAX521

The MAX521 uses a simple two-wire interface. Up to four MAX521s can be connected to one set of these two-wire interfaces. This means that a host system with two output lines can be used to program up to 32 DACs!

To send commands and data to MAX521, the host sends logic sequences on the SDA and SCL lines. Otherwise, these lines are held to "1". The two-wire interface of MAX521 is compatible with the I2C interface. To maintain compatibility with I2C, external pull-up resistors on the SDA and SCL lines would be required. Otherwise, these resistors are not required.

MAX521 is a receive-only device, so it cannot transmit any data. The host only needs two output signal lines for SDA and SCL signals. The SCL clock frequency is limited to 400 kHz. The host starts communication by first sending the address of the device followed by the rest of the information, which could be a command byte or a command byte

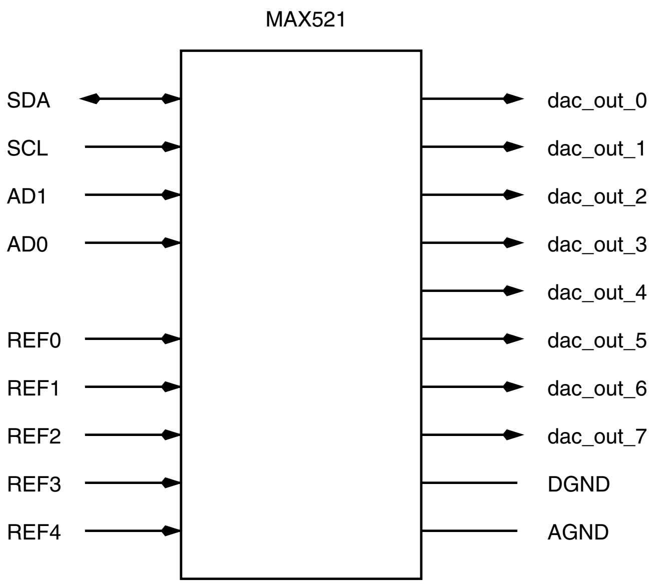


Figure 6.25 **Block diagram of MAX521 DAC.**

and data byte pair. Each such transmission begins with a START condition as illustrated in the timing diagram in Figure 6.26, followed by the device address (called the slave address) and command-byte, data-byte pairs or command byte alone. The end of transmission is signaled by the STOP condition on the SDA and SCL lines.

The SDA signal is allowed to change only when the SCL signal is low, except during the START and STOP conditions. For the START condition, the SDA signal makes a high to low transition while the SCL signal is high. Data to the MAX521 is transmitted in 8-bit packets (which could be the address byte, the command byte, or the data byte) and it needs nine clock pulses on the SCL signal line. During the ninth SCL pulse, the SDA line is held low, as illustrated in the timing diagram. The STOP condition is signaled by a low to high transition on the SDA signal line when the SCL signal is held high.

The address and command bytes transfer important information to MAX521. The address byte is needed to select one out of a maximum of four devices that could be connected to the SDA-SCL signal lines. After the host starts the communication with the START condition, all the slave devices on the bus (here the bus is referred to the SDA and SCL signal lines) start listening. The first information byte is the address byte. The slave devices compare the address bits AD0 and AD1 with the AD0 and AD1 pins condition on the IC. In case a match occurs, the subsequent transmission is for that slave device.

The next transmission is either a command byte or a command-byte, data-byte pair. In either case, the data byte, if at all, follows the command byte, as illustrated in Figure 6.27. Table 6.3 lists the bit sequence of the command byte and the function of each bit.

All the possible combinations of address byte, command byte, and data byte to a MAX521 are:

TABLE 6-2 SIGNAL DESCRIPTION OF THE MAX521 DAC

SIGNAL NAME	FUNCTION
OUT0	DAC0 voltage output
OUT1	DAC1 voltage output
OUT2	DAC2 voltage output
OUT3	DAC3 voltage output
OUT4	DAC4 voltage output
OUT5	DAV5 voltage output
OUT6	DAC6 voltage output
OUT7	DAC7 voltage output
REF0	Reference voltage input for DAC0
REF1	Reference voltage input for DAC1
REF2	Reference voltage input for DAC2
REF3	Reference voltage input for DAC3
REF4	Reference voltage input for DACs 4, 5, 6, and 7
SCL	Serial Clock input
SDA	Serial Data input
AD0	Address input 0. Sets IC's slave address
AD1	Address input 1. Sets IC's slave address
Vdd	Power supply, +5 volts
DGNC	Digital ground
AGND	Analog ground

1. START condition, slave address byte, command byte/output data byte pair, and a STOP condition, or
2. START condition, slave address byte, command byte, STOP condition, or
3. START condition, slave address byte, multiple command byte/output data byte pairs, STOP condition.

Figure 6.28 illustrates how to connect up to 4 MAX521s on a single bus from the host. The four devices are distinguished by the different addresses set on the AD0 and AD1 lines. Each of the MAX521 compares these bits with the address bits in the address byte transmission from the host.

Figure 6.29 illustrates how an AT90S2313 AVR processor can be connected to a MAX521 DAC to provide up to 8 channels of 8-bit DAC. The I2C protocol that the MAX521 DAC understands can be created on any I/O line of the processor, and the processor can communicate to the MAX521 chip under software control.

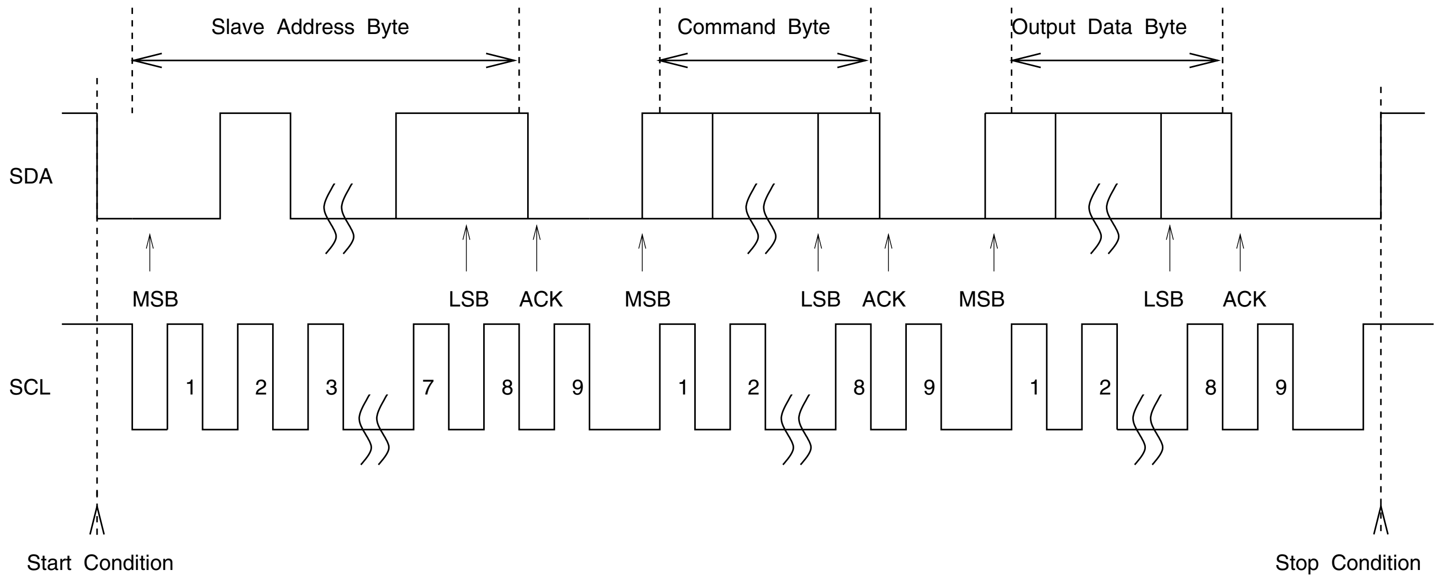


Figure 6.26 **Communication format for MAX521 serial DAC. All transmission begins with a START condition and ends with a STOP condition.**

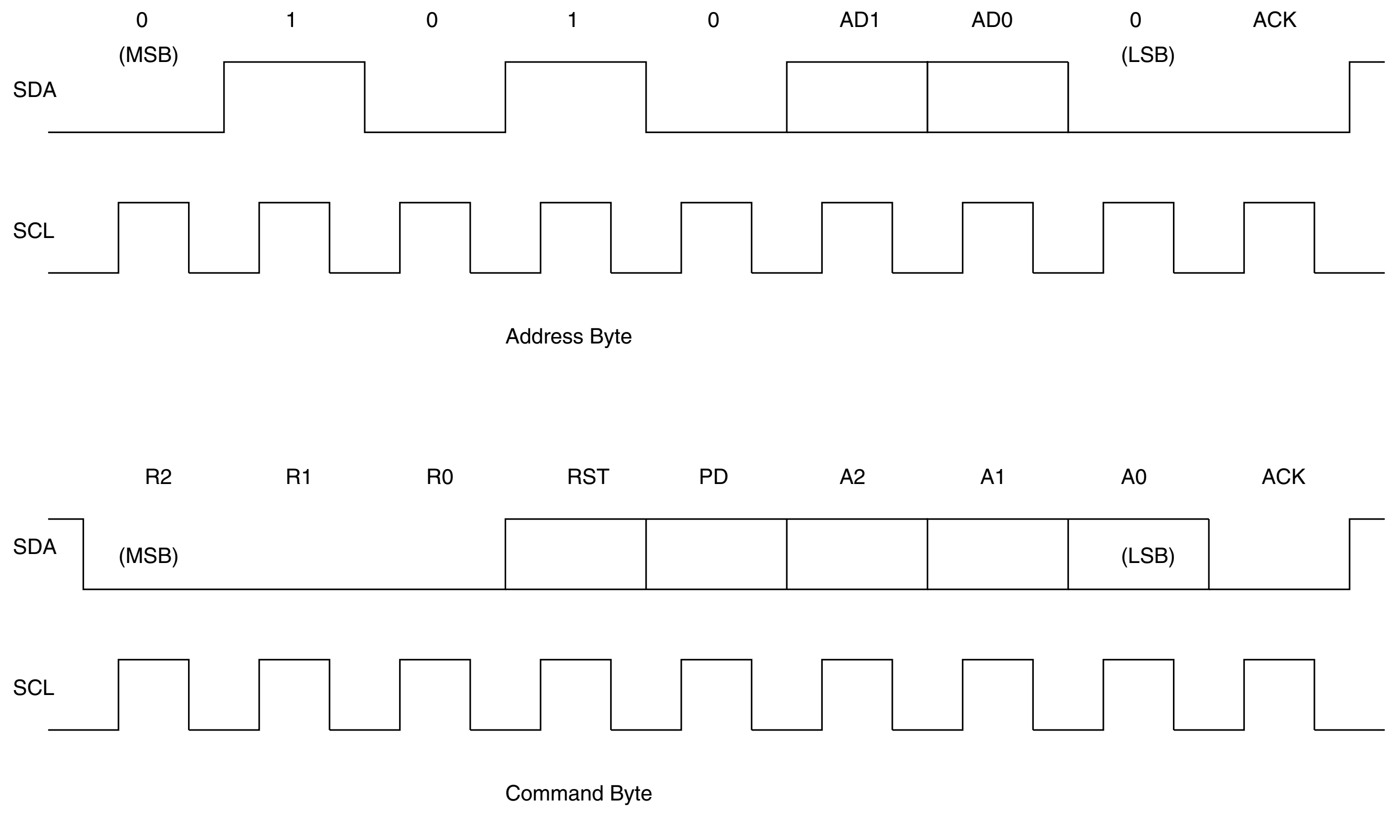


Figure 6.27 **Structure of the Address and Command bytes.**

TABLE 6-3 BITS OF THE COMMAND BYTE FOR MAX521

BIT NAME	FUNCTION
R2, R1, R0	Reserved bits. Set to “0.”
RST	RESET bit. A “1” on this bit resets all DAC registers.
PD	Power Down bit. A “1” on this bit put MAX521 in a power down mode. A “0” returns the MAX521 to normal state.
A2, A1, A0	Address bits. Defines address of the DAC to which the subsequent data byte will be addressed.
ACK	Acknowledgment bit. Set to “0.”

6.9 Interfacing LED Displays

Displays are an important component in an embedded system. They are one of the most popular ways to communicate with the system user. There are many types of display devices that can be used and interfaced with the AVR processor.

6.9.1 SEVEN-SEGMENT DISPLAYS

The simplest display device is, of course, a LED and we have already seen how it can be connected and used with the AVR. But it can provide limited information to the user. A LED seven-segment display, on the other hand, can be used to provide numeric information. It requires eight signal lines if possible and at least seven at the minimum. The display has seven LEDs labeled “a” through “g” and then there is a decimal point. Figure 6.30 illustrates a scheme to connect two LED seven-segment displays to the AVR processor.

This puts an immense resource load on the processor. Interfacing a couple of seven-segment displays takes up all the I/O pins. The situation is remedied by using a multiplex scheme. Here, at the cost of increased software complexity, some of the I/O pins can be saved while at the same time more displays can be added. This scheme has the advantage that our eye cannot follow any light change faster than about 20 Hz. So if an LED display is put on and off at a rate greater than 20 Hz, due to the persistence property of the eye, it will not feel any difference, provided that the average intensity of the LED is maintained. Thus, many LED displays can share the same I/O lines, with only one of them being lit at any time.

Figure 6.31 illustrates how four LED displays can connect to the AVR controller using the multiplex scheme. The power to each display (common anode type display) is controlled by an output signal line of the AVR through a PNP transistor switch. A “0” at the base turns it ON and provides voltage to the display. At any given time, only one of the transistors is turned ON. Once a +ve voltage is applied to a display, the cathodes being connected to port lines, the pattern of “1”s and “0”s on the port will determine which LED segment glows. A “0” on the port will sink the current from the segment and the segment will glow.

The seven-segment LED displays cannot be used to display alphabets (well, only a limited number of alphabets can be displayed). To display aphanumeric information, there are alphanumeric displays available which have sixteen segments as illustrated in Figure 6.32.

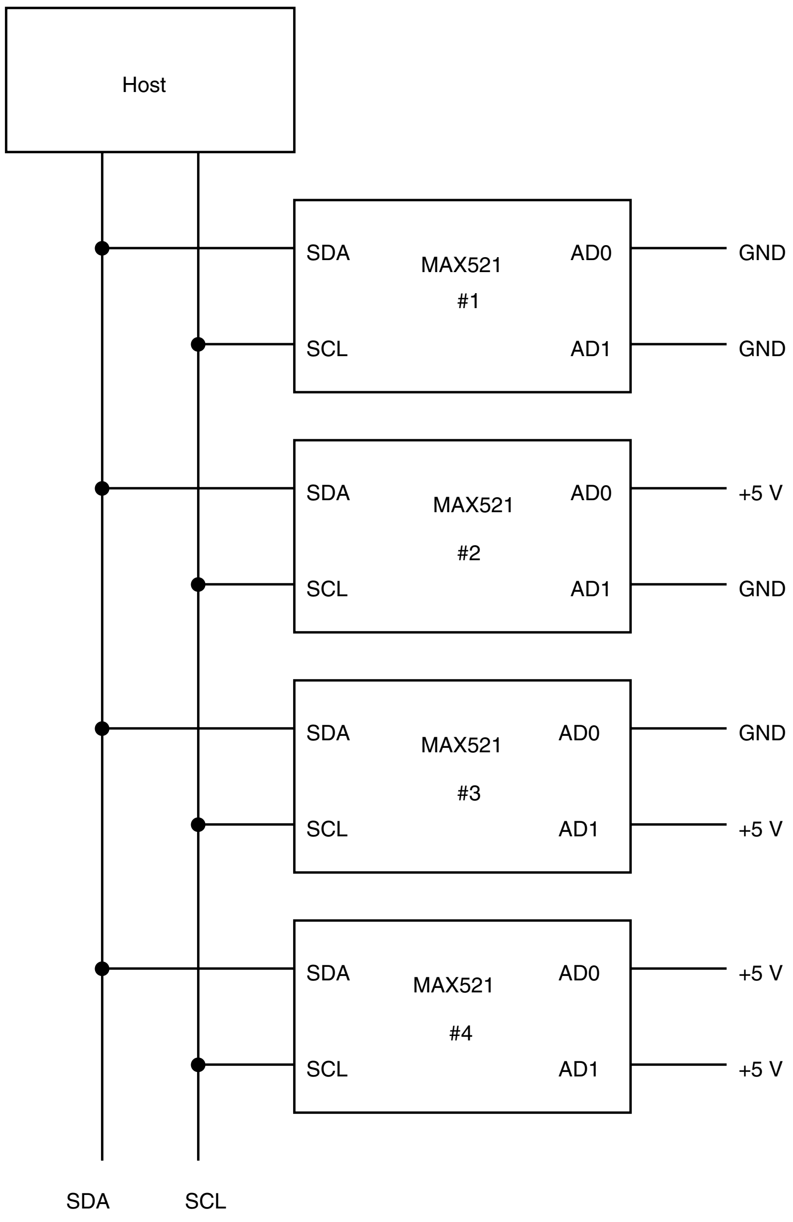


Figure 6.28 **Connecting multiple MAX521s on a single bus.**

6.9.2 DOT MATRIX DISPLAYS

Dot matrix displays are the best in terms of the type of information that can be displayed, including graphics. Dot matrix displays are necessarily multiplexed displays. Figure 6.33 illustrates an interface circuit for a 5-by-7 dot matrix display. Figure 6.34 is a circuit diagram for a AT90S2313 controlled dot matrix display. The display is arranged as five columns of seven LEDs each. Each column is refreshed at a rate of 40 Hz at 4.00-MHz

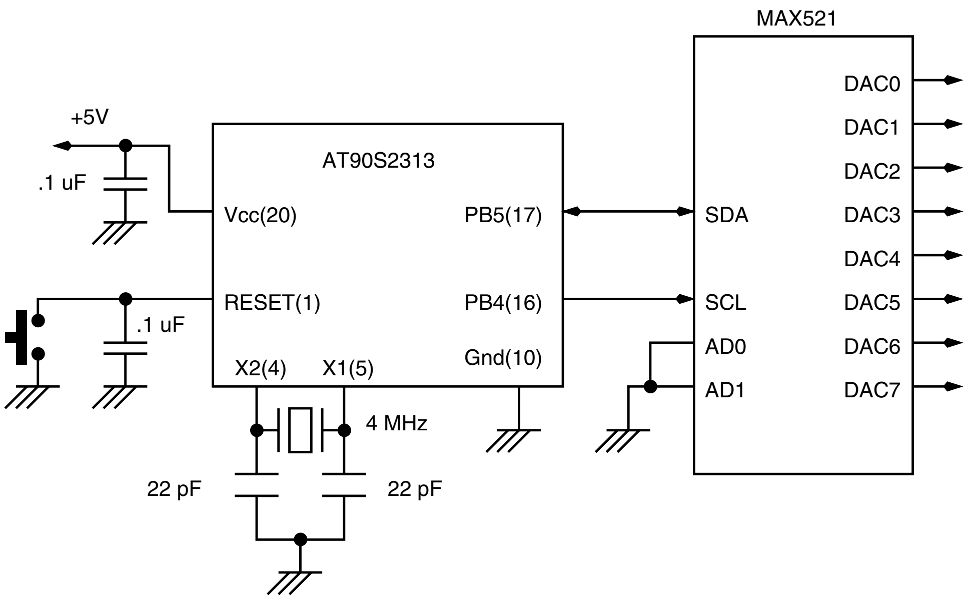


Figure 6.29 **Connecting AT90S2313 AVR processor to MAX521 DAC.**

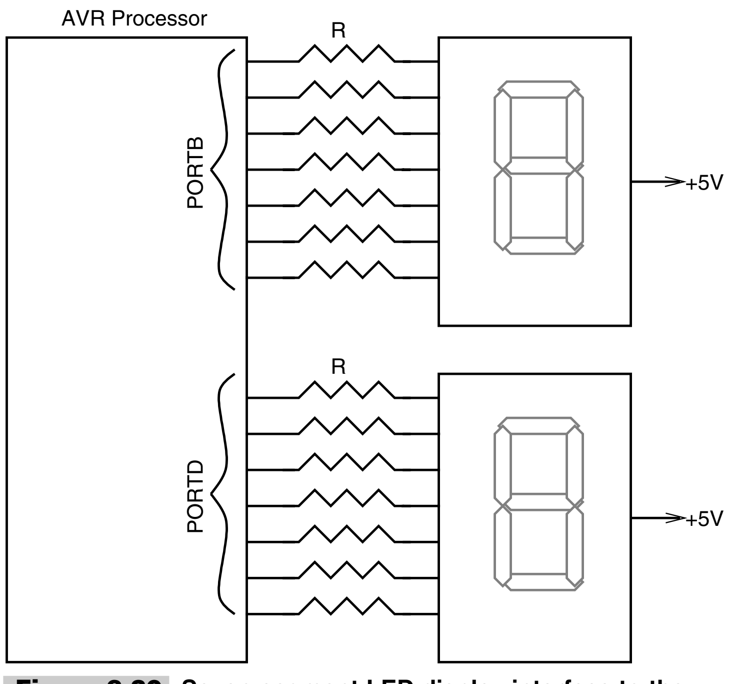


Figure 6.30 **Seven-segment LED display interface to the AVR processor.**

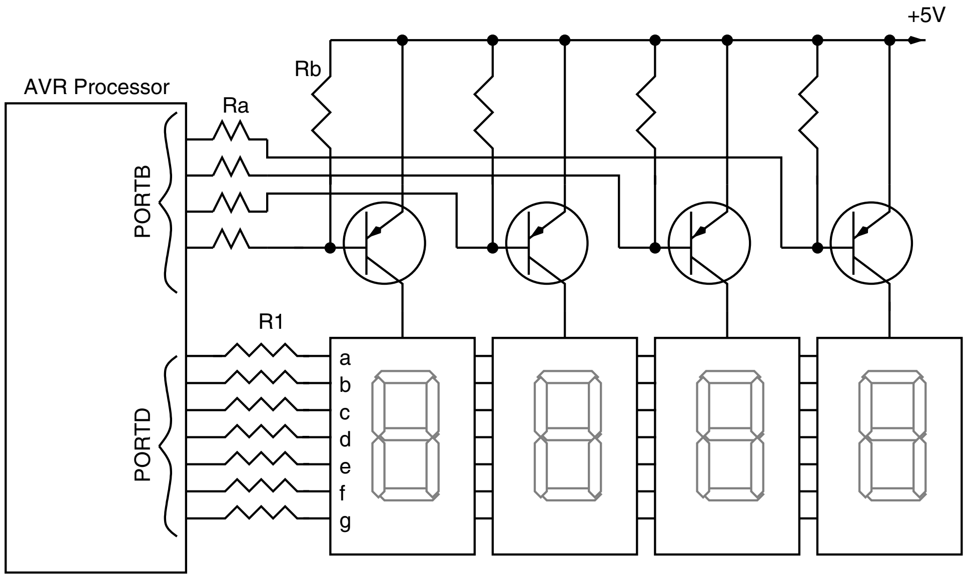


Figure 6.31 A multiplexed seven-segment LED display interface to the AVR processor.

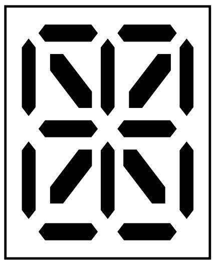


Figure 6.32 An alphanumeric LED display.

clock frequency. The actual clock frequency used for the circuit is 3.58 MHz and so the refresh rate is about 36 Hz.

This is an illustrative circuit. The code just waits for a key to be pressed, and at each key press it displays a new number or a new alphabet in a sequence.

Since there are five columns of LEDs, the duty cycle of current flowing in each column is 20%, and so to maintain the same average current (of 4 mA) the peak current is increased five times to 20 mA. The value of the current-limiting resistor is hence chosen to be 150 ohms. Figure 6.35 illustrates the test board for the 5 × 7 dot matrix display. Code for this circuit is available in the code directory in the file 5x7disp.asm.

6.10 Interfacing LCD Displays

LCD displays are very useful for displaying user information and communication. LCD displays are available in various formats. Most common are 1x16, i.e., 1 line with 16 alphanumeric characters. Other formats are 2x16, 1x40, 2x40, 4x16, etc.

The LCD displays have the following format:

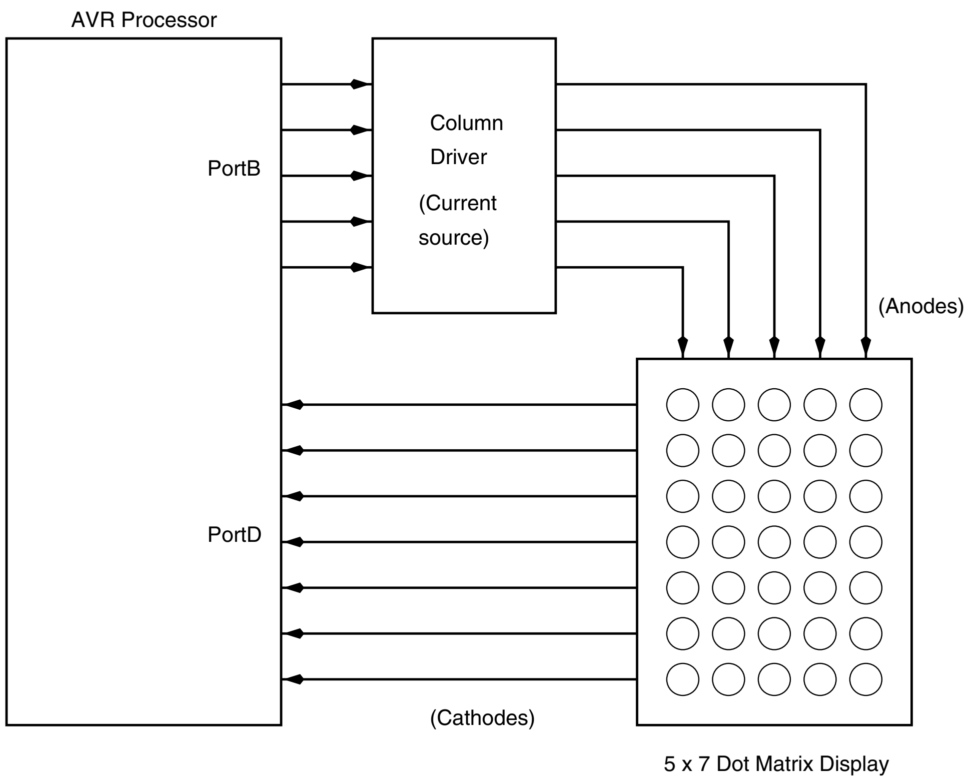


Figure 6.33 **Block diagram for a 5-x-7 dot matrix display to AVR interface.**

PIN	SYMBOL	I/O	FUNCTION
1	Vss	-	Power supply Gnd
2	Vcc	-	Power supply +5V
3	Vdd	-	Contrast adjust
4	RS	I	0 = Instruction input
			1 = Data input
5	R/W	I	0 = Write to LCD
			1 = Read from LCD
6	E	I	Enable signal
7	DB0	I/O	Data bit line 0 (LSB)
8	DB1	I/O	Data bit line 1
9	DB2	I/O	Data bit line 2
10	DB3	I/O	Data bit line 3
11	DB4	I/O	Data bit line 4

12	DB5	I/O	Data bit line 5
13	DB6	I/O	Data bit line 6
14	DB7	I/O	Data bit line 7 (MSB)

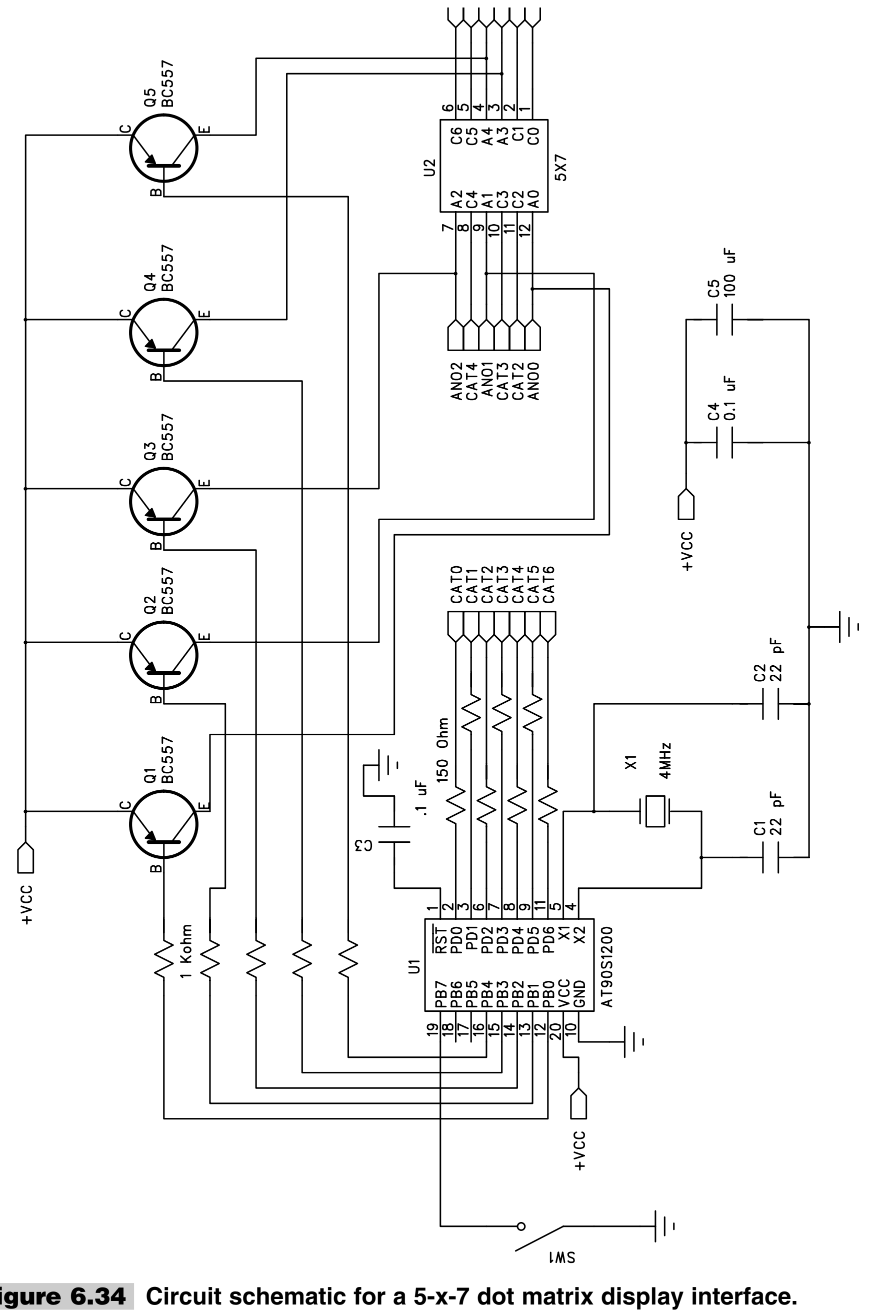


Figure 6.34 Circuit schematic for a 5-x-7 dot matrix display interface.

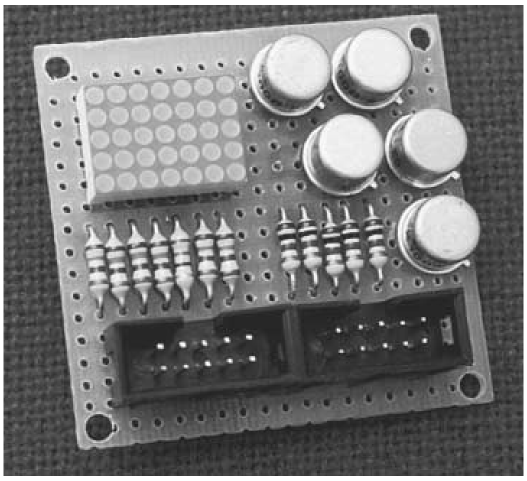


Figure 6.35 A 5-x-7 dot matrix display test board photograph.

The LCD modules have an 8-bit interface. Besides the 8-bit data bus, the interface has a few other control lines. The default data transfer between the LCD module and an external device is 8 bits, however it is possible to communicate with the LCD module using only four of the eight data lines. Figure 6.36 illustrates the character codes for the LCD, and Figure 6.37 shows how to interface a 2-x-16 line LCD module to an AT90S2313 processor. The R/W line is connected to ground and hence the processor cannot read any status information from the LCD module, but can only write data to the LCD. The source code for the LCD interface example is available on the CD in the code directory in the file my_lcd.asm.

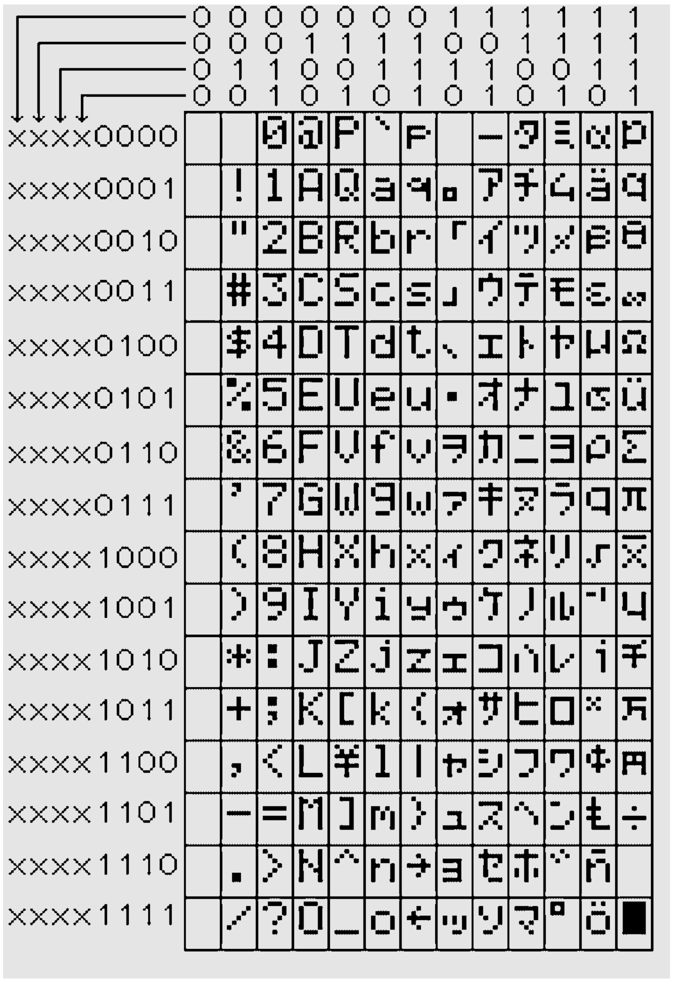
6.11 Driving Relays with AVR

The ULN2003A are high-voltage, high-current darlington arrays containing seven open collector darlington pairs with common emitters. Each of the seven channels can handle 500 mA of sustained current with peaks of 600 mA. Each of the channels has a suppression diode that can be used while driving inductive loads (such as relays) as freewheeling diodes.

The ULN2003A input is TTL compatible. Typical uses of these drivers include driving solenoids, relays, DC motors, LED displays, thermal print heads, etc.

The IC is available in a 16-pin DIP package and other packages. The outputs of the drivers can also be paralleled for higher currents, though this may require a suitable load-sharing mechanism.

Figure 6.38 shows the block diagram of the ULN2003A darlington array driver IC. For each of the drivers, there is a diode with the anode connected to the output and the cathode connected to a common point for all the seven diodes. The outputs are open-collector, which means that external load is connected between the power supply and the output of the driver. The power supply can be any positive voltage less than +50 V as specified by the data sheets. The load value should be such that it needs sustained currents less than 500 mA and peak currents less than 600 mA per driver.



	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
xxxx0000			0	@	P	`	p		ー	タ	ミ	α	p
xxxx0001		!	1	A	Q	a	q	。	ア	チ	ム	ä	q
xxxx0010		"	2	B	R	b	r	「	イ	ツ	メ	β	θ
xxxx0011		#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
xxxx0100		$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
xxxx0101		%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
xxxx0110		&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111		'	7	G	W	g	w	ァ	キ	ヌ	ラ	g	π
xxxx1000		(	8	H	X	h	x	ィ	ク	ネ	リ	√	x̄
xxxx1001		)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010		*	:	J	Z	j	z	ェ	コ	ハ	レ	j	千
xxxx1011		+	;	K	[	k	{	ォ	サ	ヒ	ロ	ˣ	万
xxxx1100		,	<	L	¥	l	|	ャ	シ	フ	ワ	¢	円
xxxx1101		−	=	M	]	m	}	ュ	ス	ヘ	ン	£	÷
xxxx1110		.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
xxxx1111		/	?	O	_	o	←	ッ	ソ	マ	゜	ö	█

LCD Character Codes

Figure 6.36 LCD character codes.

The following diagram shows how these drivers are used to drive relay coils. Figure 6.39 shows three relays being driven by the outputs of three drivers from the ULN2003A IC. One end of the relay coil is connected to the output of the driver and the other end is connected to the +ve supply voltage. The value of this voltage will depend upon the relay coil voltage ratings. The diode common point is also connected to the +ve supply voltage. The inputs to the ULN2003A IC is TTL voltages, say the output of the port pins of the AVR, for example. With this arrangement, the port signals could be used to control each of the relays.

The relay terminals labeled NC (Normally Closed), common, and NO (Normally Open) could be used to switch whatever voltage that may need to be switched. Typically, the relay terminals are used to switch the main supply (220 V AC or 115 V AC as may be the case) to the required load (a heater or a lamp, etc.), but, of course, it may be used to switch any voltage (AC or DC) as long as the relay contact can handle the voltage and the current.

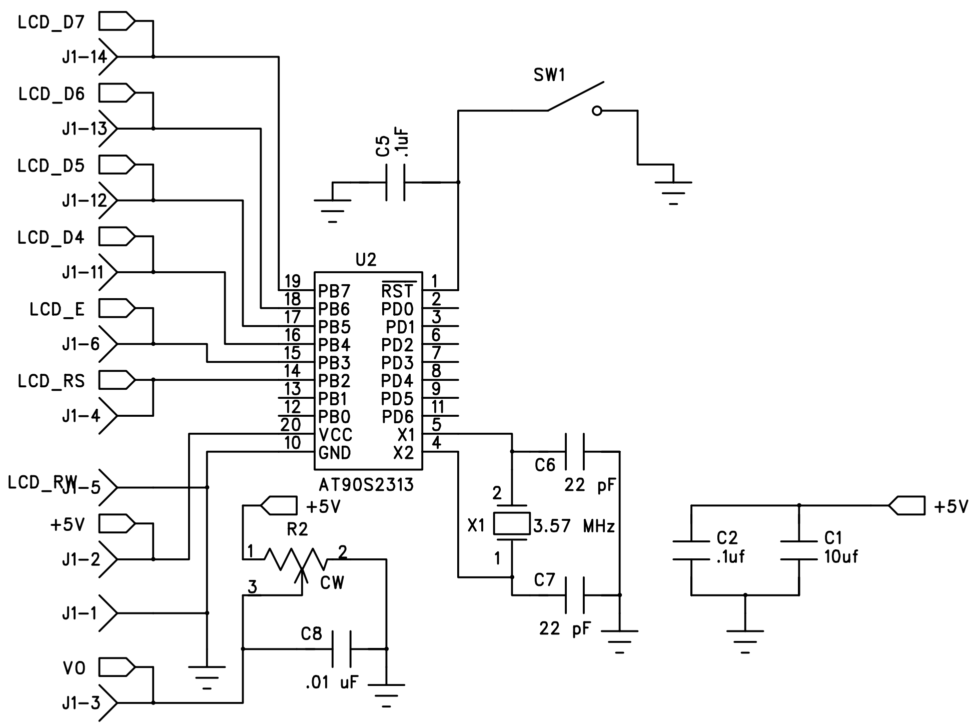


Figure 6.37 **Circuit schematic for an AT90S2313 processor interface to a 2-line, 16-character LCD.**

6.12 Stepper Motor Interface for the AVR

Figure 6.40 illustrates a very popular stepper motor sequencer and driver interface to the AVR processor. The L297/298 sequencer and driver is made by SGS Thomson (us.st.com).

L297 is a stepper motor controller IC that generates four-phase drive signals for two-phase bipolar and four-phase unipolar step motors in microcontroller-controlled applications. The motor can be driven in half step, normal, and other modes, and on-chip PWM chopper circuits permit switch-mode control of the current in the windings. The IC only requires a mode input, a clock input, and a direction input for its operation. This greatly reduces the software burden of the microcontroller.

To drive the stepper or DC motors, a matching driver IC such as the L298 is used. L298 is a dual full-bridge driver. It can be used with power supply voltages up to 48 V and total DC current up to 4 A.

For a larger drive, L2603 from SGS Thomson can be used instead of the L298. Figure 6.41 illustrates the circuit schematic of L297 and L298, which can be used with an AVR processor.

When moving motors, it is always advisable to gently increase the speed of the motors rather than operate the motor at a fixed speed right at the beginning. The motor is started slowly at a minimum speed and then gradually, the speed is increased till it reaches the

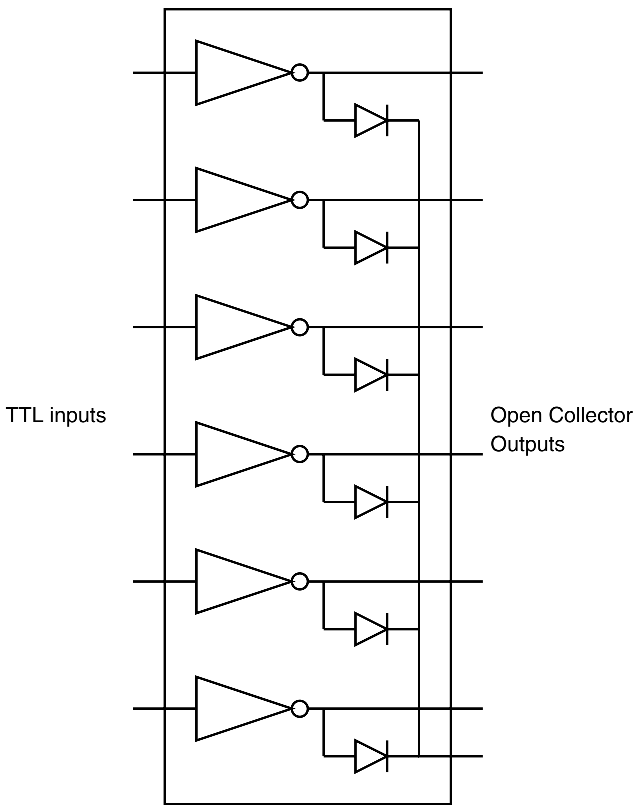


Figure 6.38 **ULN2003A darlington array.**

maximum operable speed. To bring the motor to a halt, the speed is gradually decreased before stopping. Figure 6.42 illustrates the motor speed ramping.

6.13 Interfacing to a Serial EEPROM

Serial EEPROMs are getting very popular for a variety of reasons. You can store up to 64 Kbytes of data in small 8-pin DIP package. The communication takes only two signals, since most serial EEPROMs are available with IIC interface. These EEPROMs can be written 100,000 times or even more. Even though the data write takes 10 ms, by writing an entire page at a time, the average write rate can be improved. Writing a page of data into the EEPROM also takes almost the same time as writing a byte. The page size can vary between 16 bytes for smaller-capacity EEPROMs to 128 bytes for the larger 64-Kbyte capacity EEPROMs. Thus data transfer speeds can be improved by writing in a burst mode.

Figure 6.47 (appearing later in this chapter) illustrates the circuit schematic for the AVR to EEPROM interface. A MAX232 RS-232 line translator has been connected so that the user can read and write data to the EEPROM.

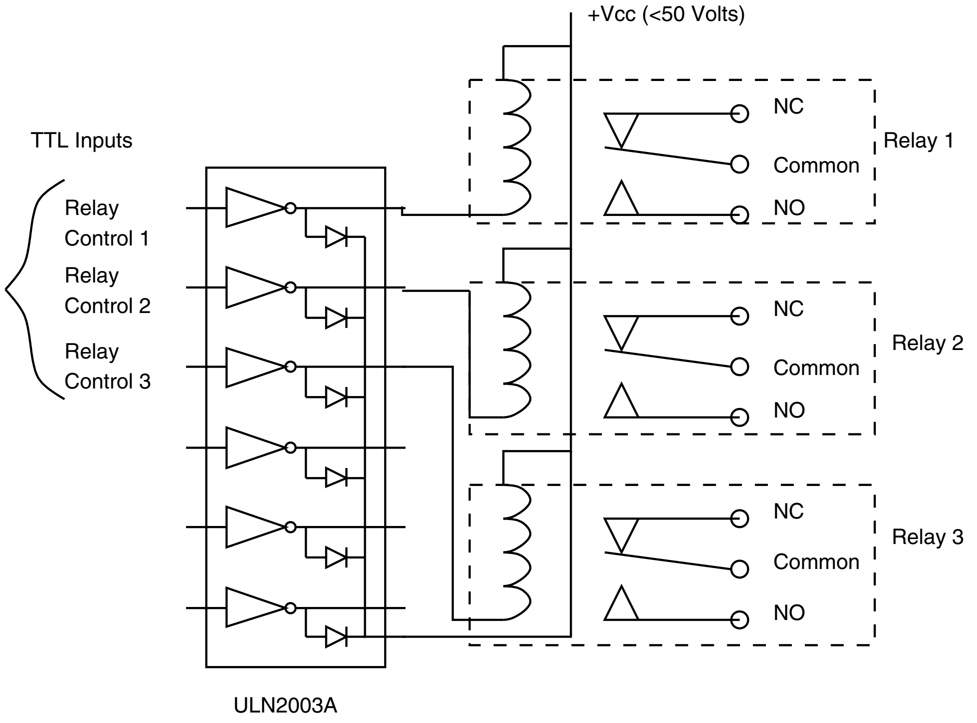


Figure 6.39 ULN2003A drivers used to drive inductive loads.

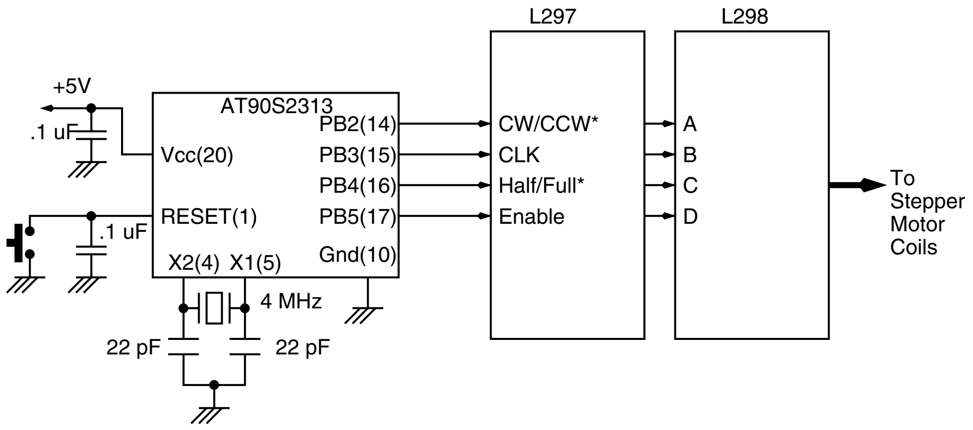


Figure 6.40 A stepper motor sequencer and driver interface to AVR.

The EEPROM has a write protect pin (WP) that can be connected to +5 V to disable any write to the EEPROM. For our use, we have connected it to ground so that we can write data to the EEPROM.

The EEPROM we have chosen is AT24C512 from Atmel. It has a capacity of 64 Kbytes. The EEPROM has two device address lines that allow up to four such EEPROM chips to be connected to the same IIC bus (Figure 6.43).

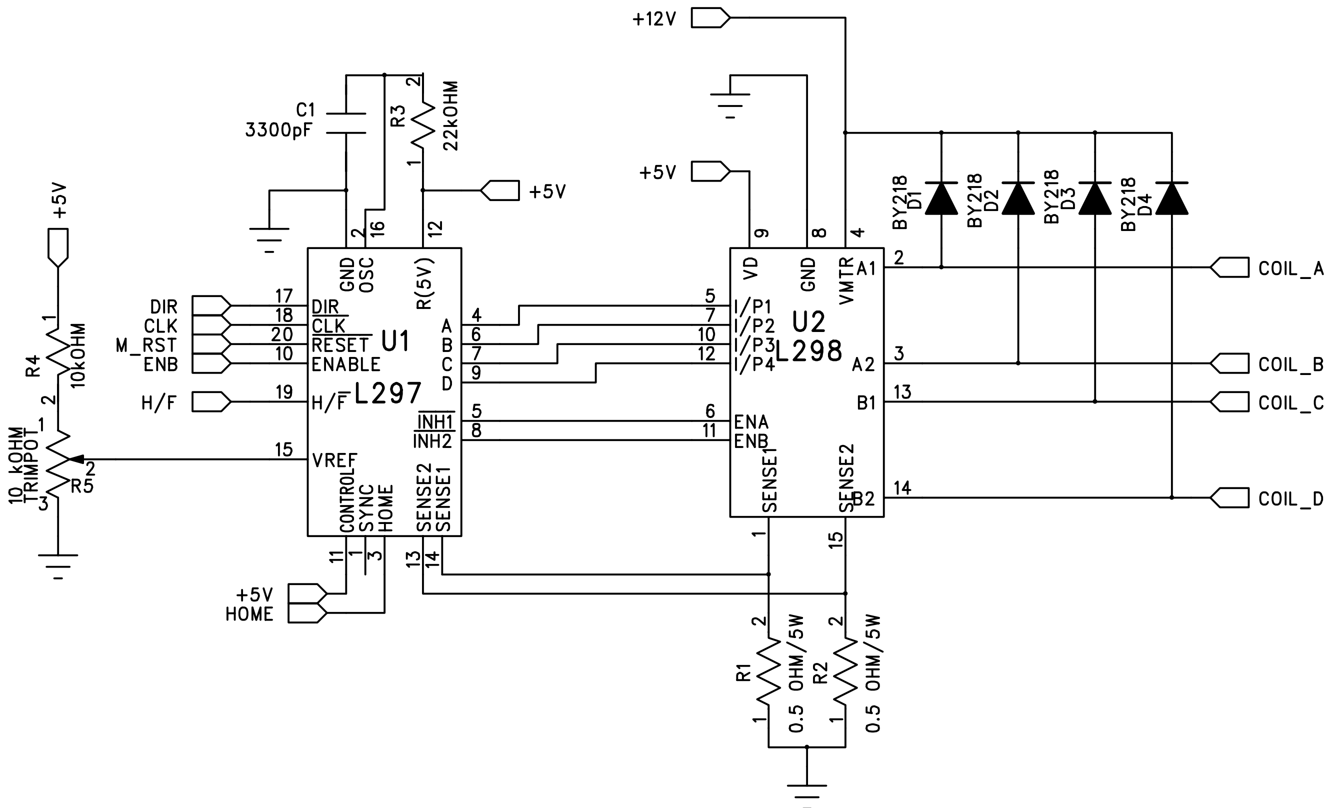


Figure 6.41 Circuit schematic for a stepper motor sequencer and driver for the AVR processors.

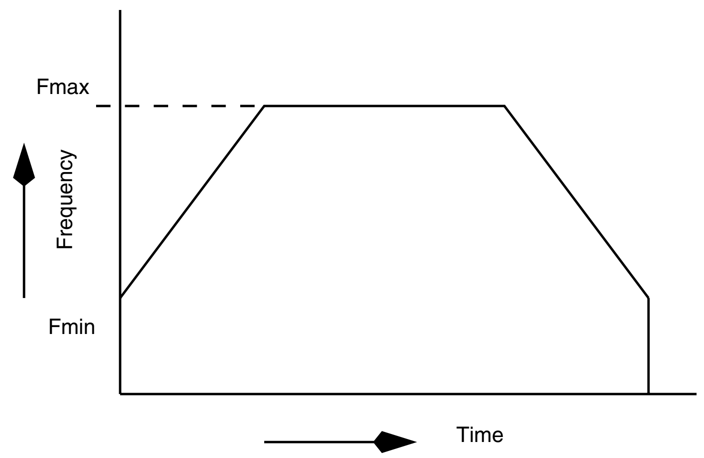


Figure 6.42 Ramping the stepper motor speed.

7	6	5	4	3	2	1	0
1	0	1	0	0	A1	A0	R/W*

Figure 6.43 EEPROM device address.

The EEPROM can be read and written in the following ways:

1. Byte Write: Figure 6.44 illustrates a byte write to the EEPROM. Following the Start sequence, the device address is transmitted with the R/W bit reset to “0”, followed by the address of the location to which the byte is to be written to. Since the capacity of the EEPROM is 64 Kbytes, a 16-bit address split as MSB address and LSB address is sent next in that order. In the end, the data byte to be written is sent. The sequence is terminated with a Stop sequence.
2. Page Write: Same sequence as the byte write, except that multiple data bytes that are to be written are transmitted before the Stop sequence is issued.
3. Current Address Read: The Start sequence is issued followed by the device address (R/W bit is set to “1”) and the data byte from the EEPROM is received. Figure 6.45 illustrates the transfer.
4. Random and Sequential Read: This requires a dummy write sequence to precede the actual read. The purpose is to provide the address from where to read the data. The Start sequence is issued followed by a device address that includes the R/W bit reset to “0” to indicate write. Then follows the MSB address and LSB address, and after that a new Start sequence is issued, followed by the device address sequence again with the R/W bit set to “1” to indicate the read operation. After this, the device provides the data from the required location and terminated with a Stop sequence. Since the initial write sequence is not terminated with a Stop sequence, the write to the location is not performed; instead, only the address gets changed. Figure 6.46 illustrates the data transfer

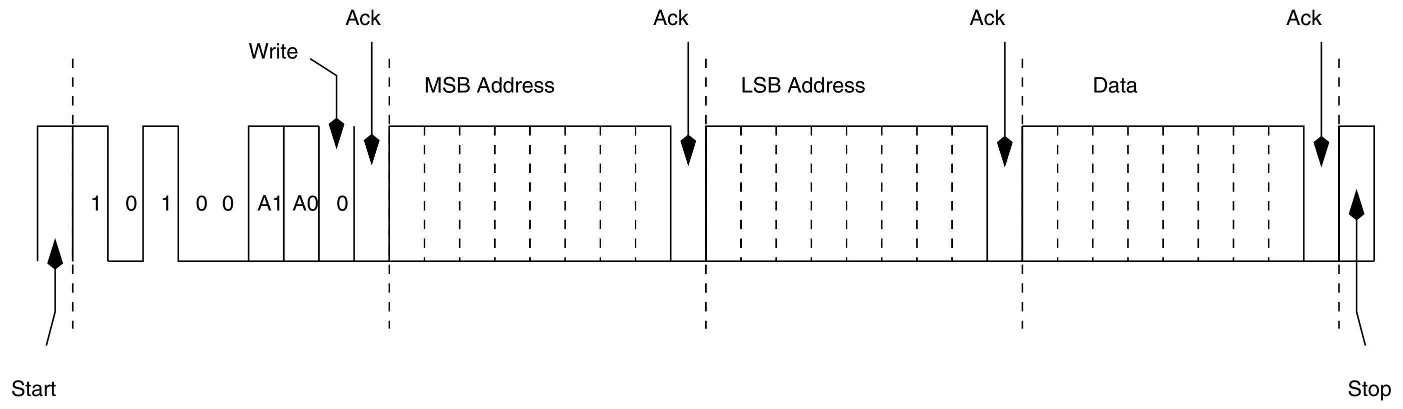


Figure 6.44 **EEPROM write byte.**

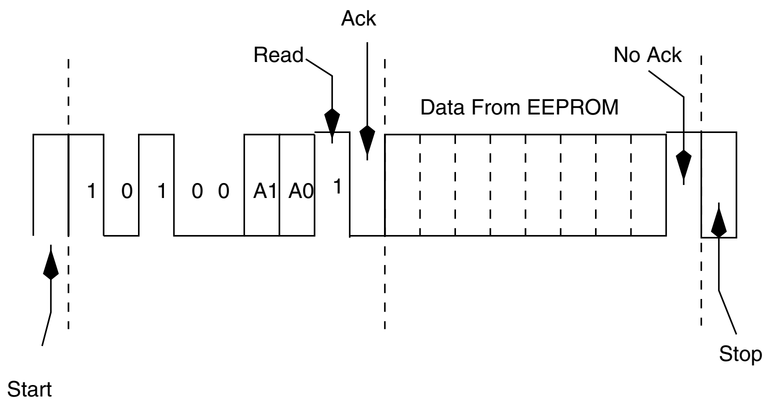


Figure 6.45 **EEPROM current address read.**

sequence. If an Ack is generated by the processor before the Stop sequence, then data byte from the next location is received, leading to sequential read operation. The last data read must be terminated with a No Ack before the Stop sequence.

The sample program to work with the circuit schematic in Figure 6.47 is available in the code directory in the file ep1byt.asm, which can read and write one byte at a time at the address you specify, and ep2byt.asm, which can read and write two bytes at a time at the specified address.

6.14 Interfacing to a Real Time Clock (RTC)

RTCs are useful devices as timekeepers in embedded systems. Many serial communication format RTCs in 8-pin DIP package with a host of features are available. We will interface DS1302 RTC from Dallas to AVR processors.

This RTC can trickle charge an external standby NiCd battery. It contains 31 bytes of SRAM. The RTC has a simple three-wire interface to a microprocessor. The RTC provides seconds, minutes, hours, day, date, month, and year information. The RTC can operate in a 12-hour format with an AM/PM indicator or a 24-hour format.

The RTC has a synchronous serial communication interface. Only three wires are required to communicate with a processor such as the AVR.

Figure 6.48 illustrates a block diagram on an AVR interface to the DS1302 RTC.

Figure 6.49 illustrates the circuit schematic. MAX232 has been added so that the user can communicate with the processor and read and write to the RTC. The code for this project is available in the code directory in the file rtc_ex.asm.

The data sheet is available here:

```
http://www.dalsemi.com/DocControl/PDFs/1302.pdf
```

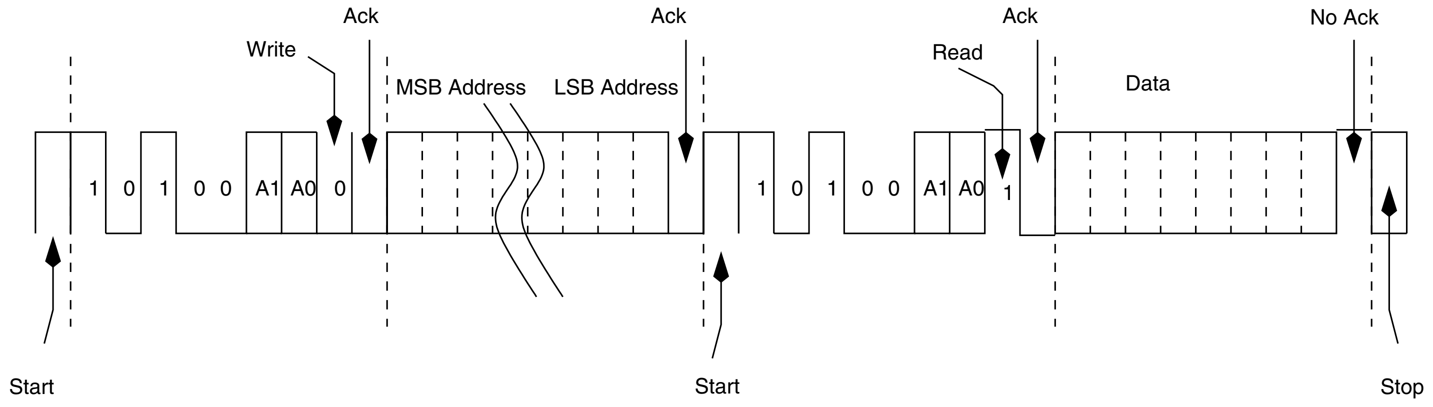


Figure 6.46 **EEPROM random read.**

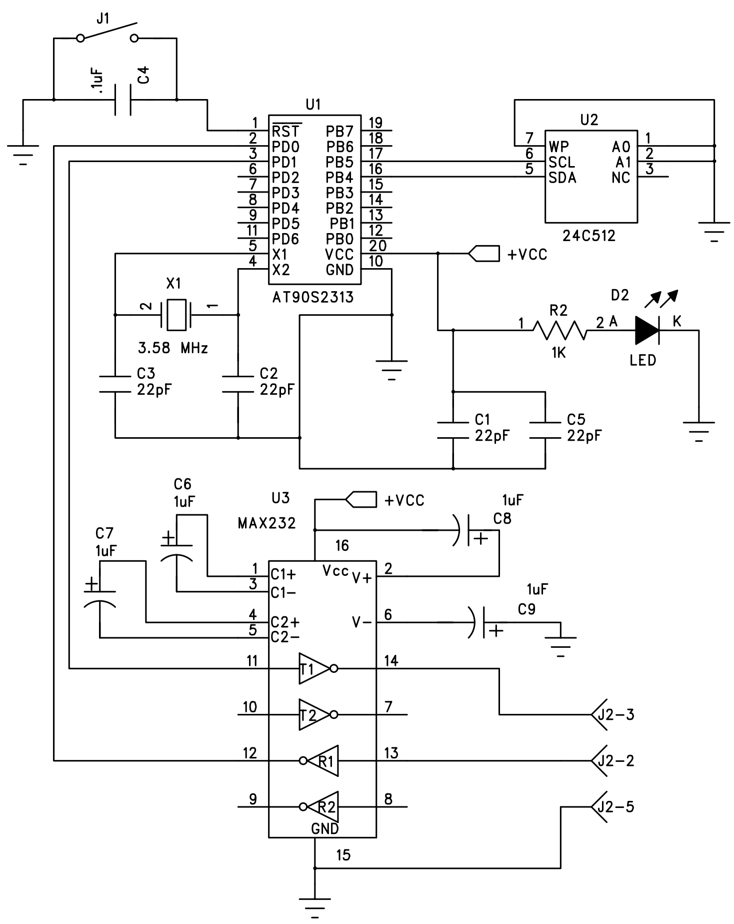


Figure 6.47 Circuit schematic for an AT90S2313 processor interface to a serial EEPROM.

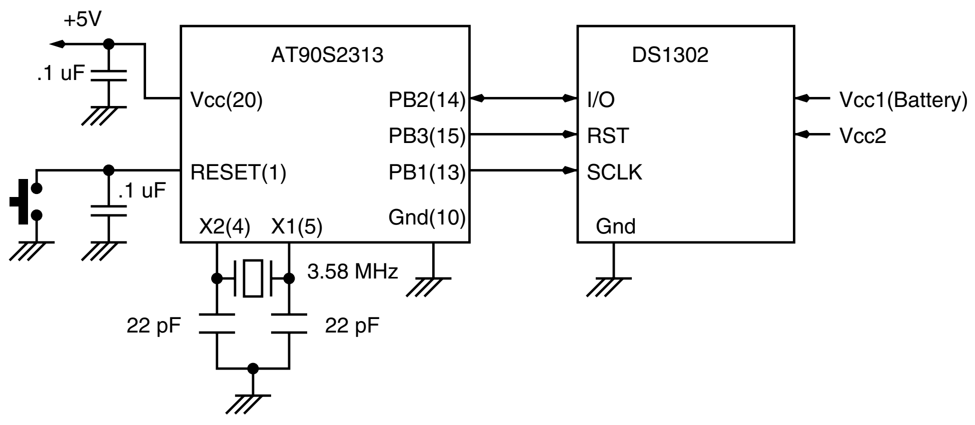


Figure 6.48 RTC interface to an AT90S2313.

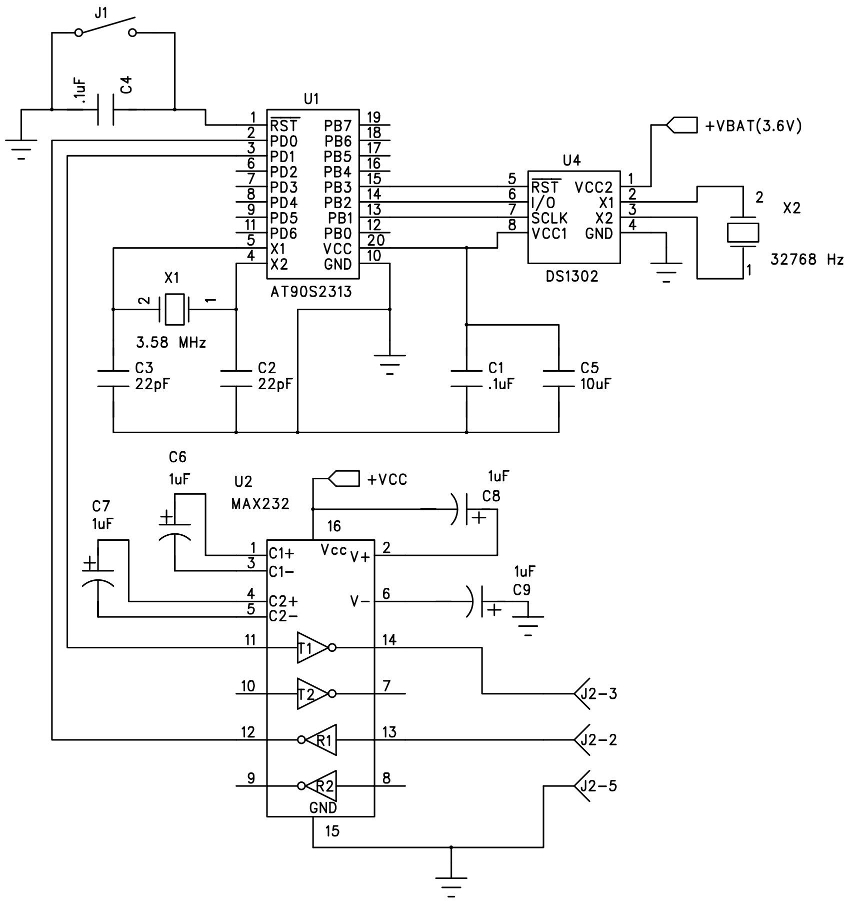


Figure 6.49 Circuit schematic for an AT90S2313 processor interface to an RTC.

6.15 Accessing a Constants Table

Most AVR processors have on-chip flash program memory as well as some amount of EEPROM. Both these memories can be used to store constants. If the constants are stored in the EEPROM, they can even be modified, while the constants stored in the flash program memory cannot be changed expect while programming the flash program memory of the chip.

The constants are stored in the program memory either at a predefined address (using the appropriate origin assembler statement such as .org for the Atmel's AVR assembler) or by identifying the start of the constants table with a label, as in the example below.

```
msg1: .db "Honk! Honk!     Honk! Honk!    "
```

To access an individual element of the table, say the tenth from the start of the table, the following program is used.

```
        ldi ZH, high(msg1*2)  ;init the pointer register
        ldi ZL, low(msg1*2)
        adiw ZL, 10           ;add an offset to the pointer
more1:  lpm                   ;read program memory.
                              ;data is now available in register R0
```

Similarly, constants and tables can be stored in the EEPROM memory. The assembler program must contain the ".eseg" directive to instruct the assembler to locate the following data in the EEPROM memory map.

```
.eseg
org 0
;Start of the message
morse_msg:
.db 2                         ;C
.db 16                        ;Q
```

Accessing the EEPROM is done as follows:

```
        ldi ZL, low(morse_msg)
eep_notrdy:
        sbic EECR,1           ;skip if EEWE clear
        rjmp eep_notrdy       ;Waits until EEPROM ready
read:
        out EEAR, ZL          ;output address low
        sbi EECR, 0           ;set EERE (Read-strobe)
        nop
        nop
        in R18, EEDR          ;inputs data
```

6.16 Arbitrary Waveform Generation

Generating digital waveforms for various applications is often required, either as a part of a design requirement or as a test pattern generator. Multichannel digital waveform generators are extremely expensive pieces of instruments. Often, you can use a digital circuit to provide a limited functionality of this expensive instrument.

The AVR, with its extremely fast program execution, is quite capable of generating fast, multichannel digital waveforms. An example of what an arbitrary digital waveform might look like is illustrated in Figure 6.50. The required waveform is drawn on a sheet of paper and then encoded as numbers as illustrated in figure wave1. These numbers are then put in a constants table in a program. The waveform generation program outputs the values of the table onto a port which provides the waveform outputs.

Figure 6.51 illustrates one of the waveform patterns being generated by an AT90S8515 processor. The waveform generator code is available on the CD in the code directory in the file wave1.asm.

6.17 A Switch-Case Implementation

The Switch statement is a popular statement used extensively in C. It is essentially a chain of if/else statements. The following code illustrates how a switch-case structure can be implemented on the AVR.

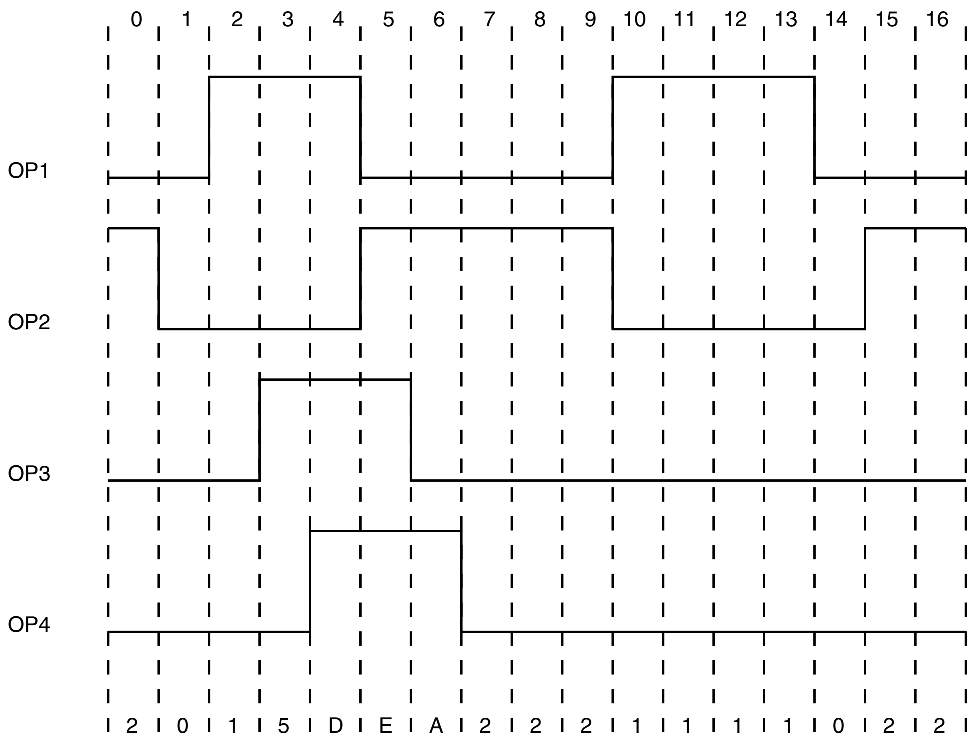


Figure 6.50 An arbitrary waveform example.

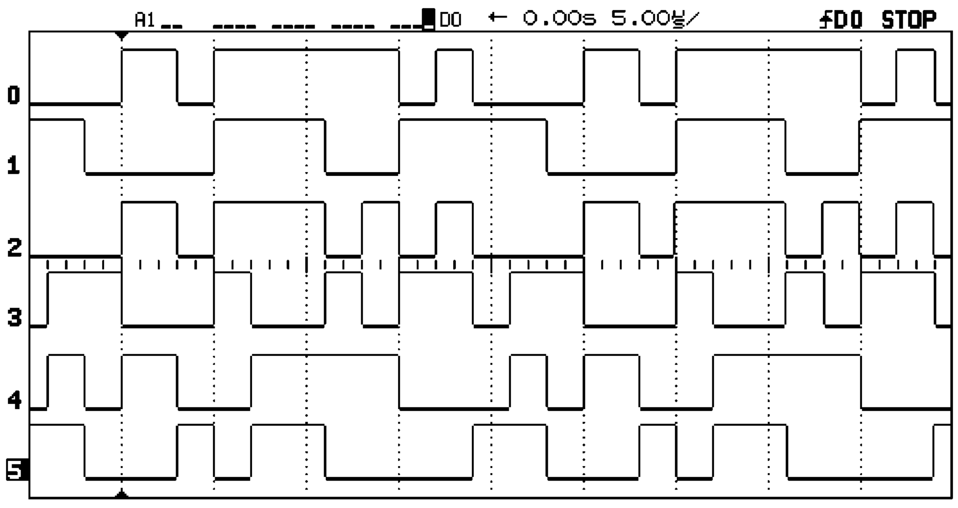


Figure 6.51 An arbitrary waveform generated by the AVR processor and captured on a logic analyzer.

```
.equ option1='A'
.equ option2='B'
.equ option3='C'
.equ option4='D'
.equ option5='E'
.equ option6='F'
.def rreg=r18
;**********************************************
;Subroutine to implement a switch-case statememt
;**********************************************
sub_case:  rcall get_byte      ;get an argument
begin_case: cpi rreg, option1  ;check if argu=option1
        brne chk2              ;else compare with option2
        rcall sub_opt1         ;if yes execute subroutine
                               ;sub_opt1
chk2:     cpi rreg, option2    ;check if argu=option2
        brne chk3              ;else compare with option3
        rcall sub_opt2         ;if yes execute subroutine
                               ;sub_opt2
chk3:     cpi rreg, option3    ;check if argu=option3
        brne chk4              ;else compare with option4
        rcall sub_opt3         ;if yes execute subroutine
                               ;sub_opt3
chk4:     cpi rreg, option4    ;check if argu=option4
        brne chk5              ;else compare with option5
        rcall sub_opt4         ;if yes execute subroutine
                               ;sub_opt4
chk5:     cpi rreg, option5    ;check if argu=option5
         brne chk6             ;else compare with option6
         rcall sub_opt5        ;if yes execute subroutine
                               ;sub_opt5
chk6:     cpi rreg, option6    ;check if argu=option6
         brne chk_default      ;nothing matches.
         rcall sub_opt6        ;if yes execute subroutine
                               ;sub_opt6
         ret
chk_default: rcall sub_default ;else execute a default
                               ;subroutine
         ret
```

6.18 Implementing a Finite State Machine

A finite state machine (FSM) is a formal concept with many applications. In general, a finite state machine is a device which has some inputs and some outputs. It stores the state of the machine, and depending upon the inputs, changes the state value. The outputs depend upon the state value. FSM has:

1. A set of finite states.
2. Inputs.
3. A transition function for each state.
4. Outputs.

FSM has great applications in pattern recognition, vending machine applications, etc. You can create a traffic light controller modeled on the FSM concept.

The first step in modeling the requirement as an FSM is to create a state transition table and an output table as illustrated in Tables 6.4 and 6.5. Another way is to create a bubble diagram description as illustrated in Figure 6.52. Some of the transitions from one state are conditional, while many others are unconditional. It is valid to have an FSM with all unconditional transitions. An ordinary binary counter is an example of an FSM with unconditional state transitions. The counter just hops from one state to the next and then resets to the starting state. The minimum time for which the FSM remans in a state is determined by the clock period of the system. In a microcontroller-based system, you could put appropriate delay routines between each transition.

The state of the FSM is maintained with the help of a variable (in a digital circuit, with the help of a register). It is important that all the possible states of the register are accounted for and, importantly, after power on, the register (or the variable) must be initialized with a default state, otherwise the FSM will not work at all. A test code for implementing a finite state machine is available in the code directory in the file fsm.asm.

TABLE 6-4 STATE TRANSITION

CURRENT STATE	NEXT STATE (NS)
S0 X = 0 and Y = 0,	NS = S1
SO X = 1, NS = S0	
S1 X = 1 and Y = 0,	NS = S2
S1 X=1 and Y = 1,	NS = S3
S2	NS = S3
S3	NS = S4
S4	X = 1, NS = S4
S4	X = 0, NS = S0

TABLE 6-5 STATE OUTPUT

STATE OUTPUTS
SO A = 0, B = 23
S1 A = 1, B = 23
S2 A = 1, B = 32
S3 A = 0, B = 22
S4 A = 1, B = 44

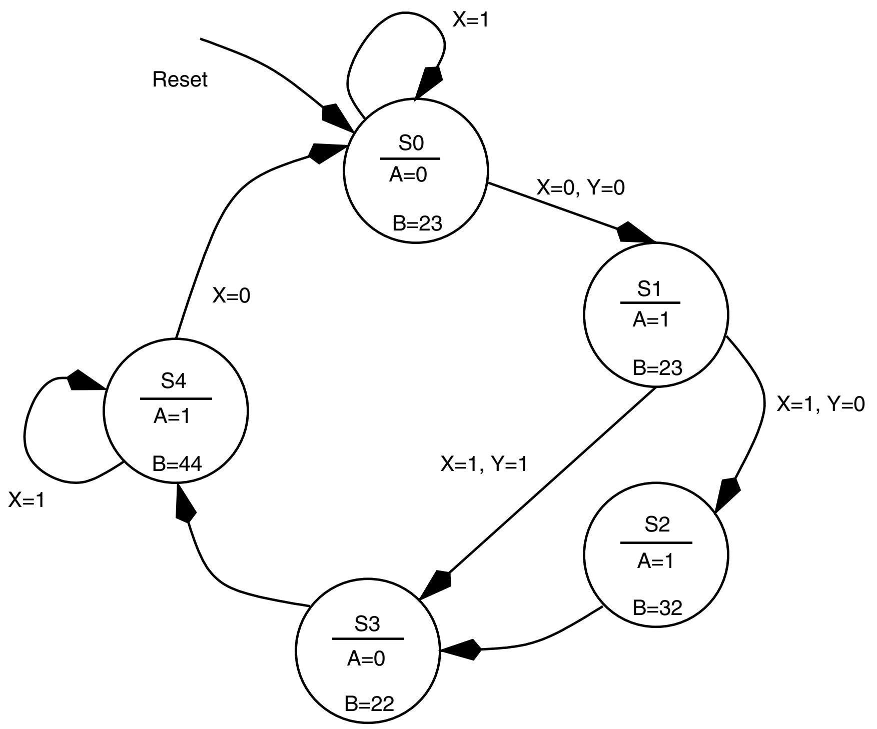


Figure 6.52 **A bubble diagram description of a state machine.**

6.19 Generating Random Numbers

Many applications such as toys and test pattern generators require random numbers. While it is almost impossible to generate a truly random number, one can approximate with a pseudorandom number.

One popular way to generate a pseudorandom number is to read the contents of a free-running counter. This is a simple scheme and can be easily implemented on the AVR controller with no extra hardware. Timer0 (or Timer1) is clocked at a certain clock frequency derived out of the system clock. Then, to get a random number, the Timer0 (or Timer1) register (TCNT0 or TCNT1) is read and what you get is a pseudorandom number. This scheme is used in the Dice project chapter.

Another way to generate a pseudorandom number is to use the concept of Linear Feedback Shift Register (LFSR). LFSRs are ordinary shift registers with some outputs (called taps) feeding the input (see Figure 16-3 in Chapter 16). LFSRs have an interesting property that if the feedback taps are chosen carefully, then outputs cycle through $2^n - 1$ sequences, for an n-bit LFSR. The sequence then repeats after $2^n - 1$ instances. If the output sequences are observed, they appear to be random. An 8-bit LFSR is illustrated in Figure 16.3. An 8-bit LFSR will have a sequence length of 255. Similarly, a 16-bit LFSR would have a length of 65535 and so on.

The LFSR can be easily implemented on the AVR controller. The LFSR must be initialized with a nonzero seed value. After the LFSR is initialized, it is clocked by shifting the values to the left and loading a new bit into the bit0 of the shift register. The new bit that is loaded into the bit0 of the shift register is calculated by XORing the bits at the selected taps of the LFSR. All of these operations can be implemented using the AVR instructions. A working example of an 8-bit LFSR implemented on the AVR controller is presented in the electronic lock project chapter.

BITS	SEQUENCE LENGTH	TAPS
9	511	3,8
10	1023	2,9
11	2047	1,10
12	4095	0,3,5,11
13	8191	0,2,3,12
14	16,383	0,2,4,13
15	32,767	0,14
16	65535	1,2,4,15
17	131,071	2,16
18	262,143	6,17
19	524,287	0,1,4,18
20	1,048,575	2,19