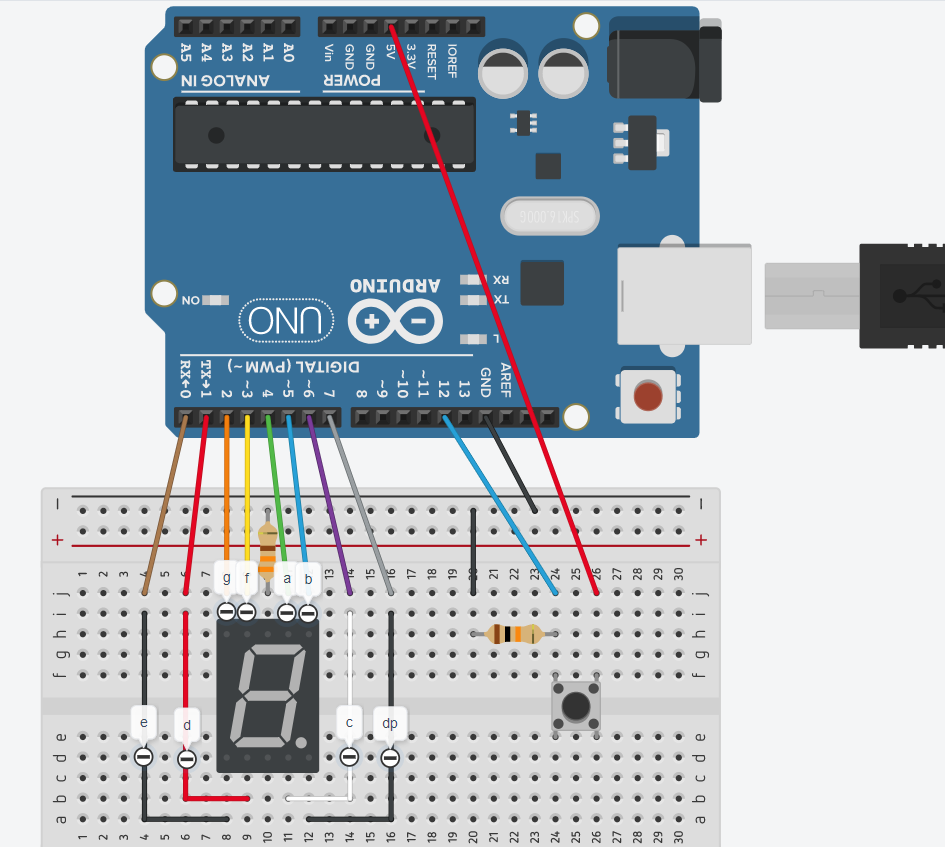
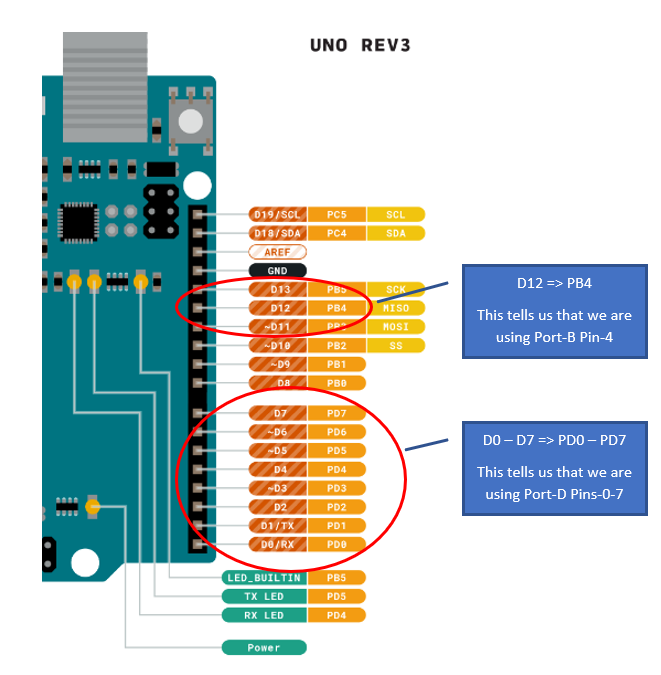
This will be a review of the Seven Segment Display circuit and code.

The following diagram shows the circuit with a common cathode seven segment component connected to Port-D Pins 0-7, and a push button switch connected to Port-B Pin-4.



We determine the Port-X and Pin-N from a Pin-Out diagram for the MCU as below:



Configuration decisions:

|  |
| --- |
| The Seven Segment is a Common Cathode (-), meaning that the wires connected to pins [a-g, & dp] are going to provide positive voltage, and the common pin is connected to a (-) ground via a resistor that is shared by each of the individual LED circuits. |

|  |
| --- |
| We create a Digit “1” on a seven-segment display by putting voltage on the wires connected to pins B and C. The current flows through the LED connected to each of these pins, and then onto a common ground wire running to the “Gnd” pins at the top and bottom of the component. We put a resistor connected to one of these pins running to a ground to complete the circuits.  If we have a resistor “R1”, then we have circuits:   * “b” -> R1 -> ground * “c” -> R1 -> ground   The single resistor provides the same resistance to both circuits, so each circuit has the same current. |

Configure Input Port Pin

We can compare the input circuit on our seven-segment display to circuits from our GPIO exercise as below:

|  |  |  |
| --- | --- | --- |
| Input Switch | Switch Open | Switch Closed |
|  |  |  |

What we have is a circuit with current flowing from a voltage source to a switch, and then to an input pin.

The input pin is held low by a pull-down circuit when the switch is open by connecting the pin to ground. This is only meant to catch stray voltage (noise) that gets into the circuit between the switch and the pin.

When the switch closes, current takes the easiest path to a ground. Since the pull-down resistor is a large quantity of 10K ohms, the easier path is to the input pin that is internally connected to ground inside the Arduino.

The input pin is configured as Input with High Impedance, as the Pin will stay low (0) while the switch is open, and an external voltage will provide current to make the Pin high (1) when the switch is closed.

; set Pin-4 on Port-B to input

cbi DDRB,DDB4

; set Pin-4 on Port-B to high-impedance

cbi PORTB,PB4

We clear the direction register for Port-B for Pin-4 to designate that the pin is in “Input” mode, and we clear the Port-B output register for Pin-4 to set the input pin in High Impedance.

Configure Output Port Pins

We now know that our Port-D pins are going to be output, and each will be set high to light the segment they connect to and cleared low to not display a segment.

Since we used all 8 pins of Port-D, we can use the OUT instruction to set the entire direction register in one instruction.

; set all pins on Port-D to output

ldi r16,0xFF

out DDRD,r16

To set all Pins on Port-D to output we load hex 0xFF or $FF or decimal 255, or binary 0b11111111 into a GPR register. The Load Immediate instruction LDI will only work with registers R16-R31.

|  |
| --- |
| The OUT instruction takes an I/O Address as a destination operand. We need to set the Direction Register for Port-D, so we us the constant DDRD with equals “$0A”. We call this an offset address as it is really 10 bytes from the start of I/O memory, or $20.  The actual address of DDRD is $2A. This means we could have used Store Direct to SRAM instruction STS as:  sts $2A,r16 |

We choose OUT over STS because OUT uses the I/O constants that make the instruction more readable, and also because OUT is a 16-bit instruction that takes 1 cycle to process, and STS is a 32-bit instruction that takes 2 cycles to process.

|  |
| --- |
| Port direction register configuration and input pin circuits for high impedance or pull-down through the output register should only be done once in a program (typically). These values do not change during the running of the program. |

Lookup Table Configuration

We had the following table of bits to represent the digits we would display on the seven-segment component:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | lsb |  |  |  |  |  |  | msb |  |  |
| Port-D | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |  |
| Digit | e | d | g | f | a | b | c | dp |  | Hex |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | = | 7B |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | = | 60 |
| 2 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | = | 37 |
| 3 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | = | 76 |
| 4 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | = | 6C |
| 5 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | = | 5E |
| 6 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | = | 5F |
| 7 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | = | 70 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | = | 7F |
| 9 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | = | 7C |

We aligned the bits based on the alignment of the external pins of Port-D on the Arduino.



The value $7B that represents digit ‘0’ is the reverse binary string 0b01111011. It helps us to store it this was as we can simply place this value in the Output Register of Port-D and we have designated the correct ‘1’ and ‘0’ for each pin to output or don’t output current to the connected pin of the Seven Segment.

Let’s look at an example. We will use digit ‘4’ to illustrate how the circuit works.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | lsb |  |  |  |  |  |  | msb |  |  |
| Port-D | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |  |
| Digit | e | d | g | f | a | b | c | dp |  | Hex |
| 4 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | = | 6C |

; display digit '4'

ldi r16,$6C

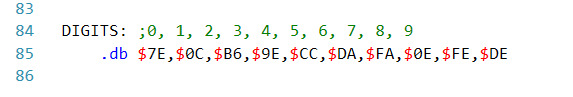
out PORTD,r16

This places value $6C or 0b01101100 at I/O address $0B or SRAM address $2B.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  | **Value** | | | | | | | |  | | **Mem.** | **I/O** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** | **Name** | | $20 | $00 |  |  |  |  |  |  |  |  |  | | $21 | $01 |  |  |  |  |  |  |  |  |  | | $22 | $02 |  |  |  |  |  |  |  |  |  | | $23 | $03 |  |  |  |  |  |  |  |  | PINB | | $24 | $04 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | DDRB | | $25 | $05 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | PORTB | | $26 | $06 |  |  |  |  |  |  |  |  | PINC | | $27 | $07 |  |  |  |  |  |  |  |  | DDRC | | $28 | $08 |  |  |  |  |  |  |  |  | PORTC | | $29 | $09 |  |  |  |  |  |  |  |  | PIND | | $2A | $0A | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | DDRD | | $2B | $0B | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | PORTD | | $2C | $0C |  |  |  |  |  |  |  |  |  | | $2D | $0D |  |  |  |  |  |  |  |  |  | |  |  | **dp** | **c** | **b** | **a** | **f** | **g** | **d** | **e** |  | | DDB4 set to Input  PB4 set to high impedance  DDD0-7 set to output  PORTB = $6C |

|  |  |
| --- | --- |
|  | We see that Pins 2, 3, 5 and 6 are set high, sending current through pins g, f, b, and c of the seven-segment display, and then through the common resistor to ground.  The digit ‘4’ is displayed through the lit LED’s. |

We place the lookup table at the bottom of our program:



We give the memory area a label ‘DIGITS’. The assembler will treat this label as the address of the first 2 bytes defined: $7E, $0C. Why 2-bytes?

The directive ‘.db’ tells the assembler that these values will be placed in program memory along with our instructions, and program memory is 2-bytes per memory address.

It is important that the address of these bytes never end up in the program counter, i.e. we need to ensure that there is some form of jump or endless loop above these defined bytes in program code. If we let the program counter index to these values, the bytes will be fetched as if they were instruction machine code, and the Decode Unit will try to convert them to something the ALU can execute.

We don’t try to define the bytes first in the program, as the Program Counter register is set to $00 when the board resets (or is first booted up), and the values would be fetched then at the start of the program.

Configure Pointer

Since the lookup table is defined program memory, we must use the Load Indirect from Program Memory instruction LPM with the Z pointer. The LPM instruction only works with the Z pointer.

If our lookup table was defined in Internal SRAM data memory, then we would use the LD instruction with either the X, Y, or Z instruction. Let’s say that we did define the lookup table in Internal SRAM starting at address $100. First, there is no way to define it up front, we would need to use LDI instructions to load each byte to a GPR, and then store that byte in SRAM using Store Direct STS, or we could initialize a pointer and use Store Indirect with instruction ST.

For instance, we could initialize one of the pointer to Address $100 like:

; set X pointer to Internal SRAM

ldi XH,$10

ldi XL,$00

Then we could start loading the bytes to a GPR and store it at each consecutive byte in Internal SRAM using the X pointer with Post Increment.

ldi r16,$7E

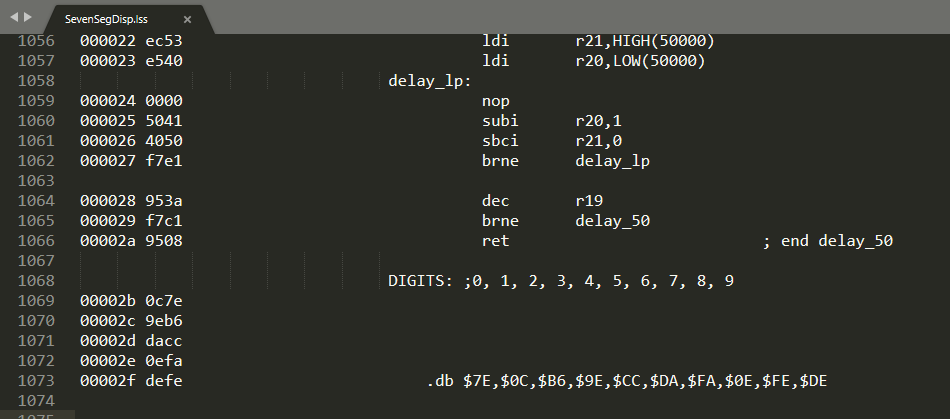
st X+,r16

ldi r16,$0C

st X+,r16

And so on to store each byte, as the post increment add 1 to the Address in XH:XL each time.

Since our lookup table is being defined in program memory, and program memory has 2 bytes at each address, we must do something special to the address. The system requires that we left shift the address one bit. For example, if we look at the bottom of the Listing file ‘.lss’ that the assembler creates in the Debug folder of our project, we see the following at the bottom of the code:



Notice the line numbers in my text editor have us around line 1068 where the ‘Digits:’ array is defined. This is because the entire “m328def.inc” file has been copied into our source code by the pre-stage processor of the assembler.

The first set of hex numbers ‘00002b’ starting on line 1069 show that address $002b hold the first two bytes of the array in Little Endian format $0C7e, where the first byte $7E represents the low-byte at that address.

Thus label ‘DIGITS’ equals $002b.

Now, when we initialize the Z pointer for use with LPM, we do it with code like the following.

; set Z pointer to nxt\_dig 0

ldi ZH,HIGH(DIGITS<<1)

ldi ZL,LOW(DIGITS<<1)

We can illustrate this as:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Before shift: | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | High Byte ZH | | | | | | | | Value | Low Byte ZL | | | | | | | | Value | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $00 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | $2B | |
|  |  |
| After shift: | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | High Byte ZH | | | | | | | | Value | Low Byte ZL | | | | | | | | Value | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $00 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | $56 | |

Note: each bit was shifted to the left 1 in the lower result, with a 0 added to the LSB of the low byte.

I greyed out the highest 2 bits of the addresses above, since the PC is only 14 bits. This shows that it didn’t matter that we left shifted the address, as we will never have program addresses above the 16k of addressable program memory or 214 = 16,384 bytes.

The CPU knows that the LPM instruction is providing a left-shifted address, so it uses the left 13 bits to represent the address, which are 0 0000 0010 1011 or $2B, and then the ‘lsb’ when 0 means the low byte at that program memory address, and when we increment the Z pointer the address is still $2B with a 1 in the ‘lsb’ which means the high byte at that program memory address.

Initialize Display with Digit ‘0’

I choose to have a register dedicated to tracking the next digit to display on the seven-segment.

.def nxt\_dig = r16 ; nxt\_dig displayed

I initialize it to Zero to indicate that is the next digit to be displayed.

; set next nxt\_dig to 0

eor nxt\_dig,nxt\_dig ; nxt\_dig being displayed

I choose to use an Exclusive OR to clear the register. This instruction is used for clearing registers in larger data-set assembly as it is always the smallest possible instruction, i.e. a 4 digit code (‘1111’), and then a 4 digit value to identify the destination register and a 4 digit value to identify the source register, with some 0’s as filler, so a 16-bit instruction that takes 1 cycle to execute. Now an LDI Rd,0 would also only take up 16-bits and take 1 cycle to execute in AVR assembly, but if this was 32 or 64 bit assembly, a similar instruction would take up 4 to 8 bytes of memory and take at least 2 instruction cycles to fetch/decode/execute. So Exclusive OR has been traditionally used.

Our AVR assembly does contain a CLR instruction, that performs and XOR of a GPR with itself, so I could have used:

; set next nxt\_dig to 0

clr nxt\_dig ; nxt\_dig being displayed

Call Display Digit

The display digit subroutine takes in the next digit to display through the GPR defined as ‘nxt\_dig’.

The routine starts by loading the current byte from program memory pointed to by the Z pointer and increments the Z pointer.

disp\_digit:

lpm r1,Z+ ; load nxt\_dig at Z and increment

It then outputs that whole byte to Port-D to display the digit on the seven-segment.

out PORTD,r1 ; output nxt\_dig to Port-D

It then increments the next digit pointer so we know what digit would be shown next.

inc nxt\_dig ; nxt\_dig index++

It then compares the value of ‘nxt\_dig’ to 10. If (nxt\_dig == 10) then we have displayed digits 0 – 9 and we need to start over.

cpi nxt\_dig,10 ; if (nxt\_dig < 10)

The compare only sets flags in the Status Register SREG by subtracting 10 from ‘nxt\_dig’. If nxt\_dig == 10, then 10 – 10 = 0 and the Zero flag is set Z= 1, else the Zero flag is cleared Z=0. The Branch Not Equal BRNE will jump to ‘disp\_ret’ if ‘nxt\_dig’ != 10 then, i.e. Z=0.

brne disp\_ret ; then return

; else

Otherwise, ‘nxt\_dig’ did equal 10, so now we need to reset ‘nxt\_dig’ to zero. Once again, I could have used the CLR instruction.

eor nxt\_dig,nxt\_dig ; set nxt\_dig=0

We also need to set the Z pointer back to the original $002b address. We could use LDI instruction with HIGH/LOW to set ZH:ZL using the ‘digits:’ label, but we know we incremented the Z pointer 10 times, so we can just subtract 10 from the address in the pointer regsiters and it should be back to where we started. We can use the Subtract Immediate from Word instruction SBIW to subtract 10 from the register pair for the Z pointer.

sbiw ZH:ZL,10 ; reset Z pointer

Then we simply return from the subroutine. Remember, the CALL instruction that called this subroutine pushed the address of the PC on the stack, and the RET instruction is going to pop that address from the stack back into the PC register.

disp\_ret:

ret ; end disp\_digit

Main Program Loop

The main processing loop checks a high in the Port-B input register PINB for Pin-4.

get\_input:

The check is performed by using a Skip if I/O Bit is Cleared SBIC instruction.

sbic PINB,PINB4 ; see if button pressed

If PINB4==0, then the next instruction is skipped, i.e. the call to ‘disp\_digit’ doesn’t happen and we move on to the delay processing.

If PINB4==1, then the skip doesn’t happen and we execute the call to ‘disp\_digit’ to display the next digit on the seven-segment.

call disp\_digit ; display next nxt\_dig

The pause routine is like other ones we have used. The subroutine is designed to wait 50ms time a value in GPR r19. We just want to wait long enough for a person pressing the switch button to release it, so we don’t catch PINB4 high multiple times for a single button press.

; pause between checking for input

ldi r19,1 ; 1 \* 50ms = .025/sec

call delay\_50

The unconditional jump here keeps us in an endless loop going back to the top of the main processing loop to check for more input. The ‘end\_main:’ label only serves to document that this ends the main area of the program.

rjmp get\_input ; check for input again

end\_main: ; end of program

Delay Subroutine

The only difference in this delay routine is that it uses a register pair R21:R20 to represent the high:low bytes of the number of loops for the delay routine, here set to 50,000 in decimal. The HIGH and LOW functions extract the high and low bytes of 50,000 or $C350.

The routine uses Subtract Immediate to subtract 1 from the low byte, i.e. decrement the low byte. We use subtract as it will set the borrow flag ‘C’ if R20 gets to zero and we subtract again.

We then use Subtract Immediate with Borrow SBCI that subtract 0 from the high byte and then subtracts the Carry flag, i.e. subtract 1 if there was a borrow from the low byte. If both SUBI and SBCI result in R21=0 and R20=0, then the Zero flag is set Z=1.

The Branch Not Equal keeps the loop going until Z=1.

The SUBI takes 1 cycle, the SBCI takes 1 cycle, and the BRNE takes 2/1 cycles, so with 1 NOP we have 5 cycles being repeated 50,000 times, or 5 x 50000 = 250,000 micro seconds and 1 second = 1,000,000 micro seconds, so our delay runs for .25 seconds if R19=1, or R19 \* .25 seconds if higher.

; delay 50ms

; r19 \* 50ms

delay\_50:

ldi r21,HIGH(50000)

ldi r20,LOW(50000)

delay\_lp:

nop

subi r20,1

sbci r21,0

brne delay\_lp

dec r19

brne delay\_50

ret ; end delay\_50

Note: had we used a register pair in the range of R25:R24 or higher, we could have used the Subtract Immediate from Word SBIW instruction. It only works with the highest 8 registers.

; delay 50ms

delay\_50:

ldi r25,HIGH(50000)

ldi r24,LOW(50000)

delay\_lp:

nop

sbiw r25:r24,1

brne delay\_lp

dec r19

brne delay\_50

ret ; end delay\_50

The SBIW instruction takes 2 cycles, so we still have 5 cycles being repeated.

Configure Input Switch Pull-Up

In the final iteration of the program we change the input switch to a pull-up configuration. The circuit changes as follows.

|  |  |  |
| --- | --- | --- |
| Input Switch | Switch Open | Switch Closed |
|  |  |  |

The input pin is configured the same in the Direction Register.

; set Pin-4 on Port-B to input

cbi DDRB,4

Then the pin is configured for pull-up in the Output Register.

; set Pin-4 on Port-B to pull-up

sbi PORTB,4

Now the Pin PINB4 will read high (1) while the switch is open, and low (0) when the switch is closed.

The only required change is where we check for input.

get\_input:

sbis PINB,4 ; see if button pressed

call disp\_digit ; display next nxt\_dig

We simply change the Skip if I/O Bit Cleared SBIC to Skip if I/O Bit Set SBIS.