CONTENTS

Exp 1 - Simulator Tutorial and the P_1 Template

Anas Ashraf

I. The Big Picture of the P_1 Template

A. Program Hierarchy

The big picture when you are trying to understand the program is to know that P_1 simply runs the mainline program which is shown in figure 1. Everything that could be defined as the identity of this template is contained in figure 1. We see that the mainline first initializes everything by setting all the appropriate values before executing everything in the loop. Thus we see it calls $\boxed{\textit{rcall Initial}}$ only once to set all values. This can be thought of as tuning the parameters when something is turned on for the first time because when the microcontroller is turned off it erases all of its previously saved memory since we only have volatile memory. Thus every time it is plugged in or turned on we must initialize all the values for our purpose. It can also be done as a safety measure if you are running several different program templates in one session, and if you want to quickly toggle between the settings and not have one program templates settings carry over, it is necessary to initialize all the settings so it is best suited for your needs. It is important to note that the rcall simply means call a label and the label Initial is not defined as of yet as it is unimportant for the big picture.

Then we jump into a loop which executes forever as is seen in line 58 that reads $\boxed{\textit{Loop}}$. It should be important to note that we don't need to initialize our microcontroller every single loop because we still want the same settings in every loop. It would waste time to initialize everything in every single loop, thus $\boxed{\textit{rcall Initial}}$ is only called once in the entire mainline program and never called again so far as the execution is not halted. It should be important to note that if we skip the instructions in lines 59-61 then the final line is $\boxed{\textit{bra Loop}}$. bra stands for branch and it means we should go to the label called Loop which you will notice is in line 58. Thus line 62 tells us we should go back to line 58. So the majority of the runtime of this program is spent on lines 59-61.

```
54  ;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
55
56  Mainline
57          rcall  Initial          ;Initialize everything
58  Loop
59          btg  PORTC,RC2          ;Toggle pin, to support measuring loop time
60          rcall  BlinkAlive       ;Blink "Alive" LED
61          rcall  LoopTime         ;Make looptime be ten milliseconds
62          bra  Loop
63
```

Fig. 1. The mainline program

B. The Initial Label

We have seen in the mainline program shown in figure 1, line 57 is $\boxed{\textit{rcall Initial}}$ which simply means call the Initial label. However, the initial label is not defined in figure 1, it is defined in figure 2. You will notice in line 57 (figure 1) when the program calls $\boxed{\textit{rcall Initial}}$ it actually jumps to the label in line 68 (figure 2). It isn't important to go into details in what happens in lines 69-76 but the main idea is that each of the registers labelled in red (e.g. ADCON1, TRISA, TRISB, etc.) are 8 bit registers. We set all of their values to the 8 bit sequence (of 1's and 0's) preceding the register declaration. The comments included next to the instructions are sufficient in understanding that the bit sequence that is passed into the 8 bit registers merely sets it up so we may use it later, that is, it sets up the input and output (I/O) for each register.

We should pay special attention to line 75 which essentially initializes Timer0 register. It turns on bit 7 which allows the timer to be turned on. If bit 7 is off or 0 it means Timer0 is turned off. Further, we also turn on bit 3 which bypasses the prescaler. A prescaler is an electronic counting circuit used to reduce a high frequency electrical signal to a lower frequency by integer division. The prescaler takes the basic timer clock frequency and divides it by some value before feeding it to the timer, according to how the prescaler registers are configured. The prescaler values, referred to as prescales, that may be configured might be limited to a few fixed values (powers of 2), or they may be any integer value from 1 to 2^P, where P is the number of prescaler bits. The purpose of the prescaler is to allow the timer to be clocked at the rate a user desires. For shorter (8 and 16-bit) timers, there will often be a tradeoff between resolution (high resolution requires a high clock rate) and range (high clock rates cause the timer to overflow more quickly). For example, one cannot (without some tricks) achieve $1\mu s$ resolution and a 1 sec maximum period using a 16-bit timer. In this example using $1\mu s$ resolution would limit the period to about 65ms maximum. However, the prescaler allows tweaking the ratio between resolution and maximum period to achieve a desired effect.

The last command in this initalizing subroutine is *return* in line 77 which basically tells the compiler it must go back to the instruction it came from and move on. In this instance it would go back to line 57 (figure 1) which called the Initial label and move on to the next line.

```
66  ; This subroutine performs all initializations of variables and registers.
67
68  Initial
69          MOVLF  B'10001110',ADCON1  ;Enable PORTA & PORTE digital I/O pins
70          MOVLF  B'11100001',TRISA  ;Set I/O for PORTA 0 = output, 1 = input
71          MOVLF  B'11011100',TRISB  ;Set I/O for PORTB
72          MOVLF  B'11010000',TRISC  ;Set I/O for PORTC
73          MOVLF  B'00001111',TRISD  ;Set I/O for PORTD
74          MOVLF  B'00000000',TRISE  ;Set I/O for PORTE
75          MOVLF  B'10001000',T0CON  ;Set up Timer0 for a looptime of 10 ms;  bit7=1 enables timer; bit3=1 bypass prescaler
76          MOVLF  B'00010000',PORTA  ;Turn off all four LEDs driven from PORTA ; See pin diagrams of Page 5 in DataSheet
77          return
78
```

Fig. 2. The Initial Label defined

Before we move on we must also note the command that prefaces each of instruction in lines 69-76 which is *MOVLF*. So far we have implicitly interpreted as moving the bits into the register but it is important to note that this command is made up of several instructions and is thus a macro. The definition of *MOVLF* is in figure 3. We can see that the macro takes two arguments *literal, dest* based on line 37. Literal would be the value we want to move into the destination register. So far we only worked with binary values specified in the code for example as $\boxed{B'0101010'}$. Of course we can also pass decimal or hex values if we wanted to. Further, we know that *dest* is the destination register that we wish to change.

```
35  ;;;;;;; Macro definitions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
36
37  MOVLF    macro  literal,dest
38           movlw  literal           ;move literal value to WREG
39           movwf  dest              ;move WREG to f= dest, which is specified by user
40           endm
41
```

Fig. 3. Macro definition for MOVLF

II. The Main Loop

A. Toggling Bit RC2

We know in line 59 (figure 1) the first instruction in the loop is *btg PORTC, RC2*. btg stands for bit toggle and RC2 is the value of the third bit in the 8-bit register PORTC. Thus the first instruction at the beginning of each loop is to change the value of the third bit in register PORTC. We do this so we can measure the loop time. This helps keep track between one loop and the other. The first instruction in the loop is rather simple.

B. The BlinkAlive Label

The second instruction in the loop block (line 60 in figure 1) is *rcall BlinkAlive*. We already know that *rcall* means we need to call a label and in this instance the label is named *BlinkAlive*. So let us know see how this label is defined in figure 4. The first instruction in the subroutine is *bsf PORTA, RA4*, where bsf stands for bit set. Thus, we set the fifth bit in the register PORTA. Setting this fifth bit causes the LED D_2 to turn off because it is connected inversely to pin RA4. Thus, when RA4 is high D2 is turned off but when RA4 is low, D2 is turned on.

The next instruction is *decf ALIVECNT, F*, where decf means you decrement the value stored into the register ALIVECNT and store it back into ALIVECNT. So for instance if ALIVECNT has a value of 13, then decf will cause the new value to be

12. The final F in the instruction means we want to store the decremented value in the ALIVECNT register. If, instead we had *decf ALIVECNT, WREG* it would mean we decrement the value in ALIVECNT but do **NOT** store the new value in the same ALIVECNT register. Instead, we store it in the new WREG register.

The next instruction is *bnz BAend* which means if the value in ALIVECNT is **NOT** zero then you go to the label in line 122 called *BAend* and proceed from there, essentially skipping lines 120-121. If however, the value in ALIVECNT is exactly zero then we do **NOT** skip the lines 120-121, but instead proceed naturally to the next line. Doing so would land us on the instruction *MOVLF 250, ALIVECNT*. We have already defined the macro in subsection I-B. Essentially, what this instruction does is it feeds the value of 250 (a decimal value) into the 8-bit register of ALIVECNT. The binary for 250 is 11111010 thus we could have reasonably replaced this instruction with *MOVLF B'11111010', ALIVECNT* and it would have been equally valid. We chose to keep the decimal value 250 because its easier for a human to read and understand and also because this controls the amount of time in which we allow the LED to be ON or OFF. You will soon see that the current amount of time the LED is OFF is 2.5 seconds and we get that number because of $250 \cdot 10ms = 2.5sec$. We will show later how the $10ms$ came about, but it is important to keep the special number 250 in the back of your mind.

The last instruction is *bcf PORTA, RA4*, where bcf means clear bit, thus we clear the fifth bit in PORTA causing the LED D2 to turn on and stay on until it turned off again when we loop back to line 117 which sets RA4 again. The amount of time the LED is turned ON is 10ms (based on factors we will see later). Finally, we get to *return* which sends us back to where we came from in the mainline program.

Before we leave it might be a bit complicated understanding the branching in BlinkAlive so we present to you a pseudo-code representation of lines 116-123 in listing 1.

```
111  ;;;;;;; BlinkAlive subroutine ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
112  ;
113  ; This subroutine briefly blinks the LED next to the PIC every two-and-a-half
114  ; seconds.
115
116  BlinkAlive
117          bsf   PORTA,RA4           ;Turn off LED
118          decf   ALIVECNT,F         ;Decrement loop counter and return if not zero
119          bnz   BAend
120          MOVLF   250,ALIVECNT      ;Reinitialize BLNKCNT
121          bcf   PORTA,RA4           ;Turn on LED for ten milliseconds every 2.5 sec
122  BAend
123          return
```

Fig. 4. BlinkAlive label definition

```
def BlinkAlive
{
    set(Register = PORTA, bit=4);    #Turn OFF the LED D2
    ALIVECNT = ALIVECNT -1;
    if(ALIVECNT == 0)
    {
        ALIVECNT = 250;      #This number times 10ms is the amount of time for which the LED will stay OFF
        clear(Register = PORTA, bit=4); #Turn ON the LED D2
    }
    return;
}
```

Listing 1: Pseudocode representation for BlinkAlive subroutine in figure 4.

C. The LoopTime Subroutine

When we return from the BlinkAlive subroutine we go back to the mainline program in figure 1 and our very next instruction to execute is *rcall LoopTime*. This calls the label LoopTime and it is defined in figure 5. Recall we are working with an 8-bit machine so the only way to have a 16-bit counter is to get creative. We create a pseudo 16-bit register by joining both

TMR0H and TMR0L to make a cumulative of 16-bits. To understand the big picture you must first understand the instruction that reads *Bignum equ 65536-25000+12+2*. We first need to understand where the seemingly random numbers are coming from. Let us first try to find out the significance of 65536. Notice we have a 16-bit counter thus we have access to all values ranging from 0 to 2^{16}. The number $2^{16} = 65536$, thus we have 65536 states to work with. However, we do not want to wait 65536 clock periods. In this instance we want to wait 10ms given that the clock period is $4\mu s$. Thus the total number of clock periods we must wait is shown in equation 1.

$$
\begin{aligned}
\text{Desired Clock Cycles} &= \frac{\text{Desired Wait time}}{\text{Clock Period}} \\
\text{Desired Clock Cycles} &= \frac{10ms}{4\mu s} \\
\text{Desired Clock Cycles} &= 25000
\end{aligned}
\tag{1}
$$

From a maximum number of states of $2^{16} = 65536$ we only want to keep 25000, thus we must subtract $65536 - 25000 = 40536$ cycles from the total number of allowable cycles. However, we also need to account for the few cycles it will take to execute instructions that we don't want to be executed within the desired number of clock cycles we desire to stall our microcontroller for. Thus to account for those cycles we add 14 more cycles, but since they come from different sources, we split them up to designate so letting the additional buffer we need to account for be represented as $12 + 2$ cycles.

It is not entirely necessary to understand lines 93-108, what must be taken away is that this instruction set counts from 0 to the highest allowed value set by *Bignum* and then rolls over, essentially stalling for 10 ms. This is why the LED is ON in BlinkAlive for 10ms while being off for 2.5 seconds.

```
 79  ;;;;;;; LoopTime subroutine ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 80  ;
 81  ; This subroutine waits for Timer0 to complete its ten millisecond count
 82  ; sequence. It does so by waiting for sixteen-bit Timer0 to roll over. To obtain
 83  ; a period of precisely 10000/0.4 = 25000 clock periods, it needs to remove
 84  ; 65536-25000 or 40536 counts from the sixteen-bit count sequence.  The
 85  ; algorithm below first copies Timer0 to RAM, adds "Bignum" to the copy ,and
 86  ; then writes the result back to Timer0. It actually needs to add somewhat more
 87  ; counts to Timer0 than 40536.  The extra number of 12+2 counts added into
 88  ; "Bignum" makes the precise correction.
 89
 90  Bignum  equ     65536-25000+12+2
 91
 92  LoopTime
 93          btfss  INTCON,TMR0IF    ;Wait until ten milliseconds are up OR check if bit TMR0IF of INTCON == 1, skip next line if true
 94          bra  LoopTime
 95          movff  INTCON,INTCONCOPY  ;Disable all interrupts to CPU
 96          bcf  INTCON,GIEH
 97          movff  TMR0L,TMR0LCOPY  ;Read 16-bit counter at this moment
 98          movff  TMR0H,TMR0HCOPY
 99          movlw  low  Bignum
100          addwf  TMR0LCOPY,F
101          movlw  high  Bignum
102          addwfc  TMR0HCOPY,F
103          movff  TMR0HCOPY,TMR0H
104          movff  TMR0LCOPY,TMR0L  ;Write 16-bit counter at this moment
105          movf  INTCONCOPY,W      ;Restore GIEH interrupt enable bit
106          andlw  B'10000000'
107          iorwf  INTCON,F
108          bcf  INTCON,TMR0IF      ;Clear Timer0 flag
109          return
```

Fig. 5. LoopTime label definition

III. Final Flowchart

The final succinct summary of our results and finding are presented in figure 6. A flowchart breaks down everything into easy to follow steps.

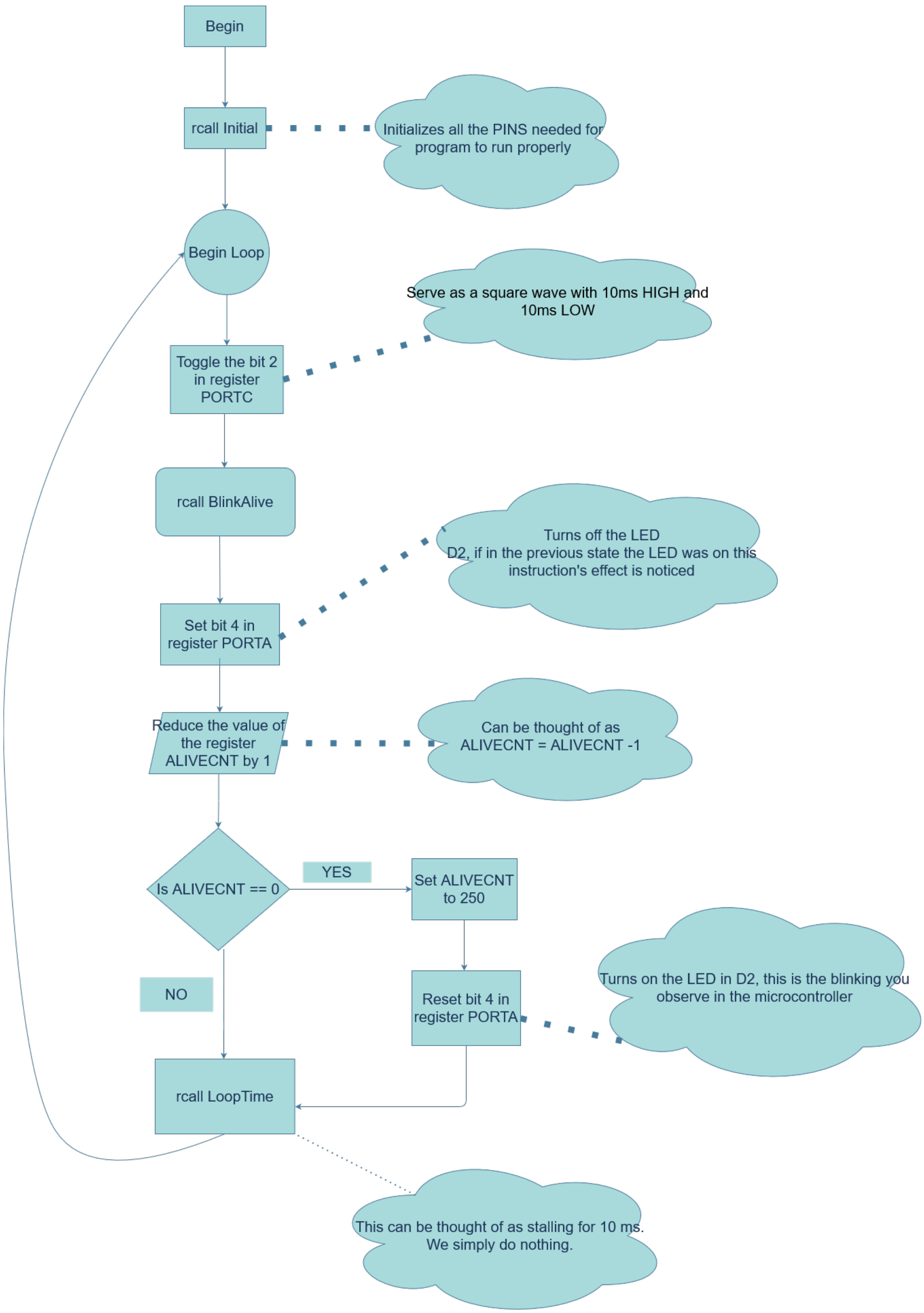

Fig. 6. General flowchart for the entire P_1 project template