Contents

List of Flowcharts

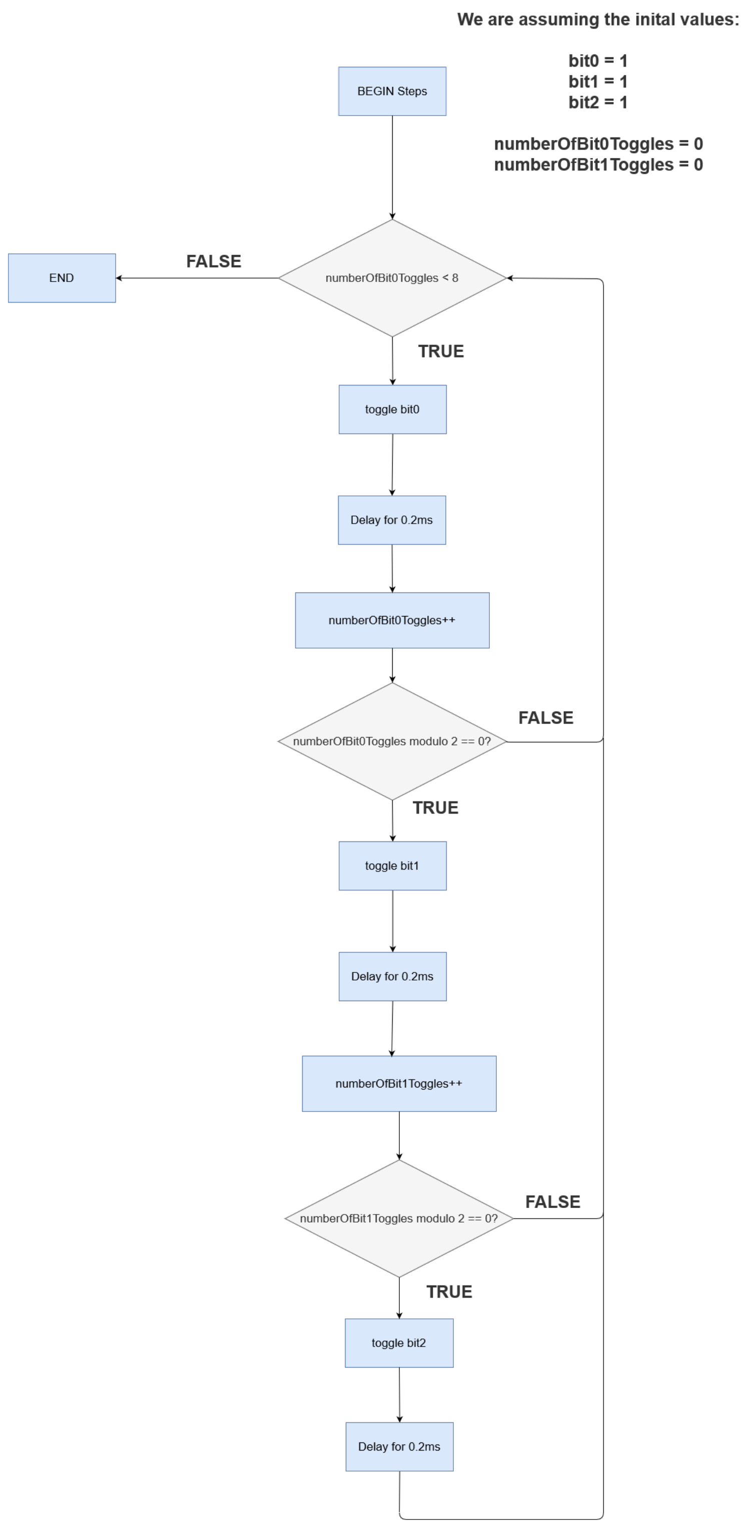


Fig. 1. Subroutine for Steps in RA1, RA2, and RA3 in LPISR

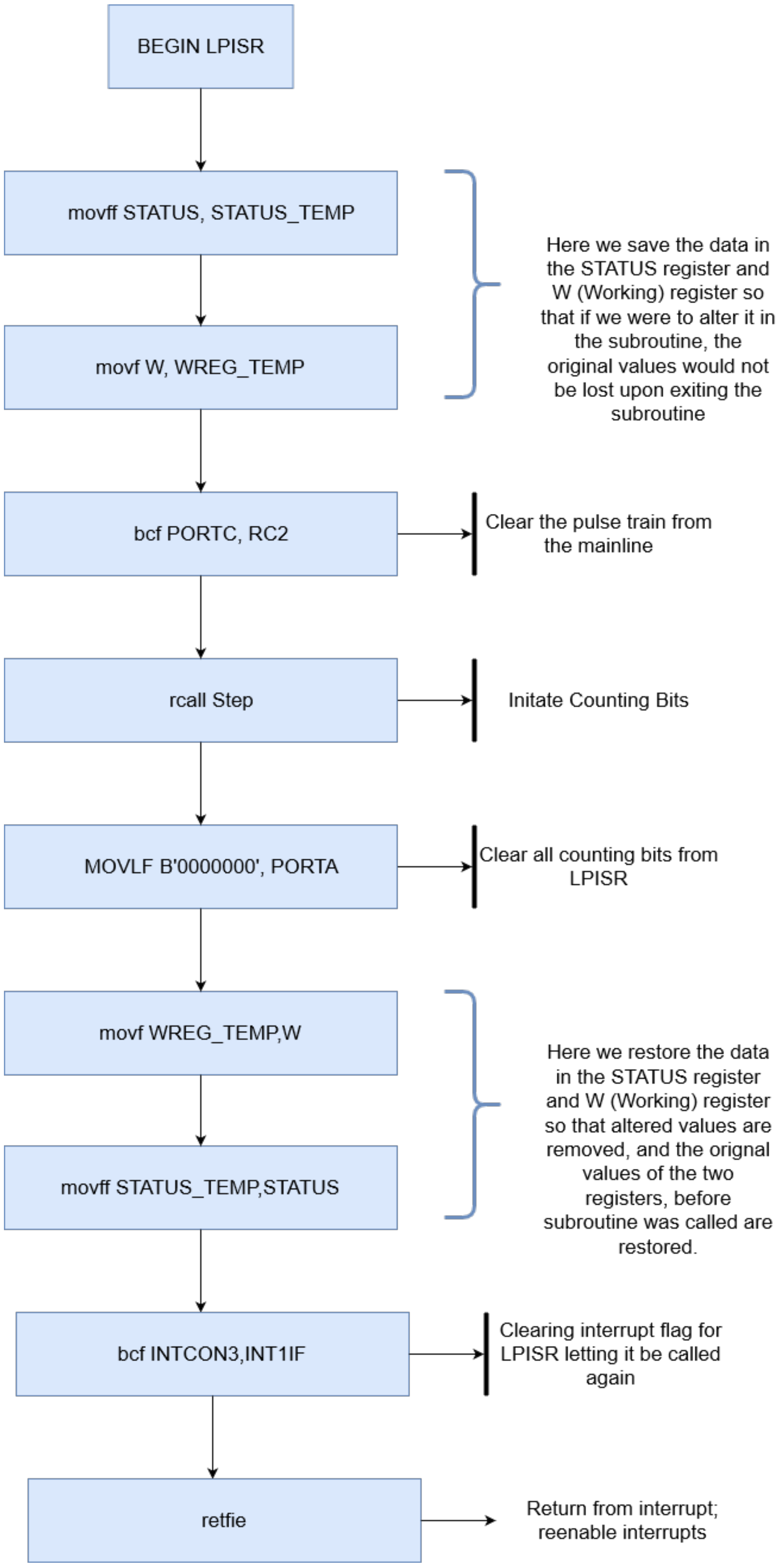

BEGIN LPISR
movff STATUS, STATUS_TEMP
movf W, WREG_TEMP
Here we save the data in the STATUS register and W (Working) register so that if we were to alter it in the subroutine, the original values would not be lost upon exiting the subroutine
bcf PORTC, RC2
Clear the pulse train from the mainline
rcall Step
Initate Counting Bits
MOVLF B'0000000', PORTA
Clear all counting bits from LPISR
movf WREG_TEMP,W
movff STATUS_TEMP,STATUS
Here we restore the data in the STATUS register and W (Working) register so that altered values are removed, and the orignal values of the two registers, before subroutine was called are restored.
bcf INTCON3,INT1IF
Clearing interrupt flag for LPISR letting it be called again
retfie
Return from interrupt; reenable interrupts

Fig. 2. Low Priority Interrupt Service Routine

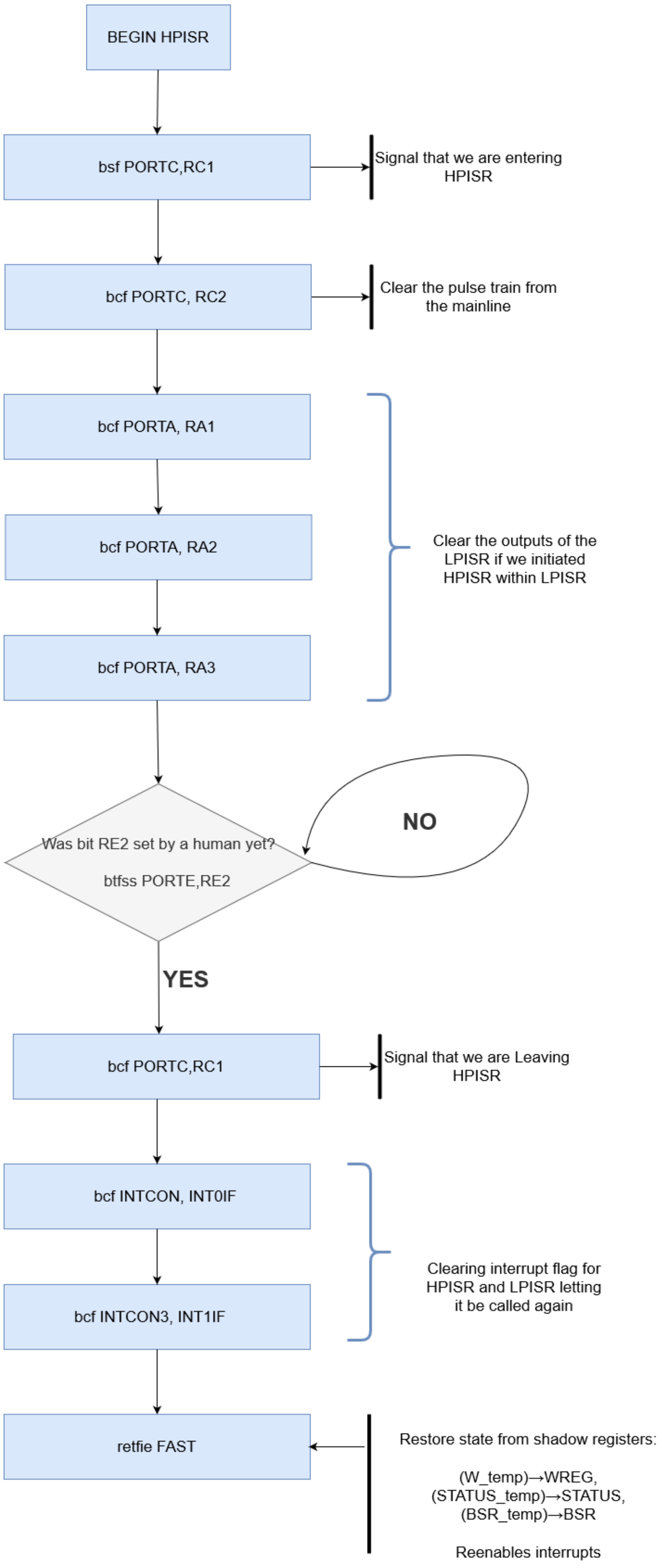

BEGIN HPISR
bsf PORTC,RC1
Signal that we are entering HPISR
bcf PORTC, RC2
Clear the pulse train from the mainline
bcf PORTA, RA1
bcf PORTA, RA2
bcf PORTA, RA3
Clear the outputs of the LPISR if we initiated HPISR within LPISR
Was bit RE2 set by a human yet?
btfss PORTE,RE2
NO
YES
bcf PORTC,RC1
Signal that we are Leaving HPISR
bcf INTCON, INT0IF
bcf INTCON3, INT1IF
Clearing interrupt flag for HPISR and LPISR letting it be called again
retfie FAST
Restore state from shadow registers:
(W_temp)→WREG,
(STATUS_temp)→STATUS,
(BSR_temp)→BSR
Reenables interrupts

Fig. 3. High Priority Interrupt Service Routine

Exp 3 - Interrupt Service Routine

Anas Ashraf

I. Introduction and Problem Outline

In experiment 3 we learn about interrupt service routines and how they can be initiated anywhere within the code to break out of the mainline loop to go into another (non-obligate infinite loop) after which we will eventually return to the mainloop. Hardware interrupts arise from electrical conditions or low-level protocols implemented in digital logic, are usually dispatched via a hard-coded table of interrupt vectors, asynchronously to the normal execution stream (as interrupt masking levels permit), often using a separate stack, and automatically entering into a different execution context (privilege level) for the duration of the interrupt handler's execution. In general, hardware interrupts and their handlers are used to handle high-priority conditions that require the interruption of the current code the processor is executing. In our template provided to us for this experiment we see that the mainline is simply a block of code that tells us to create a square wave of a 50% duty cycle and a half-period of 0.1ms. The output of this mainline can be seen in figure 4. Notice it is only confined to bit RC2 which (as we will see later on) is always cleared and stopped during the duration of any interrupt service routine. Our goal is to implement two classes of interrupt service routines that achieve two different tasks, a high priority interrupt service routine and a low priority interrupt service routine.

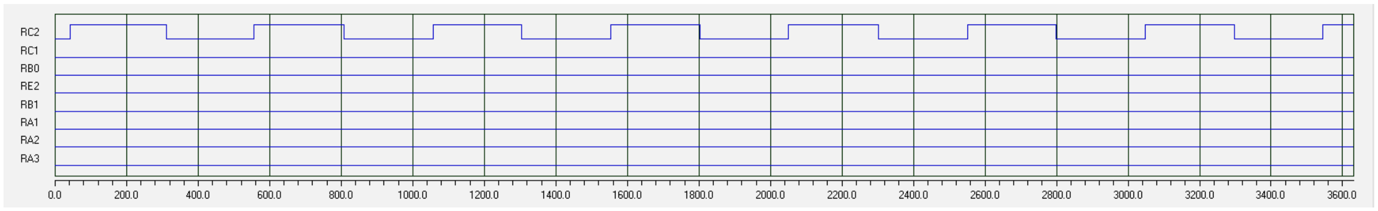


Fig. 4. Mainline output of 0.1ms and 50% duty cycle

II. Initial Subroutine

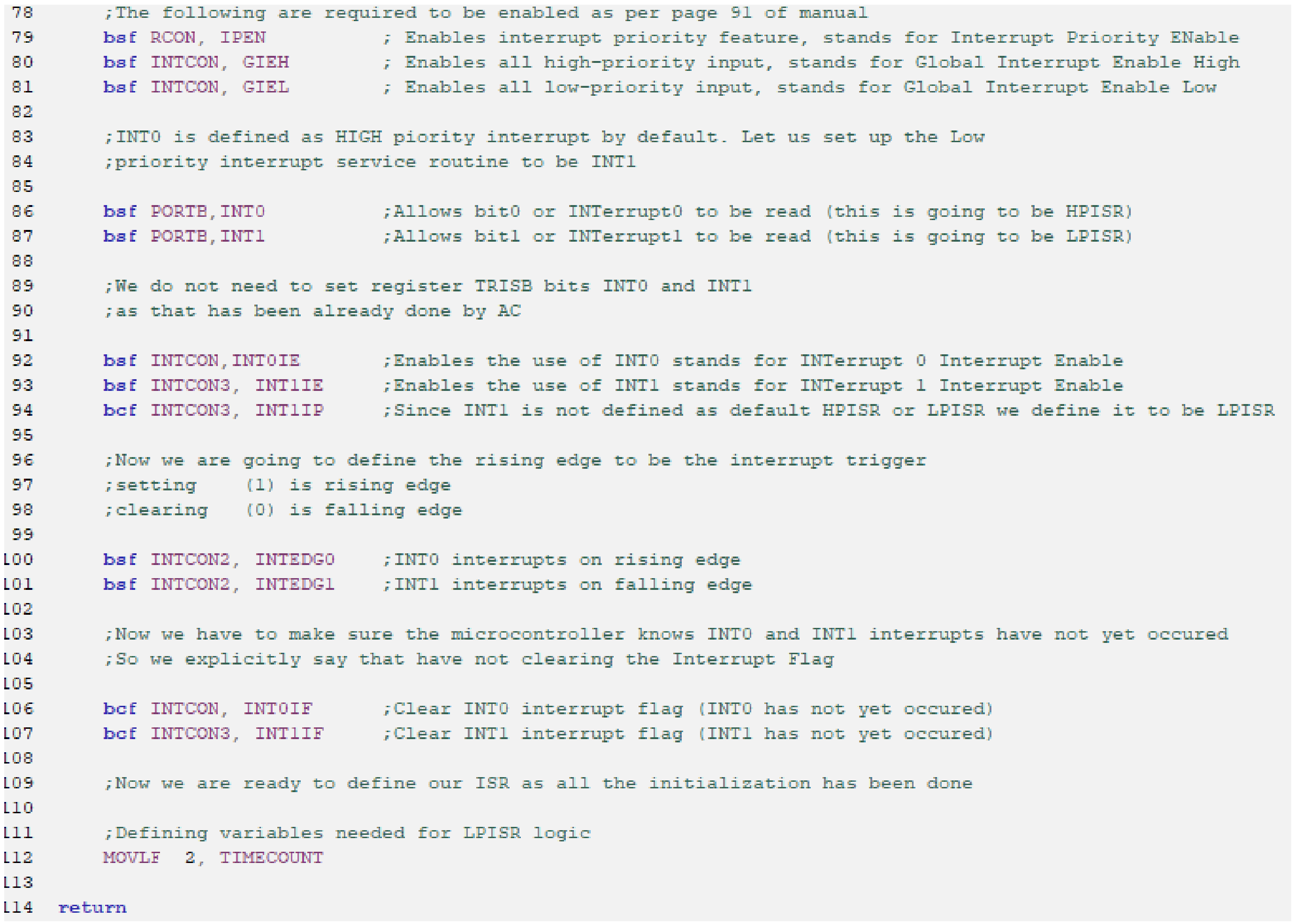
The initial subroutine, as expected, does not have any logic to implement. We simply have to set up the proper values to allow for interrupts to take place. The code can be seen in figure 5. It should be important to note that the code seen in figure 5 is not the complete code in the initial subroutine, this is simply the most important part that enables us to have interrupts in the micro-controller.

We first set bit IPEN in register RCON by calling $\boxed{\textit{bsf RCON, IPEN}}$. This basically enables the interrupts to have priority. Thus we can have a high priority interrupt subroutine (HPISR) and low priority interrupt subroutine (LPISR). Without this we only have one level of interrupts. Now we enable global interrupts from anywhere in the code that are both high and low. Without this the interrupts are constrained to only a small section of the code. We might sometimes want to disable the execution of interrupts during critical sections of the code, in those instances we would clear GIEH and GIEL. We know from page 91 of the manual that INT0 or RB0 is automatically defined as a HPISR but INT1 and INT2 can be defined to be either high or low priority interrupts. We choose to define INT1 as the low priority interrupt and choose to disregard INT2. In lines 86 and 87 we set INT0 and INT1 so that they are allowed to be read. As the comment in lines 89 and 90 states, we would have had to set bits 0 and 1 of register TRISB if we were to allow INT0 and INT1 as an input, but that has already been taken care of by the template provided to us in the section of code that has been omitted. Lines 92 and 93 allow the use of INT0 for an interrupt enabler and INT1 as an interrupt enabler respectively. Line 94 simply defines the priority of INT1 as a low priority input as it has not been previously defined (like it was automatically defined for INT0 as high).

We then move on to define (explicitly) that the rising edges of INT0 and INT1 are to indicate that an interrupt flag has been raised. If we chose to omit the lines 100 and 101 it would have been implicitly defined but we choose to explicitly define so as to not allow for any room for error.

Finally, we clear the flags for INT0 and INT1 allowing the micro-controller to know that we are currently not in INT0 HPISR or INT1 LPISR. As long as those flags are clear the microcontroller knows we are not in an ISR.

You can ignore line 112 for now as it pertains to the logic in LPISR that we have not yet discussed.



```
    ;The following are required to be enabled as per page 91 of manual
    bsf RCON, IPEN           ; Enables interrupt priority feature, stands for Interrupt Priority ENable
    bsf INTCON, GIEH         ; Enables all high-priority input, stands for Global Interrupt Enable High
    bsf INTCON, GIEL         ; Enables all low-priority input, stands for Global Interrupt Enable Low

    ;INT0 is defined as HIGH piority interrupt by default. Let us set up the Low
    ;priority interrupt service routine to be INT1

    bsf PORTB,INT0           ;Allows bit0 or INTerrupt0 to be read (this is going to be HPISR)
    bsf PORTB,INT1           ;Allows bit1 or INTerrupt1 to be read (this is going to be LPISR)

    ;We do not need to set register TRISB bits INT0 and INT1
    ;as that has been already done by AC

    bsf INTCON,INT0IE        ;Enables the use of INT0 stands for INTerrupt 0 Interrupt Enable
    bsf INTCON3, INT1IE      ;Enables the use of INT1 stands for INTerrupt 1 Interrupt Enable
    bcf INTCON3, INT1IP      ;Since INT1 is not defined as default HPISR or LPISR we define it to be LPISR

    ;Now we are going to define the rising edge to be the interrupt trigger
    ;setting    (1) is rising edge
    ;clearing   (0) is falling edge

    bsf INTCON2, INTEDG0     ;INT0 interrupts on rising edge
    bsf INTCON2, INTEDG1     ;INT1 interrupts on falling edge

    ;Now we have to make sure the microcontroller knows INT0 and INT1 interrupts have not yet occured
    ;So we explicitly say that have not clearing the Interrupt Flag

    bcf INTCON, INT0IF       ;Clear INT0 interrupt flag (INT0 has not yet occured)
    bcf INTCON3, INT1IF      ;Clear INT1 interrupt flag (INT1 has not yet occured)

    ;Now we are ready to define our ISR as all the initialization has been done

    ;Defining variables needed for LPISR logic
    MOVLF  2, TIMECOUNT

return
```

Fig. 5. Initial Subroutine Code

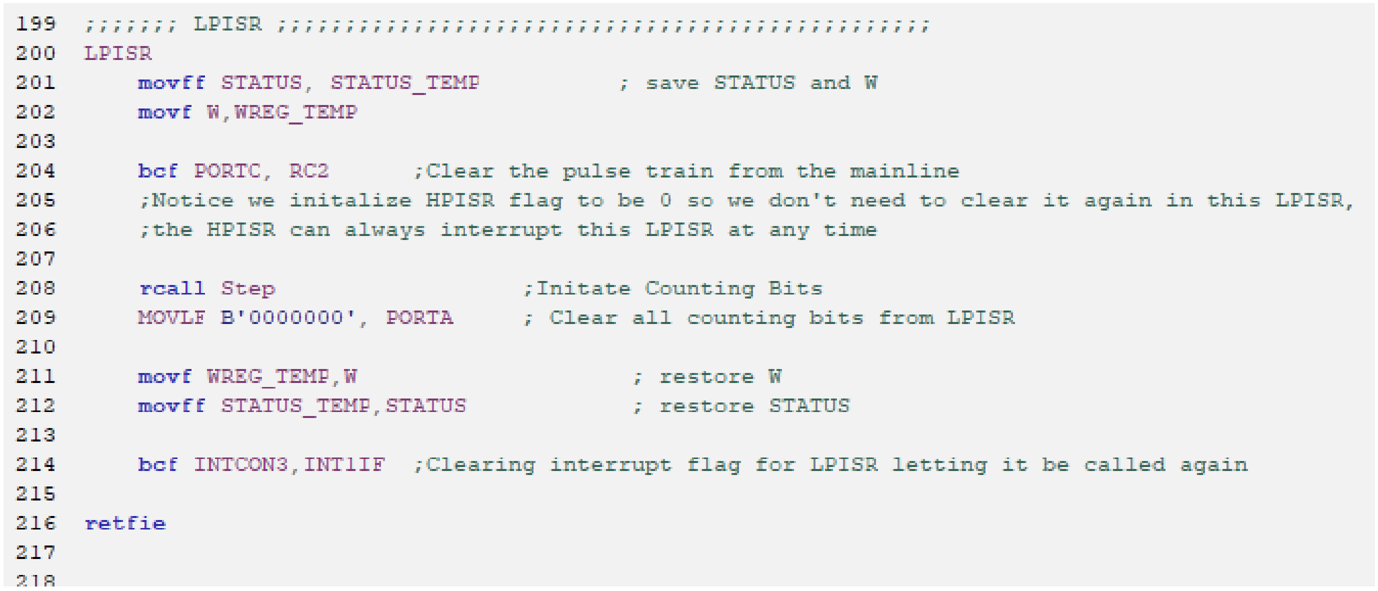
III. Low Priority Interrupt Service Routine (LPISR)

The LPISR's flowchart seen in figure 2 begins by moving the STATUS register and the W (working) register into temporary (shadow registers). This is to preserve the original value in the two registers since they might be altered in the subroutine. When we are about the exit this subroutine we will restore the values of the registers to what they were before we began the ISR. The next block is us clearing RC2 so as to clear the pulse train generated by the mainline. So far no logic has been designed or used. The next block is $\boxed{\textit{rcall Step}}$ and we have not yet seen the step subroutine but just keep it in mind as producing a sequence of values in RA1, RA2, and RA3. Thus after we have produced the sequence of values in RA1, RA2, and RA3 that we desired we exit the Step subroutine. This essentially marks the end of the LPISR and we began to make our departing changes. We clear all the values of PORTA, essentially clearing RA1, RA2, and RA3 to clearly mark the end of the desired sequence and we move back the original values from the temp (shadow) working and STATUS register into the real working and STATUS register. We clear the INT1 interrupt flag allowing the LPISR to be initiated again if desired and to clear the values of any LPISR that may have been triggered during the LPISR. Lastly we return from interrupt. The code implementation of this flowchart can be seen in figure 6.

So far the only logic in the flowchart is contained in the subroutine of Step. The Step subroutine is supposed to produce the following sequence of values in the bits RA1, RA2, and RA3 (see table I).

As is quite evident in table I we are simply counting backwards from 7 to 0 in binary. Counting from 0 to 7 is the same as 7 to 0, it is simply that the initial states of each bit is reversed, the logic is still the same. The logic is represented in pseudocode in listing 1.

We see in listing 1 that we would like to count up to 8 so we set maxCount to 8. This could be 100 if we would like to continue the cycle of the same sequences a 100 times. In this example it is 8 because we wish to see only the unique binary representations of numbers with three bits. We set the two counters that we will be using in lines 4 and 5 to be 0 so that the



```
;;;;;;; LPISR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
LPISR
    movff STATUS, STATUS_TEMP          ; save STATUS and W
    movf W,WREG_TEMP

    bcf PORTC, RC2      ;Clear the pulse train from the mainline
    ;Notice we initalize HPISR flag to be 0 so we don't need to clear it again in this LPISR,
    ;the HPISR can always interrupt this LPISR at any time

    rcall Step                  ;Initate Counting Bits
    MOVLF B'0000000', PORTA     ; Clear all counting bits from LPISR

    movf WREG_TEMP,W                    ; restore W
    movff STATUS_TEMP,STATUS            ; restore STATUS

    bcf INTCON3,INT1IF  ;Clearing interrupt flag for LPISR letting it be called again

retfie
```

Fig. 6. LPISR code

TABLE I
SEQUENCE OF VALUES THAT THE STEP SUBROUTINE MUST DISPLAY IN BITS RA1, RA2, AND RA3

Step	RA3	RA2	RA1
1	1	1	1
2	1	1	0
3	1	0	1
4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1
8	0	0	0

counter starts at 0. Lines 8-10 in the listing are the initial values from RA1, RA2, and RA3 or of the three bits of interest. If we would like to count forward from 0 to 7 we set the inital values to 000 instead of 111. The while loop goes on for as many iterations as we define the maxCount to be. In every iteration we toggle the least significant bit (LSB) and we increase the counter for bit0 toggles in line 15. Then we check to see if an even number of bit0 toggles have taken place by the if statement in line 16, if so then we know it is time to toggle the 2nd LSB and if there have been an even number of togglings of bit1 then we can toggle the 3rd LSB or bit2 and so on. This is the fundamental way of binary counting that computers use. Let us run through a few steps using the flowchart implementation of this listing in figure 1.

We start with initial values for bit2, bit1, and bit0 to be 111 respectively. Further, the counters for the number of toggles for bit0 and bit1 are 0. We check at first the numberOfBit0Toggles <8 since 0<8 and since this evaluates to true we toggle bit0 to obtain 110. Then we delay for 0.2 ms (ignore the delays for now as they are part of the spec sheet but not important in the logic of counting). Since we toggled bit0 we increase numberOfBit0Toggles by 1. The new value of numberOfBit0Toggles is 1 as it was initially 0. We then check to see if numberOfBit0Toggles modulo 2 evaluates to 0. 1 modulo 2 is 1. This is false so we go back to start.

We check to see if the numberOfBit0Toggles <8 and it evaluates to true since 1 <8. We then toggle bit0 to get 111 from

```
int maxCount = 8; //Cardinality of the set of sequences you would like to go up to

//counter of toggles always starts at 0
int numberOfBit0Toggles = 0;
int numberOfBit1Toggles = 0;

//If we are counting from 0 to 7 instead set these to be 0
bool int0 = 1;  //Similar to RA1
bool int1 = 1;  //Similar to RA2
bool int2 = 1;  //Similar to RA3

while(numberOfBit0Toggles < maxCount)
{
    toggle bit0;    //The least significant bit (LSB) always toggles in every iteration
    numberOfBit0Toggles++;
    if(numberOfBit0Toggles modulo 2 == 0)
    {
        toggle bit1;    //The second LSB toggles every other toggling of LSB
        numberOfBit1Toggles++;
        if(numberOfBit1Toggles modulo 2 == 0)
        {
            toggle bit2;     //The third LSB toggles every other toggling of second LSB
        }
    }
}
```

Listing 1: Pseudocode method to count from 7 to 0

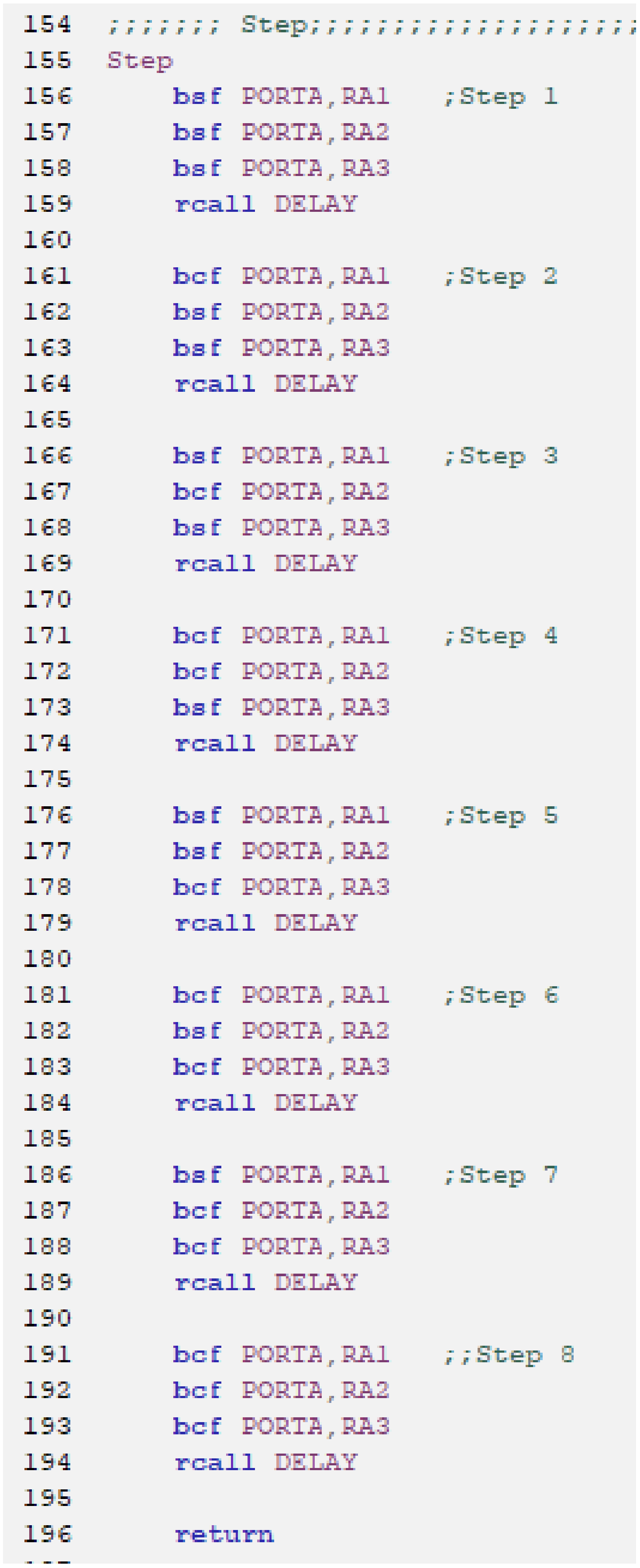
previously 110. We then increase numberOfBit0Toggles from 1 to 2. Now we check to see if numberOfBit0Toggles modulo 2 is 0. 2 modulo 2 is indeed 0. This statement evaluates to true so we can toggle bit1. We now have 101. We increase the numberOfBit1Toggles from 0 to 1. We have 100 We check to see if numberOfBit1Toggles modulo 2 is 0 and since 1 modulo 2 is not zero we take the FALSE route and go back to start.

We check to see that the numberOfBit0Toggles <8 and since 2<8 this is true so we continue. We increment numberOfBit0Toggles so now numberOfBit0Toggles is 3. We check to see if numberOfBit0Toggles modulo 2 is 0 and since 3 modulo 2 is 1 we see that this is false and we go back to start.

We check to see that numberOfBit0Toggles <8 and since 3<8 this is true. We toggle bit0 to go from 100 to 101 and we increment numberOfBit0Toggles from 3 to 4. We check to see if numberOfBit0Toggles modulo 2 is 0 and 4 mod 2 is indeed 0 so we can toggle bit1 to go from 101 to 111. We increment numberOfBit1Toggles from 1 to 2. We check to see if numberOfBit1Toggles modulo 2 is 0 and since 2 modulo 2 is 0 we go down the TRUE path. We toggle bit2 so we go from 111 to 011. We then loop back to the beginning.

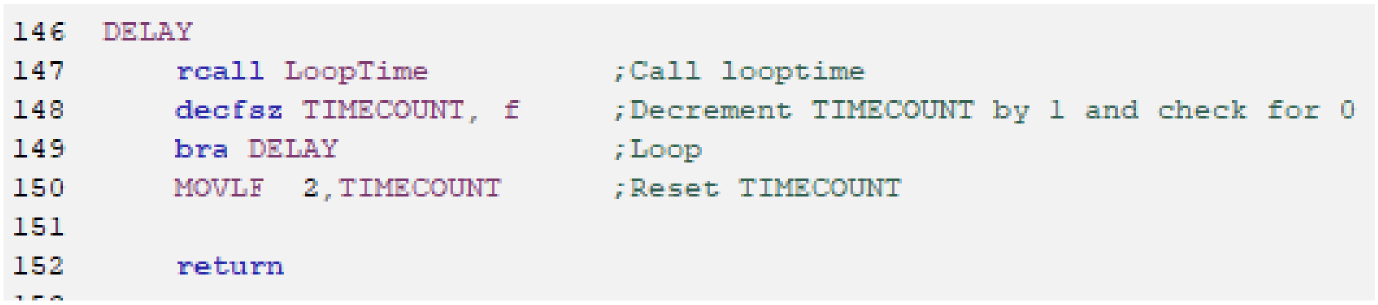
Since the author has procrasitnated doing this assignment (for which there was more than enough time to complete) he does not have enough time to implement the solution in a sophisticated manner so you will have to bear with the hamfisted attempt in figure 7. The author feels the need to apologize in advance. The logic behind the implementation in figure 7 is still the same as that of listing 1, but it is less generalized. You will notice in the step subroutine there is yet another subroutine of DELAY essentially makes the microcontroller pause for 0.2ms and preserve the previous state. The code definition for this can be seen in figure 8. The logic is the same as the one we have covered in experiment 2 where we decrement TIMECOUNT (which is set to 2 in the initialized in line 112 in figure 5) until it hits zero, after which a zero register is raised which skips line 149 to reset TIMECOUNT by moving 2 back into it and returning.

Examples of the delay subroutine can be seen in figure 9 where we see another LPISR signal cannot interrupt an ISR that is already in place, and when the initial LPISR has finished it returns to mainline. The other graph in figure 10 shows that when INT1 is set to HIGH and LPISR is underway we can interrupt it with a HPISR (as is done in 2100). Notice an RE2 signal cannot interrupt a LPISR but it will interrupt HPISR because that is how we define this specific HPISR. The HPISR cannot be interrupted by another HPISR or LPISR but only ended by RE2 being set. When HPISR is finished (recall HPISR was triggered during LPISR) we go back to finishing our LPISR. After we finish the LPISR we go back to the mainline.



```
;;;;;;; Step;;;;;;;;;;;;;;;;;;;
Step
        bsf PORTA,RA1    ;Step 1
        bsf PORTA,RA2
        bsf PORTA,RA3
        rcall DELAY

        bcf PORTA,RA1    ;Step 2
        bsf PORTA,RA2
        bsf PORTA,RA3
        rcall DELAY

        bsf PORTA,RA1    ;Step 3
        bcf PORTA,RA2
        bsf PORTA,RA3
        rcall DELAY

        bcf PORTA,RA1    ;Step 4
        bcf PORTA,RA2
        bsf PORTA,RA3
        rcall DELAY

        bsf PORTA,RA1    ;Step 5
        bsf PORTA,RA2
        bcf PORTA,RA3
        rcall DELAY

        bcf PORTA,RA1    ;Step 6
        bsf PORTA,RA2
        bcf PORTA,RA3
        rcall DELAY

        bsf PORTA,RA1    ;Step 7
        bcf PORTA,RA2
        bcf PORTA,RA3
        rcall DELAY

        bcf PORTA,RA1    ;;Step 8
        bcf PORTA,RA2
        bcf PORTA,RA3
        rcall DELAY

        return
```

Fig. 7. (Poorly Implemented) Step Subroutine Code



```
DELAY
    rcall LoopTime              ;Call looptime
    decfsz TIMECOUNT, f         ;Decrement TIMECOUNT by 1 and check for 0
    bra DELAY                   ;Loop
    MOVLF   2,TIMECOUNT         ;Reset TIMECOUNT

    return
```

Fig. 8. Code for the delay subroutine

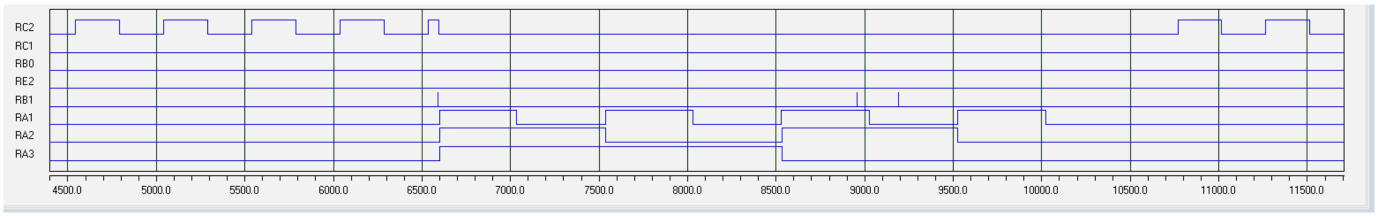


Fig. 9. LPISR with LPISR signals in between the interrupt

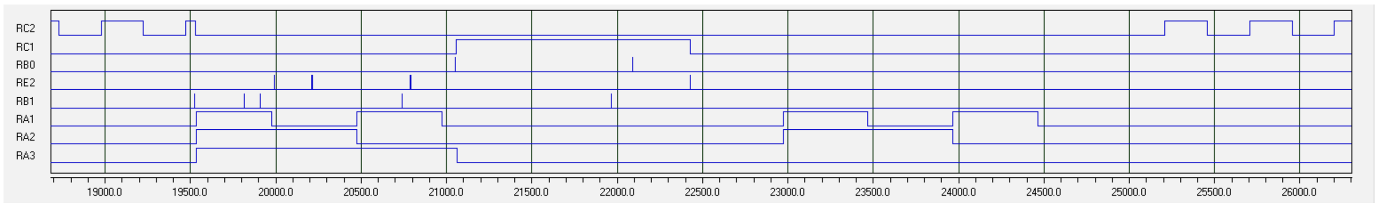


Fig. 10. LPISR with both LPISR and HPISR signals and RE3 being triggered

IV. High Priority Interrupt Service Routine (HPISR)

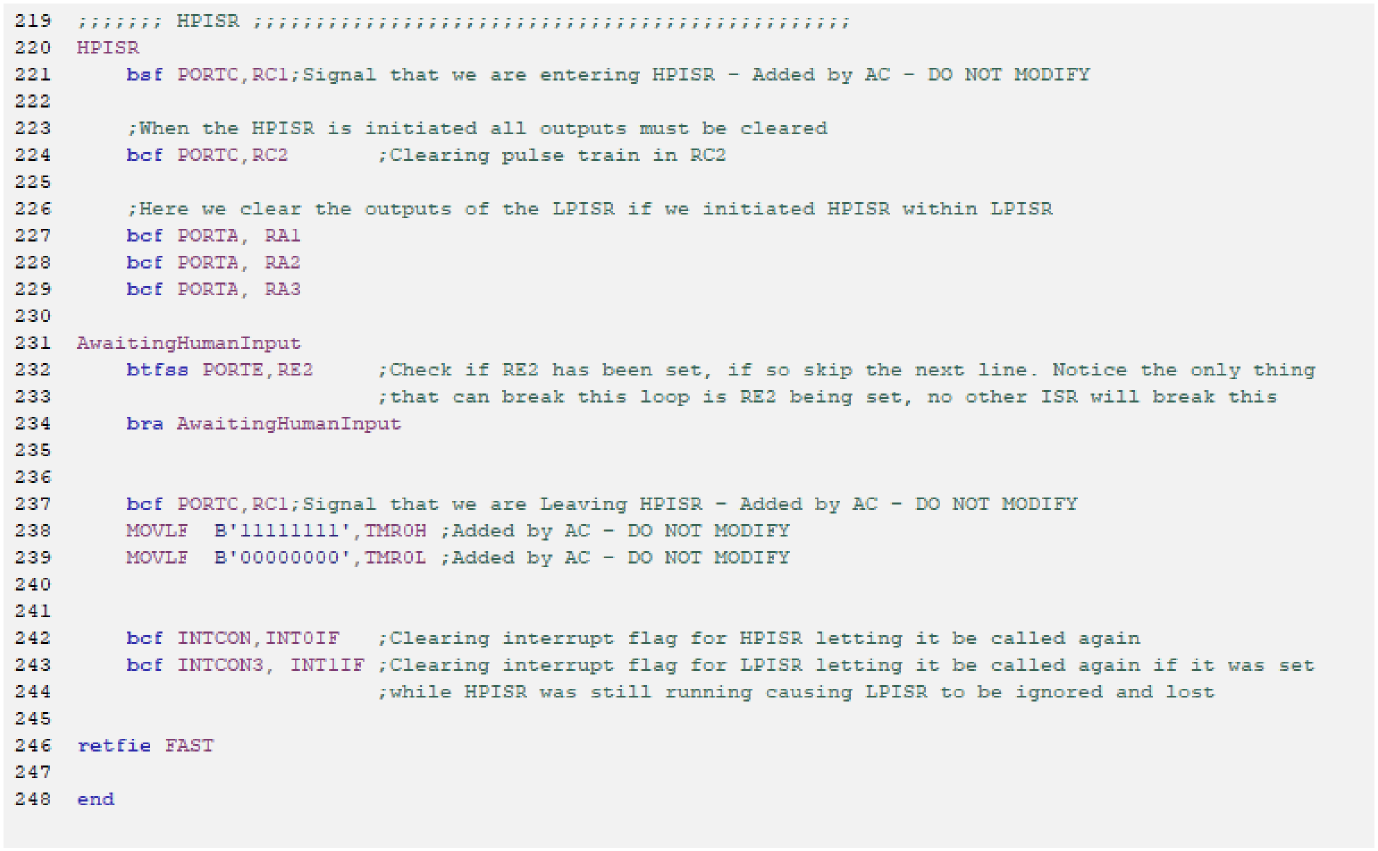
Let us refer to the flowchart in figure 3. We begin the HPISR by setting RC1 in PORTC. As long as we remain in HPISR we will have RC1 raised as HIGH. Then our next step is to clear RC2 which records the pulse train from the mainline program. The next 3 instructions are to clear bits used in LPISR, that is RA1, RA2, and RA3. This exists in case we have called HPISR during LPISR and the values of bits RA1, RA2, and RA3 is not LOW like it should be in the mainline.

So far we have only done what was asked by the specification sheet. We have not yet implemented any logic on our own. Another condition in the spec sheet is that the HPISR can only be interrupted if bit RE2 was raised HIGH by a human input. Thus if we have triggered HPISR we cannot exit it without a human input. The logic here is extremely simple and can be thought of as an empty while loop with the condition being $\boxed{\textit{while(RE2 == 0)}}$. As long as RE2 is 0 we loop back to the while loop (being stuck in an infinite loop) and we only break out when RE2 is set by a human. The assembly version of this empty while loop would be $\boxed{\textit{btfss PORTE, RE2}}$ which essentially checks bit 2 of PORTE and if the bit is set then we skip the next instruction which would be to branch back into the loop. This can be seen more clearly by the code implementation of the flowchart that can be seen in figure 11. As you can see in figure 11 in line 232 if we have set RE2 then we automatically skip line 234 and rbeak out of the infinite loop.

The next step int he flowchart is to clear RC1 which indicates we are leaving the HPISR. We indicate we entered HPISR by setting RC1 and we indicate our departure be clearing RC1. Finally we clear the interrupt flags of both INT0 and INT1 allowing for the calling of LPISR or HPISR since we have come to the end of HPISR. It is interesting to note we clear interrupt flag of LPISR incase we have called LPISR during HPISR but as we know, any and all ISR during a HPISR are ignored.

Line 246 in figure 11 is $\boxed{\textit{retfie FAST}}$. Retfie is simply return from interrupt. The FAST denotes that we restore the original values of the working (W) register and the STATUS register from the shadow registers or the temporary registers.

Using the logic analyzer we can see the HPISR in figure 12. You can see we set RB0/INT0 at 15300 and we can be seen entering HPISR since RC1 is set HIGH. We exit HPISR at around 16500 when RE2 is set HIGH and we see that RC1 is set to LOW. The second time we SET RB0/INT0 at 17280 you can see all further INT0/RB0 and INT1/RB1 signals are ignored and it is only when RE2 is SET at 18200 do we exit HPISR.



```
219 ;;;;;;; HPISR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
220 HPISR
221     bsf PORTC,RC1;Signal that we are entering HPISR - Added by AC - DO NOT MODIFY
222 
223     ;When the HPISR is initiated all outputs must be cleared
224     bcf PORTC,RC2       ;Clearing pulse train in RC2
225 
226     ;Here we clear the outputs of the LPISR if we initiated HPISR within LPISR
227     bcf PORTA, RA1
228     bcf PORTA, RA2
229     bcf PORTA, RA3
230 
231 AwaitingHumanInput
232     btfss PORTE,RE2     ;Check if RE2 has been set, if so skip the next line. Notice the only thing
233                         ;that can break this loop is RE2 being set, no other ISR will break this
234     bra AwaitingHumanInput
235 
236 
237     bcf PORTC,RC1;Signal that we are Leaving HPISR - Added by AC - DO NOT MODIFY
238     MOVLF  B'11111111',TMR0H ;Added by AC - DO NOT MODIFY
239     MOVLF  B'00000000',TMR0L ;Added by AC - DO NOT MODIFY
240 
241 
242     bcf INTCON,INT0IF   ;Clearing interrupt flag for HPISR letting it be called again
243     bcf INTCON3, INT1IF ;Clearing interrupt flag for LPISR letting it be called again if it was set
244                         ;while HPISR was still running causing LPISR to be ignored and lost
245 
246 retfie FAST
247 
248 end
```

Fig. 11. Code implementation of flowchart in figure 3

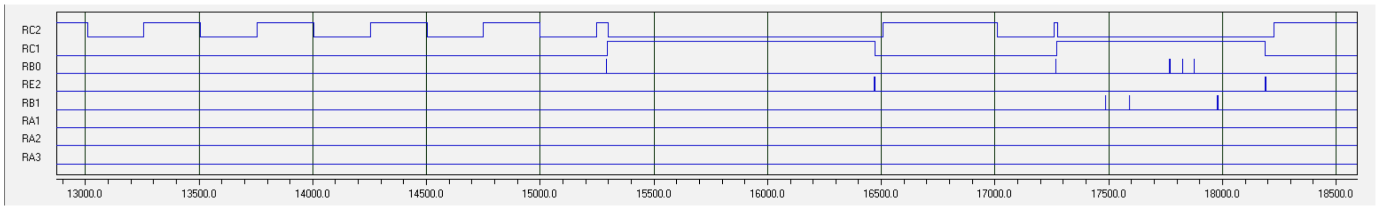


Fig. 12. HPISR output on logic analyzer