CONTENTS

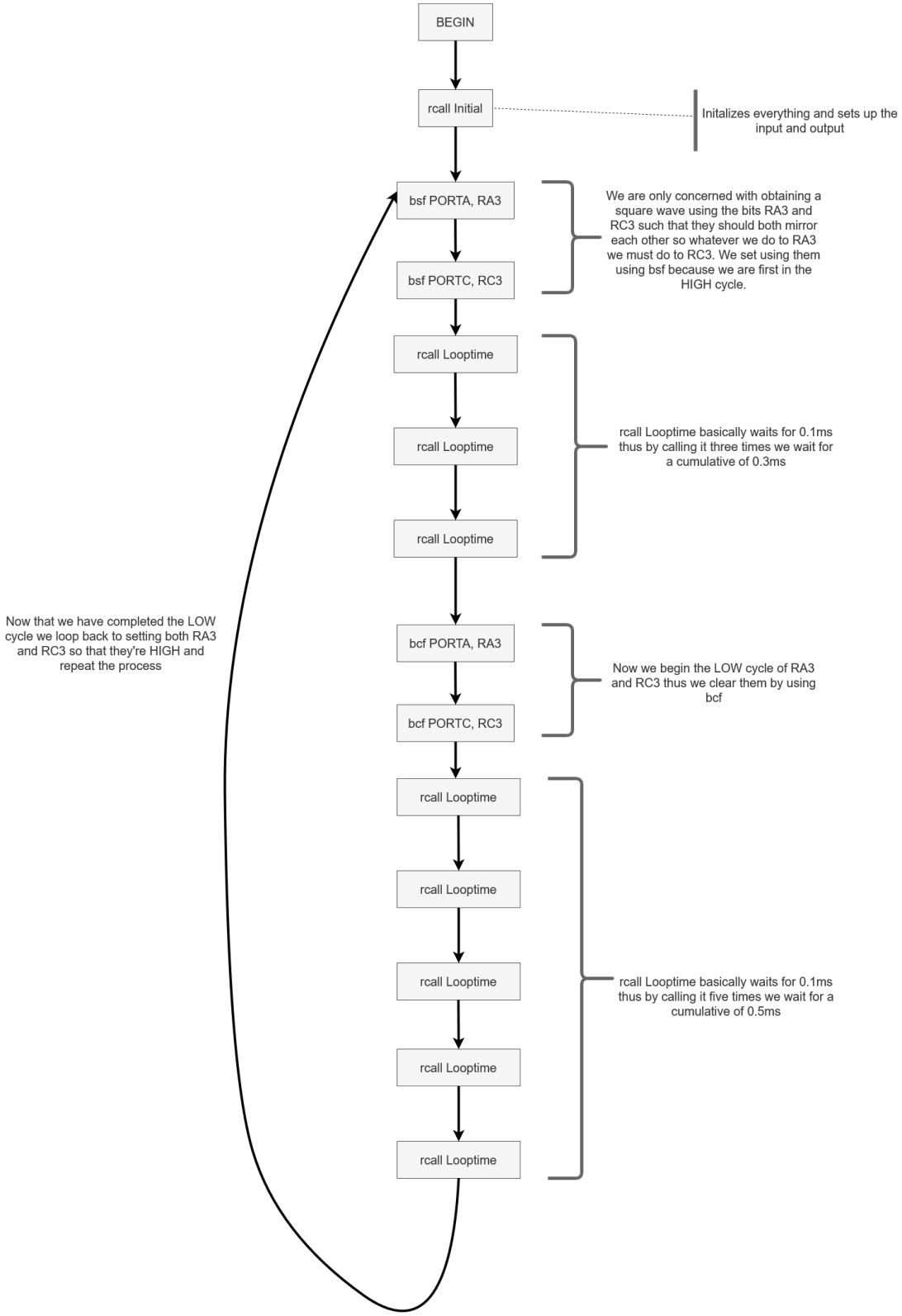

Fig. 1. Naive implementation of a pulse train without reference timer

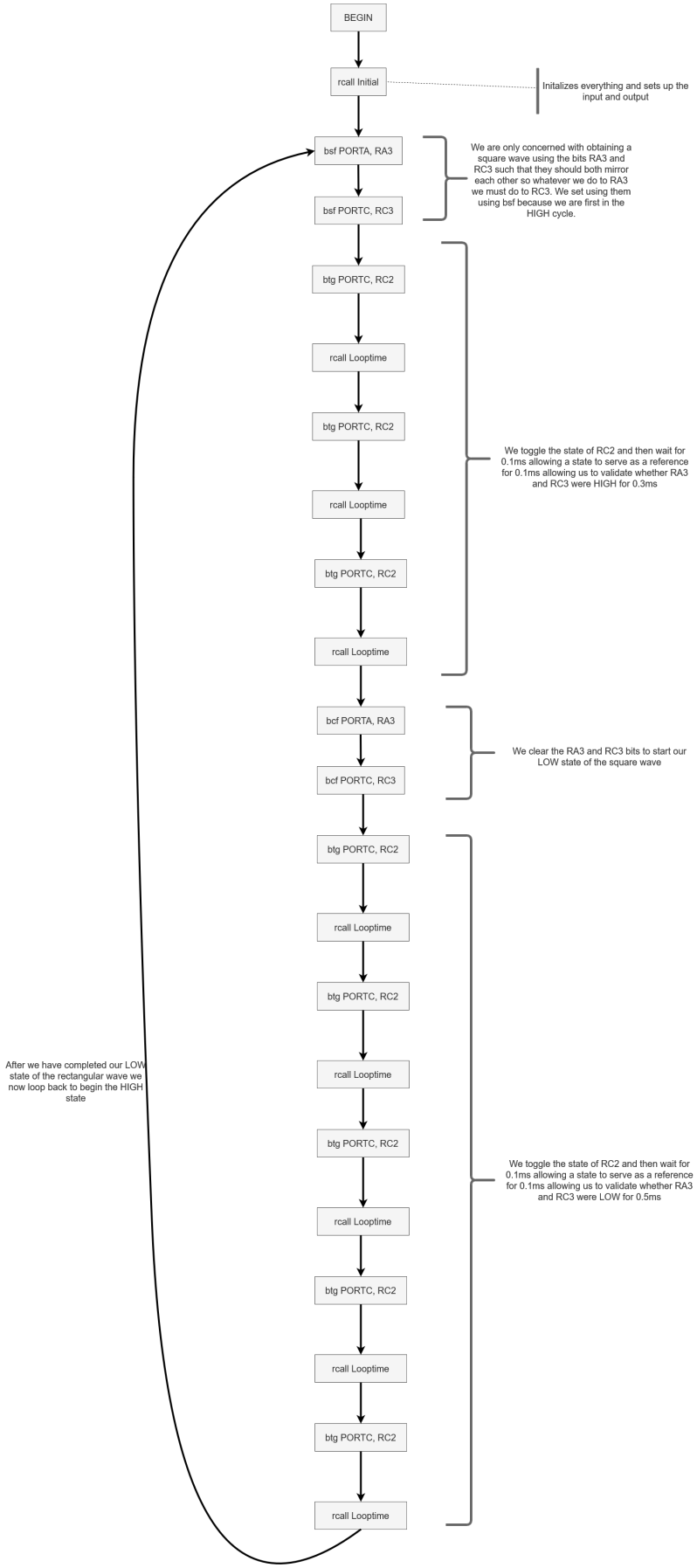

Fig. 2. Naive implementation of pulse train with reference timer

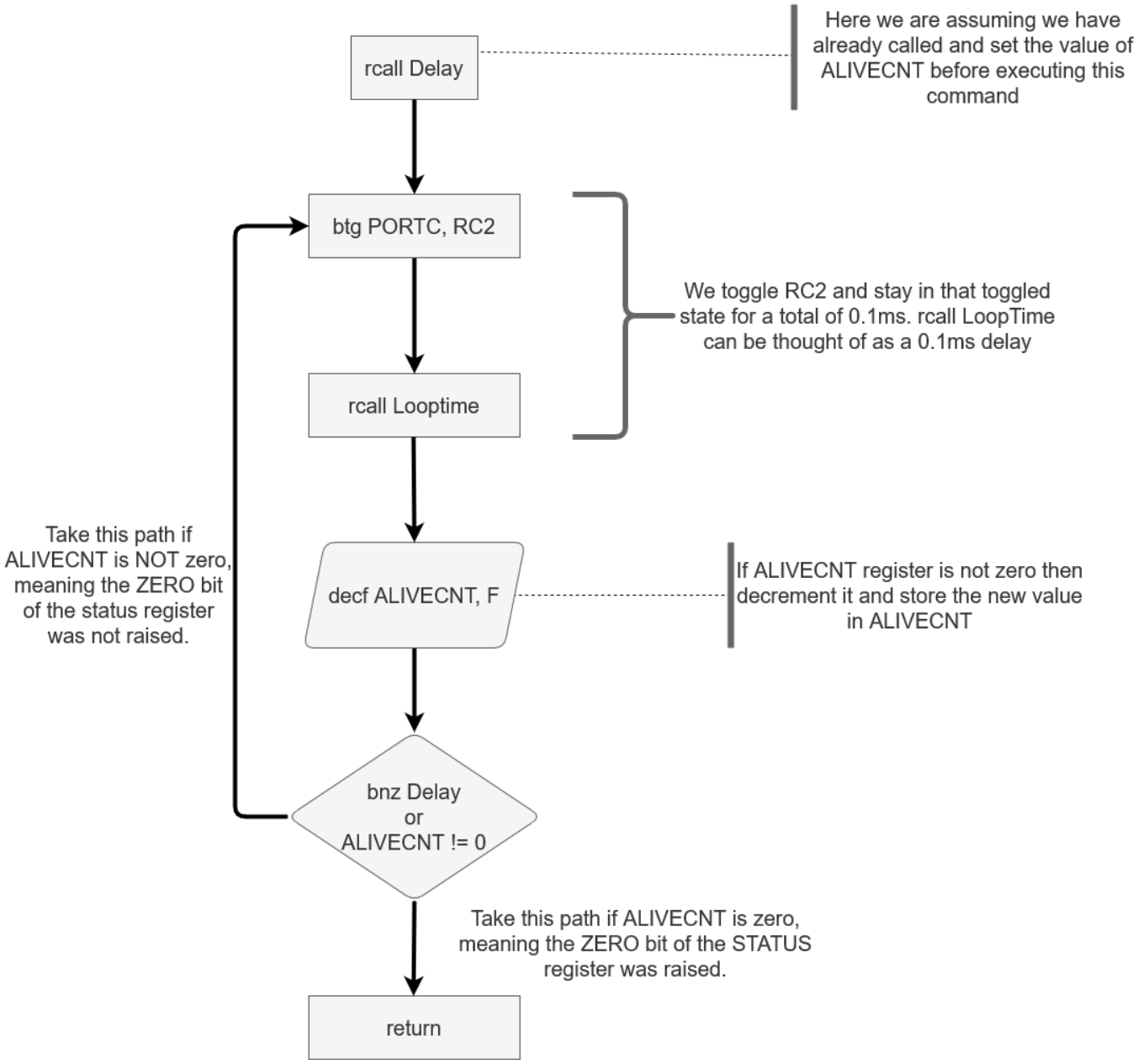

Fig. 3. Delay subroutine flowchart

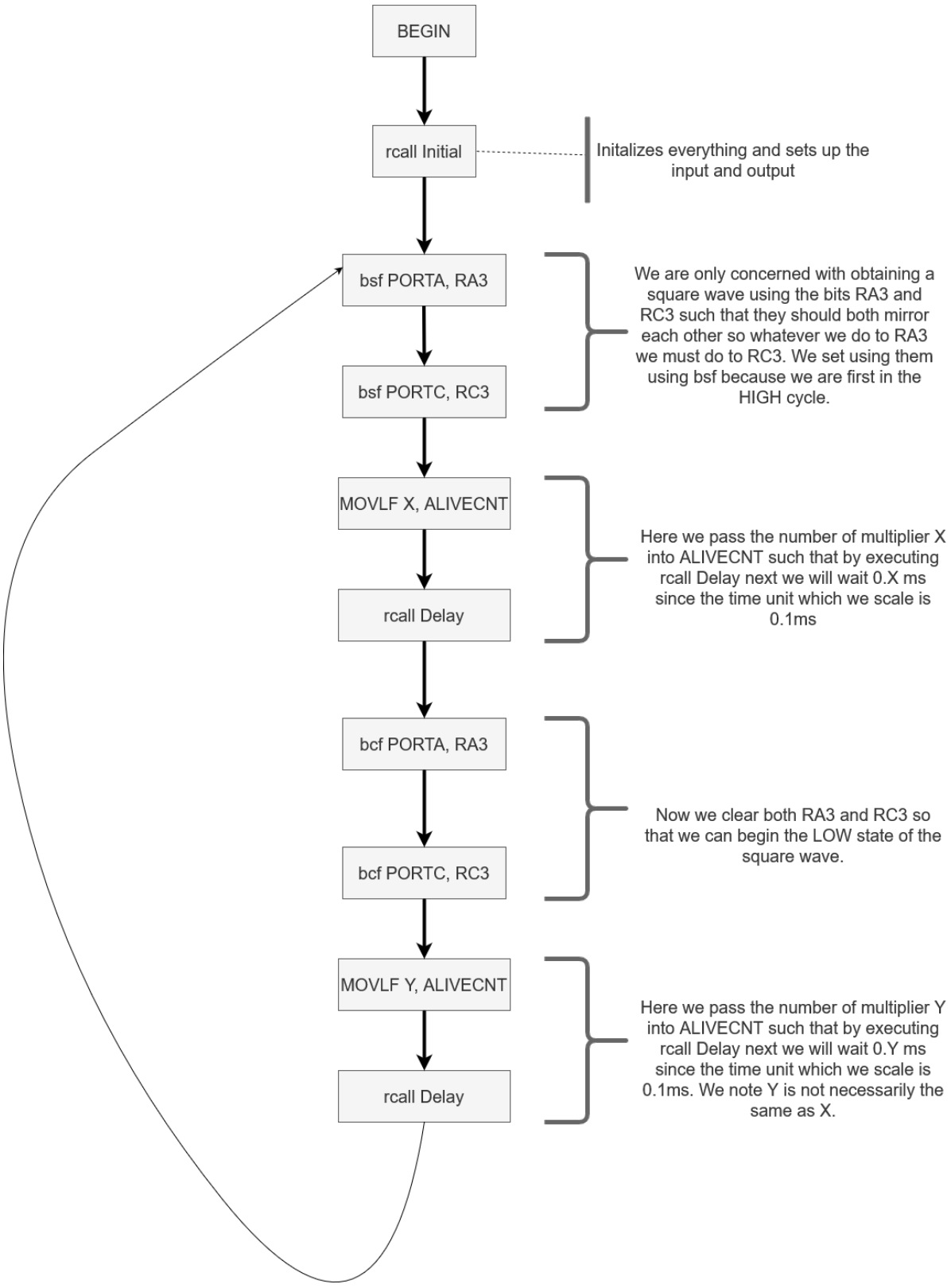

Fig. 4. The mainline program flowchart that can generate any square wave.

Exp 2 - Pulse Train Generator

Anas Ashraf

I. The Naive Approach to a Pulse Train Generator

A. Problem Outline

The problem presented is to create a pulse train at the RA3 bit of PORTA and RC3 bit of PORTC for a desired duty cycle, wherein the duty cycle is defined:

$$Duty\ Cycle = \frac{T_{HIGH}}{T_{LOW} + T_{HIGH}} \times 100\% \tag{1}$$

We are presented four different cases, each with different T_{HIGH} and T_{LOW} but for now let us work with abstractions as we are in the early phase of our code design. Before we begin with the code design it is important to consider the resources that are available to us already from the template that we are building upon. We see that the mainline program in figure 5 already has a delay in line 60 which is *rcall Looptime*. This can be thought of as an elementary delay element which allows for a delay of 0.1ms which we shall from now on consider as one time unit or T_{unit}. For the rest of this paper $T_{unit} = 0.1ms$

```
53  ;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
54
55  Mainline
56          rcall  Initial          ;Initialize everything
57  Loop
58          btg  PORTC,RC2          ;Toggle pin, to support measuring loop time
59          rcall  BlinkAlive       ;Blink "Alive" LED
60          rcall  LoopTime         ;Make looptime be 0.1 milliseconds
61
62          bra  Loop
```

Fig. 5. Mainline program for P_0 template

We do not need to read about the definition of the subroutine for Looptime, it is sufficent to know that calling it will cause a 0.1ms delay. Notice, that we do not need lines 58 and 59 as of now so we are safe in commenting them out. Now that we have an elementary delay by invoking *rcall LoopTime* we can set and clear bits RA3 and RC3 of registers PORTA and PORTC respectively. Let us solve for the most simplest of cases where:

$$T_{LOW} \neq T_{HIGH} \tag{2}$$

$$T_{LOW},\ T_{HIGH} \geq T_{unit} \tag{3}$$

$$T_{unit} \mid T_{HIGH} \tag{4}$$

$$T_{unit} \mid T_{LOW} \tag{5}$$

The properties of the problem are outlined in equations 2-5. Now that we have the conditions of the problem defined we can begin working on a possible solution with some numbers plugged into the general problem.

B. A Naive Implementation

Let us choose to solve for the hardest case provided to us where $T_{HIGH} = 0.3ms$ and $T_{LOW} = 0.5ms$ and design a solution or possible implementation around that. It should be noted that we have the general logic followed by figure 1. We believe it is important we go over the flowchart before we go over the code and implementation of said flowchart. We begin with *rcall Intial* this is also present in figure 5 and its purpose is simply to initialize all the PINS for input and output. Now after we have initialized everything we would like to declare bits RA3 and RC3 of PORTA and PORTC respectively to be HIGH and keep them HIGH for 0.3ms as per the problem. Thus we would like the microcontroller to do nothing for 0.3ms. Recall that by invoking *rcall Looptime* we can delay for an elementary unit of time which is 0.1ms. Thus, if we invoke this subroutine 3 times we have a successful stall of 0.3ms. Now look back at flowchart in figure 1.Notice after setting bits RA3 and RC3 in PORTA and PORTC respectively by executing *bsf PORTA, RA3* and then *bsf PORTC, RC3* we then cause a delay to occur by invoking *rcall Looptime* subroutine three times. Thus, after setting both the desired bits to HIGH we wait

exactly for 0.3ms. We can see this in the code in figure 6 where in line 56 we *rcall Initial* and then set the bits in line 61 and 62 and finally invoke *rcall Looptime* three times waiting for a cumulative of 0.3 ms.

After 0.3 ms have elapsed and the criterion for $T_{HIGH} = 0.3ms$ has been fulfilled it is time to move on to the LOW state of the rectangular square wave. Now return back to the flowchart in figure 1 and see that we clear RA3 and RC3 so that we may begin the LOW state of the rectangular square wave. Then we invoke a delay of 0.5ms by invoking *rcall Looptime* five times. Then, to repeat everything we loop back to setting RA3 and RC3, this can been seen in line 85 of figure 6. Now we have satisfied the criteria for $T_{HIGH} = 0.3ms$ and $T_{LOW} = 0.5ms$ and we can see the output waveform in figure 7. Notice in the waveform the arbitrary units on the bottom don't help us discern time. We can clearly see that RA3 and RC3 mirror each other and the HIGH to LOW ratio is clearly 3 time units HIGH and 5 time units LOW but we don't know the time units without a reference wave. For all we know we could have anywhere between hours to nanoseconds for time units. Thus it is important to have a reference wave with which we can compare our results.

```
53  ;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
54
55  Mainline
56          rcall  Initial          ;Initialize everything
57
58
59  Loop
60  ;First we do the HIGH cycle so we set both bits to HIGH
61      bsf PORTA, RA3
62      bsf PORTC, RC3
63
64
65  ;Now we wait for 0.3ms as per problem (D)
66      rcall LoopTime      ;wait 0.1 ms
67      rcall LoopTime      ;wait 0.1 ms
68      rcall LoopTime      ;wait 0.1 ms
69
70
71  ;After 0.3 ms HIGH have elapsed we turn off both bits
72  ;and switch to our LOW cycle
73      bcf PORTA, RA3
74      bcf PORTC, RC3
75
76
77  ;Now we wait for 0.5ms as per problem (D) so we have
78      rcall LoopTime      ;wait 0.1 ms
79      rcall LoopTime      ;wait 0.1 ms
80      rcall LoopTime      ;wait 0.1 ms
81      rcall LoopTime      ;wait 0.1 ms
82      rcall LoopTime      ;wait 0.1 ms
83
84
85      bra Loop    ;repeat ad infinitum
```

Fig. 6. The naive implementation

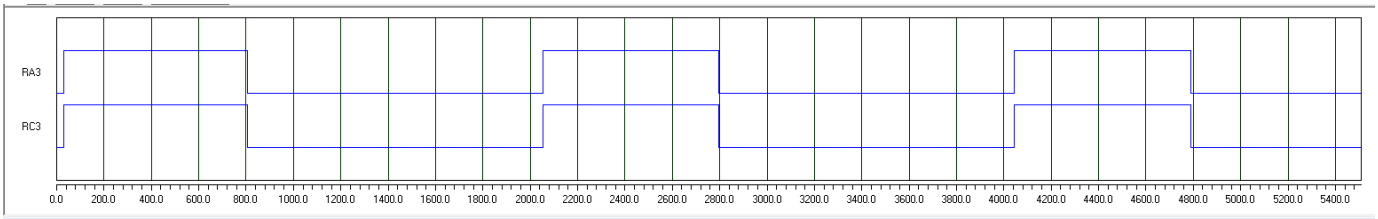

Fig. 7. Naive solution waveform for RA3 and RC3 by executing code in figure 6.

C. An Improved Albeit Naive Solution

To serve as a reference square wave let us designate RC2 of PORTC to toggle every 0.1ms. This can be easily accomplished by taking 6 and adding *btg PORTC, RC2* above/below every line that invokes the subroutine *rcall LoopTime*. It does not matter whether we toggle the bit RC2 before or after the delay, so long as we are consistent with our placement of the command *btg PORTC, RC2* it is sufficient to achieve our desired result. You will notice in flowchart 2 we have simply copied the

blocks in figure 1 and chosen to pair every rectangle that invokes *rcall LoopTime* with *btg PORTC, RC2*. It is quite easy to tell by flowchart in figure 2 that we have not fundamentally changed the logic but simply added an extra block before every delay thus to explain this would be redundant. The implementation of this flowchart can be seen in figure 9. Notice, again by comparing code in figure 6 and 9 that the only thing that has changed is that we complemented every invocation of the subroutine *rcall LoopTime* with *btg PORTC, RC3*. The fundamental logic has still stayed the same. The output we receive from animating the code in figure 9 is shown in figure 8. Notice in the beginning there is some transient response which we can safely ignore however, as the simulation progresses we see that RC3 and RA3 are both HIGH for 3 state toggles of RC2 wherein each state is held in place for 0.1ms thus confirming that we $T_{HIGH} = 0.3ms$. Further, we see that RC3 and RA3 are both LOW for five state toggles of RC2 (wherein each state lasts for only 0.1ms) thereby confirming we have $T_{LOW} = 0.5ms$.

Notice a huge flaw in this approach, if we wish to have a LOW state for 100 seconds and we had the $T_{unit} = 0.1\mu s$ we would have to copy and paste the code to toggle RC2 and then invoke the delay for $0.1\mu s$ a total of 2 billion times (1 billion for *btg PORTC, RC2* and the other billion for *rcall LoopTime*. That is only for the LOW state and not mentioning the HIGH state of the two bits. Thus we need to implement a logical algorithm where we don't have to manually copy and paste the code billions of times.

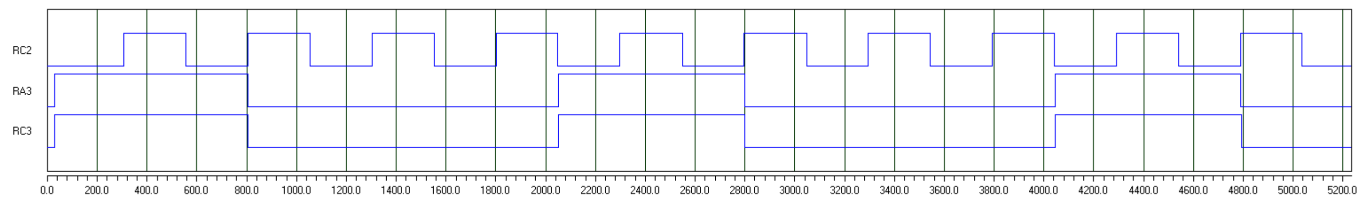

Fig. 8. The final desired waveform with $T_{HIGH} = 0.3ms$ and $T_{LOW} = 0.5ms$ and a reference bit RC2 having each state be 0.1ms

```
;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Mainline
        rcall  Initial          ;Initialize everything


Loop
;First we do the HIGH cycle so we set both bits to HIGH
    bsf PORTA, RA3
    bsf PORTC, RC3


;Now we wait for 0.3ms as per problem (D)
;We also include RC2 now to serve as a reference timer
;where each HIGH or LOW state for RC2 means 0.1ms have
;elapsed


    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms



;After 0.3 ms HIGH have elapsed we turn off both bits
;and switch to our LOW cycle

    bcf PORTA, RA3
    bcf PORTC, RC3


;Now we wait for 0.5ms as per problem (D) so we have

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms

    btg PORTC, RC2
    rcall LoopTime      ;wait 0.1 ms


    bra Loop    ;repeat ad infinitum
```

Fig. 9. Naive implementation with RC2 as reference square wave of 0.1ms

II. A Competent Solution

A. The Delay Subroutine

We can use figure 8 as an inspiration to create a much smarter way of doing things. We know that we repeat the two instructions *btg PORTC, RC2* and *rcall LoopTime* many times during the code and we repeat the instructions N number of times. We can let the number of times we repeat the two instructions be represented by some counter. A pseudo-code representation of this idea is seen in listing 1.

```
1  def Delay(int Counter)  #Here we pass the number of time units we would like to repeat the delay
2  {
3      while(Counter != 0)
4      {
5          toggle(Register = PORTC, bit=RC2);
6          sleep(0.1ms);       #this can be thought of as rcall LoopTime
7          Counter = Counter - 1;
8      }
9      return;
10 }
```

Listing 1: Pseudocode representation for a variable delay cycle that can be invoked during HIGH or LOW state.

Here we define the Delay subroutine. We pass an argument into the subroutine *int Counter* which counts the number of time units (0.1ms) we would like the microcontroller to stall. So for instance if we would like the microcontroller to stall for 0.5ms we would pass the argument *Counter = 5*. A simple equation to find out the integer you should pass into the function can be seen in equation 6.

$$Counter = \frac{Desired\ time\ to\ stall\ (in\ ms)}{T_{unit}} \tag{6}$$

Recall we define T_{unit} to be 0.1ms so for all intents and purposes the denominator is 0.1ms. Next in listing 1 we see the while loop in line 3 which is very easy to implement in assembly and is easier than any other loop to implement. We see that as long as the Counter is not zero we should toggle RC2 and sleep for 0.1ms which means to stall the microcontroller for one time unit. Then at the end we decrement the Counter so we don't get stuck in an infinite loop. Notice, even if we pass the argument that Counter is zero there will be no error in code execution, the only error we will run into is if we pass a non-positive integer we will never hit zero and thus be stuck in an infinite loop but passing a signed integer in assembly is rather explicit and we define our registers in Assembly that we will pass to be unsigned.

Now that we have begun thinking about the pseudocode we have to replace the lines with reasonable assembly analoges. The function definition will be the same as a subroutine definition where we can define the Delay subroutine as shown in figure 10.

```
148  ;;;;;Delay SUBROUTINE;;;;;;;;;;;;;;;;;
149  Delay
150      btg PORTC, RC2      ;toggle RC2 so as to reflect a time unit
151      rcall LoopTime      ;wait 0.1 ms
152      decf ALIVECNT, F    ;decrement loop counter and return if not zero
153      bnz Delay           ; if ALIVECNT not zero then repeat this subroutine otherwise end the delay subroutine
154      return
```

Fig. 10. Delay subroutine: a translation of pseudo-code in listing 1

Notice in the subroutine shown in figure 10 we see that we first pair together *btg PORTC, RC2* and *rcall LoopTime* where the latter can be thought of as a 0.1ms delay. Now a problem we ran into is that we do not know how to explicitly pass an argument to a subroutine, what we can do is notice that since we are not using *rcall BlinkAlive* subroutine in the mainline program in figure 5, we have a free register leftover. Thus, we use that register as a COUNTER. Based on the value we assign to ALIVECNT before calling the delay subroutine, that is how long we will stall the microcontroller. Notice, in line 152 of figure 10 we see *decf ALIVECNT, F* which is analagous to line 7 in listing 1, where we decrement the counter so as to not be stuck in an infinite loop. Further, the F means we store the new decremented value back into ALIVECNT register. Finally we see *bnz Delay* which checks if the STATUS register has the bit 2 (ZERO bit) raised. The ZERO bit in the STATUS register is raised (equal to 1) when the arithmetic result of an operation (in this instance decrementing ALIVECNT by 1) is zero. If ALIVECNT is not zero then we repeat the Delay subroutine and wait another 0.1ms and if ALIVECNT is zero then

we return back to the original place in the mainline where the Delay subroutine is called. This is the implementation of the while loop we saw in listing 1. Thus, the Delay subroutine not only toggles RC2 but also allows for an argument to be passed if we explicitly set ALIVECNT before calling the Delay subroutine. We must always remember to set ALIVECNT before calling Delay. The complementary flowchart to the Delay subroutine can be seen in figure 3. Here we assume before calling the Delay subroutine we have already set the value for the ALIVECNT register. Let us say ALIVECNT is 2 for the sake of demonstration. We then toggle the state of RC2 and delay for 0.1ms by calling LoopTime. Then we decrement ALIVECNT by 1 and store the value of it inside ALIVECNT. Now the value of ALIVECNT is 1. Then we check to see if the ALIVECNT register is zero. We know the value of it is 1. Thus we take the left arrow and loop back around. Now we toggle RC2 and wait another 0.1ms. So far we have waited a cumulative of 0.2ms. Then we decrement the value of ALIVECNT by 1 again. Now the new value of ALIVECNT is 0. Now we check to see if ALIVECNT is zero. We see that ALIVECNT is indeed zero and then return. It is importnat to note that we don't actually check to see if ALIVECNT is zero but we see if the ZERO bit of STATUS register was raised (is 1) when we compute the arithmetic ooperation of *decf ALIVECNT,F*. The ZERO bit is only raised if ALIVECNT is indeed 0 after being decremented.

B. Generation of any square wave

Now we are ready to create the final flowchart. We look at figure 2 and replace every grouping of *btg PORTC, RC2* and *rcall LoopTime* with two instructions that are *MOVLF X, ALIVECNT* followed by *rcall Delay*. Here the X in *MOVLF X, ALIVECNT* stands for the number of time units you would like to stall the microcontroller. We notice that we would like to make the flowchart such that it is in agreement with the general parameters of the problem outlined in equations 2 - 5. Thus we note that the amount of time units we would like to stall during the HIGH state is not the same as the amount of time we would like to stall for the LOW state. Considering all of this, we arrive to the flowchart shown in figure 4. Here we begin the mainline program and initialize everything. Then we set RA3 and RC3 to begin our HIGH state of the square wave. Then we move a decimal value (denoted X) into ALIVECNT which is the number of times we wish to stall for 0.1ms. If we choose X to be 5 then we stall of 0.5ms. We see that *rcall Delay* takes care of the timer and reference wave which we have already covered in figure 3. Regardless, the next step in figure 4 is to clear RA3 and RC3 to begin the LOW state of the rectangular square wave. Now we pass a new value into Y for how long we wish to wait or delay our microcontroller for. If we input 10 then we delay for 1 ms. Now we loop back into setting RA3 and RC3 and loop infinitely. Notice, by simply changing X and Y we can easily design any square wave (in multiples of 0.1ms) that we desire.

We implement the flowchart in figure 4 into the mainline program as seen in figure 11. Here you will notice we have plugged in values for X and Y such that they solve for question D. We begin by initializing the values by calling *rcall Initial*. Then we enter into our main loop. We set RA3 and RC3 to begin our HIGH state for the square wave and then execute *MOVLF 3, ALIVECNT* to set $ALIVECNT = 3$. Then we invoke *rcall Delay* thereby causing a delay for 0.3 ms since we set ALIVECNT to 3. Then we clear RA3 and RC3 to begin our LOW state of the square wave. Now we set *MOVLF 5, ALIVECNT* before calling the Delay subroutine so we have a LOW state for 0.5ms. Then on line 81 we jump back to Loop label in line 57. By running this we ge the same square wave as was shown in figure 8. We are now ready to solve all of the questions that were assigned to us.

```
;;;;;;; Mainline program ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Mainline
        rcall  Initial          ;Initialize everything
Loop

;First we wish to do the high cycle where RA3
; and RC3 are both high for 0.3 ms or three time units

bsf PORTA, RA3  ;set them to be high
bsf PORTC, RC3  ;set them to be high
;we want to wait for a total of 0.3 ms
MOVLF 3, ALIVECNT       ;set number of time units (0.1 ms) you want to be HIGH
rcall Delay



; now we clear them both so they're LOW for 0.5 ms
    bcf PORTA, RA3
    bcf PORTC, RC3

MOVLF 5, ALIVECNT       ;set number of time units (0.1 ms) you want to be LOW
rcall Delay




;repeat everything
    bra Loop
```

Fig. 11. Mainline program to output any square wave.

III. Square Wave Generation Problems

A. Part A: $T_{HIGH} = T_{LOW} = 0.1ms$

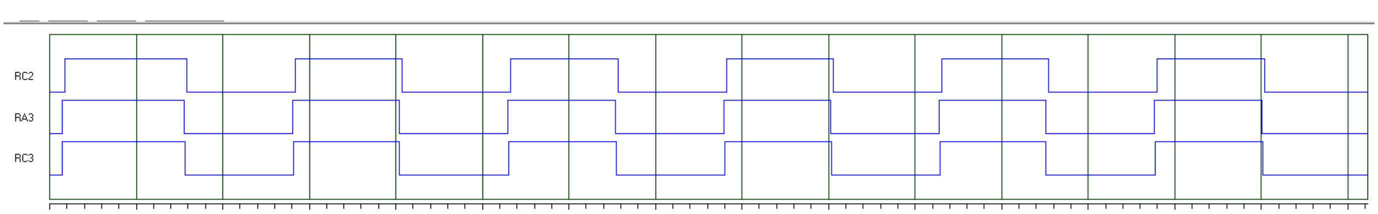

Fig. 12. $T_{HIGH} = T_{LOW} = 0.1ms$ with each state of RC2 lasting for 0.1ms

B. Part B: $T_{HIGH} = 0.1$ and $T_{LOW} = 0.4ms$

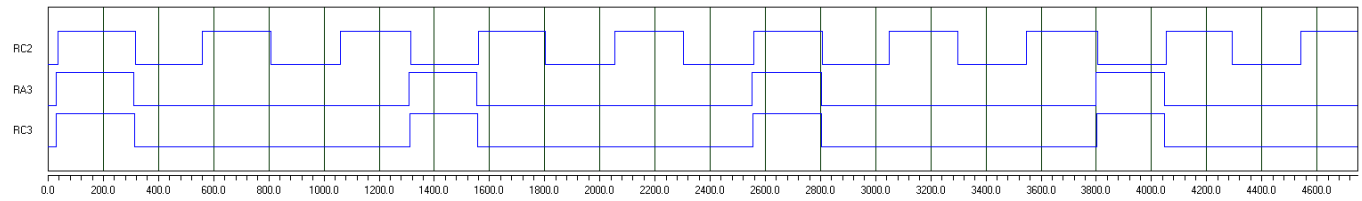

Fig. 13. $T_{HIGH} = 0.1$ *and* $T_{LOW} = 0.4ms$ with each state of RC2 lasting for 0.1ms

C. Part C: $T_{HIGH} = T_{LOW} = 0.3ms$

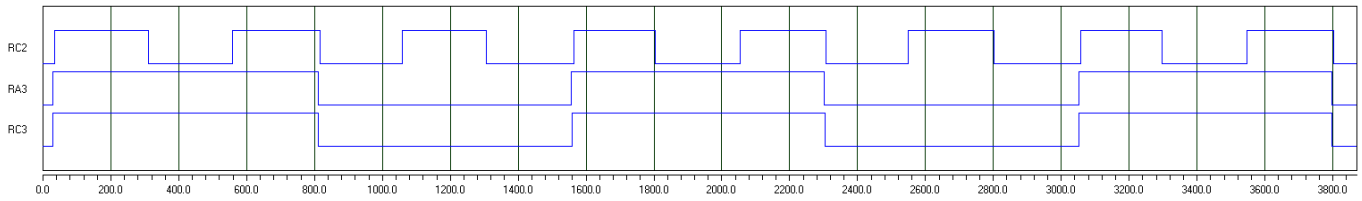

Fig. 14. $T_{HIGH} = T_{LOW} = 0.3ms$ with each state of RC2 lasting for 0.1ms

D. Part D: $T_{HIGH} = 0.3$ and $T_{LOW} = 0.5ms$

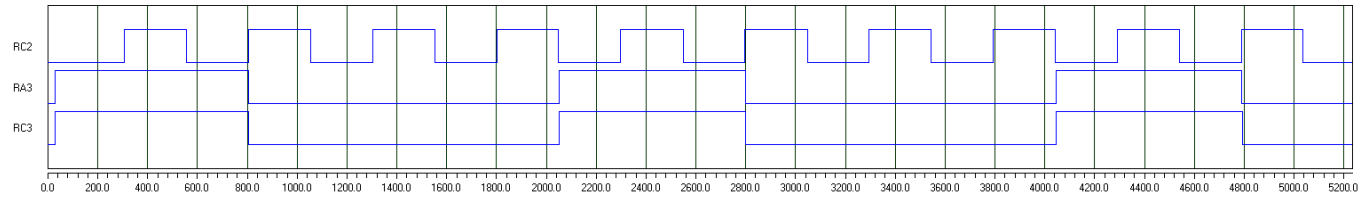

Fig. 15. $T_{HIGH} = 0.3$ *and* $T_{LOW} = 0.5ms$ with each state of RC2 lasting for 0.1ms