

In this post, I will attempt to explain how to write a subroutine (a function) in Microchip PIC assembly that makes the processor wait for given period of time (in terms of cycles).

Our subroutine takes one parameter over accumulator (working register). The parameter defines amount of cycles to wait. The time spent initializing the parameter is included in the total time intended to spend. So does the time spent returning from the subroutine.

Calculating the total time to wait, we will use:

- n : the amount of time slices to wait
- t_i : the time spent entering the subroutine
- t_u : the time spent in a non-final turn of the loop that is used to keep the processor busy (the time slice. In the following example that I'll give: 5 cycles)
- c : the iteration count of the busy waiting loop except the final iteration (final iteration is considered as leaving the subroutine)
- t_o : the time spent in the last turn of the loop plus the time spent returning from the subroutine.

We shall have this equation:

$$n * t_u = t_i + c * t_u + t_o \quad (1)$$

We will evaluate this formula in the following way:

$$n * t_u = (t_i + t_o) + c * t_u \quad (2)$$

if we arrange $t_i + t_o$ so that it is equal to an integer factor of t_u

$$n * t_u = (k * t_u) + c * t_u \quad (3)$$

The easiest use of this formula is choosing k as 1 or 2 so that loop count c will be $(n - 1)$ or $(n - 2)$. Let's implement this using PIC16F628A as our subject:

```

LIST P=16F628A, R=DEC      ;we'll use decimal radix
                           ;in our examples.
#include <p16f628a.inc>

ORG 0
Start:
    movlw 10                ;1 cycle of  $t_i$ 
    call waitCy             ;2 cycles of  $t_i$ 

    ;some code goes here

    goto $

;=====
;waitCy: waits for (w) * 5 cycles
waitCy:
    addlw -1                ;1 cycle of  $t_i$ 
L1:
    addlw -1                ;normal iteration 1 cycle
    nop                    ;normal iteration 1 cycle
    btfss STATUS, Z        ;normal iteration 1 cycle,
                           ;final iteration 2 cycles
    goto L1                ;normal iteration 2 cycles

    return                  ;2 cycles of  $t_o$ 

END

```

Figure 1

In *Figure 1*, we see that the t_u is set to 5 cycles: $1 + 1 + 1 + 2$. Initialization time t_i is set to 4 cycles: $1 + 2 + 1$. The t_o is set to 6 cycles: $1 + 1 + 2(\text{btfss when zero bit is set}) + 2(\text{return})$.

As seen above, $t_i + t_o$ is equal to 2 times of t_u . That means, $c = (n - 2)$ thus $k = 2$.

The tricky part here is: since we selected c to be $(n - k)$ we can not initialize the working register W with values smaller than k , in this example 2. Because if we do so, the value $c = (n - k)$ will be smaller than 0, which results in an unexpected way: how many times, except the last turn, will the loop repeat? Generally resulting in this -unintentional- way: loop count equals to the number that corresponds to the unsigned interpretation of the signed result of $(n - k)$.

Now whenever we would like our processor to wait for $k * 5 \leq n * 5 < 1280$ cycles to wait, we can do the following:

```
movlw n
call waitCy
```

We can combine some WaitCy subroutine calls *nop*'s and *goto \$+1*'s in order to wait for exact cycle count. Suppose that we want to wait for 493 cycles. We do the following:

```
movlw 98
call waitCy    ; waits for 490 cycles
goto $+1      ; + 2 cycles
nop           ; + 1 cycle
```

In order to be able to block the processor for more than 1275 cycles, one can do subsequent calls to WaitCy subroutine. For example in order to wait for 2574 cycles, one can write the following code:

```
movlw 255
call waitCy    ;waits for 1275 cycles
movlw 255
call waitCy    ;waits for another 1275 cycles
movlw 4
call waitCy    ;waits for 20 cycles
goto $+1      ;2 cycles
goto $+1      ;2 cycles

;In total, 2574 cycles are spent to come here.
```

As seen in the above example, sometimes it's hard to make the processor wait for cycle counts more than 1275; e.g. in order to wait for 2574 cycles, 3 subsequent calls to WaitCy have to be made, if we tried to make the processor wait for 167819 cycles, we had to write many more subsequent calls to WaitCy. In order to handle this situation, we are going to implement a new subroutine that uses WaitCy internally, and waits for (W) * 1000 cycles. We will call our new subroutine WaitKc (Kc stands for Kilo Cycles).

In order to call the subroutine the following has to be done:

```
movlw n        ;1 cycle
call waitKc    ;2 cycles
```

We want to use WaitCy subroutine while implementing the WaitKc subroutine, so the WaitKc subroutine will take it's parameter over the working register, but it can't use working register as the loop counter since working register will be used by WaitCy. This is why we will use a file register MCOUNT as loop counter:

```
movwf MCOUNT ;1 cycle
```

Now we will try the code the main loop inside the subroutine. Our first trial begins with choosing $k = 1$, thus one loop turn should take 1000 cycles:

```
waitKc:
    movwf MCOUNT
L2:
    movlw 199
    call waitCy          ;995 cycles
    decfsz MCOUNT     ;non-final loop:1 cycles
    goto L2             ;2 cycles. In total 998 cycles here.
    return
```

We have to add a `goto $+1` to somewhere in the loop:

```
waitKc:
    movwf MCOUNT
L2:
    goto $+1            ;2 cycles
    movlw 199
    call waitCy          ;995 cycles
    decfsz MCOUNT     ;non-final loop:1 cycles
    goto L2             ;2 cycles. In total 1000 cycles here.
    return
```

Now we have t_u : as 1000 cycles. How about t_i and t_o ? Since k is 1 until now, total execution time of entering the subroutine, last loop of subroutine and returning from the subroutine should take 1000 clock cycles. Let's see:

Initialization (t_i):

```
movlw n          ;1 cycle
call waitKc      ;2 cycles

movwf MCOUNT   ;1 cycle,  $t_i$  is 4 cycles in total.
```

Final turn of the loop and return from the subroutine (t_o):

```
L2:
    goto $+1      ;2 cycles
    movlw 199
    call waitCy   ;995 cycles
    decfsz MCOUNT ;2 cycles (it skips)
    goto L2       ;skipped
    return        ;2 cycles,  $t_o$  is 1001 cycles in total.
```

We see that the time $t_i + t_o = 1005$ cycles. We have to decrease this value to 1000 somehow, because it has to be equal to the value of t_u . We will achieve this goal by decreasing the amount of cycles spent by `WaitCy` call, so that the final turn and return will take 996 cycles in total (+4 comes from t_i , thus $t_i + t_o$ will be 1000 cycles). And in order to keep the $t_u = 1000$ condition, we will spend the rest of the cycles after we test if it's the final turn. If it's not the final turn, enough number of cycles will be spent to make a turn take 1000 cycles:

```

L2:
    goto $+1          ;2 cycles
    movlw 198
    call waitCy      ;995 to 990 cycles
    decfsz MCOUNT  ;2 cycles (it skips)
    goto L2          ;skipped
    return           ;2 cycles,  $t_o$  is 996 cycles in total.

```

But now, we see that a non-final turn of the loop takes:

```

L2:
    goto $+1          ;2 cycles
    movlw 198
    call waitCy      ;990 cycles
    decfsz MCOUNT  ;1 cycles
    goto L2          ;2 cycles, in total 995 cycles.

```

A non-final turn should take 1000 cycles. We can do it by writing

```
goto L3
```

instead of goto L2 and

```

L3:
    nop              ; +1 cycles makes 996 cycles
    goto $+2        ; +2 cycles makes 998 cycles
    goto L2         ; +2 cycles makes 1000 cycles

```

As you can see, this way we could add 5 cycles, and everything would be perfect. But! We can do this with less code. How?

Let's just change the way loop terminates. Now a non-final turn of the loop looks this way

```

L2:
    goto $+1          ;2 cycles
    movlw 198
    call waitCy      ;990 cycles
    decf MCOUNT     ;1 cycles
    btfsc STATUS, Z  ;2 cycles
    return           ;this one is skipped.
    goto $+1         ;2 cycles
    nop              ;added this to make 1000 in total
    goto L2          ;2 cycles, in total 1000 cycles.

```

We put the return inside the loop. A final turn of the loop looks this way:

```

    goto $+1          ;2 cycles
    movlw 198
    call waitCy      ;990 cycles
    decf MCOUNT     ;1 cycles
    btfsc STATUS, Z  ;1 cycles
    return           ;2 cycles in total 996 cycles

```

That's all! Our subroutine looks like this now:

```
=====
;waitKc waits for (W) * 1000 cycles
waitKc:
    movwf MCOUNT
L2:
    goto $+1
    movlw 198
    call waitCy
    decf MCOUNT
    btfsc STATUS, Z
    return
    goto $+1
    nop
    goto L2
=====
```

Before writing the *WaitKc* subroutine, we stated that “if we tried to make the processor wait for 167819 cycles, we had to write many more subsequent calls to *WaitCy*”. Now, we will combine a few *WaitKc* and *WaitCy* calls, one or two *goto \$+1*'s and *nop*'s, and all will be done this easy!

See how to wait for 167819 cycles:

```
movlw 167
call waitKc    ;167000 cycles
movlw 163
call waitCy    ;815 cycles
goto $+1      ;2 cycles
goto $+1      ;2 cycles, and in total 167819 cycles to get here.
```

Here's the conclusion: you can use these subroutines to wait for an arbitrary number of cycles to synchronise, or for any other reason!

Good Luck!

Hayri Uğur KOLTUK

08 February 2009

