

Timer Interrupt Usage in 16bit PIC Microcontrollers

A Dynamic Wait Implementation

Omar Bennani
Electrical and Computer Engineering
Michigan State University
Friday, March 30, 2007

Abstract

Many microcontroller designs require delay operators and a method to parallelize processes. PIC microcontrollers have built in timers which have many potential uses. A timer can be used to create a dynamically adjustable wait statement or have multiple processes which will automatically interrupt each other and run automatically. They can also be used to control A/D sampling and act as a free counting register. This application note will cover the general usage and a specific application of a dynamic wait.

Keywords: PIC microcontroller timer interrupt wait idle

Introduction

PIC microcontrollers have a diverse range of applications and are used throughout industry and hobbyists alike. Many applications call for low power. As a result it is very advantageous to put a microcontroller into an idle state when there are no important tasks running so that power can be saved. Also, many real world signals and controls do not operate at processor speeds so it is necessary to create dynamic wait or hold times. This project describes timer usage in general and a method wait precise time periods dynamically.

Objective

The objective is to setup an easy to use wait statement that will still operate if the processor is put into idle. This is a much more advantageous setup vs. for loops. The code is simpler and short, it allows the processor to be put into idle while waiting for the timer to count, or the processor can continue computing other tasks and not be held in a loop waiting for one particular function to finish.

Timer Initialization

Timer interrupts are most easily generated using the included compiler functions in the timer.h header file. The other option is to manually set the flags and registered that configure the timer. I will focus on the timer.h method because the end effect is the same. Also if any changes are desired to be made via manual register setting then it is very easy to read the header file and accompanying functions to find what the register settings are.

The most important function to call is `OpenTimerx`. The 'x' stands for the numbers 1-5, 23, or 45. The 'x' syntax will apply from this point onward. There are two timer sizes 16 and 32 bits. When using just the number 1-5 it will be a 16bit timer and when using 23 or 45 it's a 32bit timer. The difference will be explained later. The syntax for `OpenTimerx` calls for two inputs, config and period. The config input has the following input variables. The text inside the parenthesis is the possible parameters.

- `Tx_(ON OFF)` : Turn module on or off
 - Ex. `T1_ON` → For an `OpenTimer1`, it will turn the module on
- `Tx_IDLE_(CON STOP)` : Run or Stop during Idle
 - Ex. `T1_IDLE_CON` → `Timer1` continue to run during Idle
- `Tx_PS_1_(1 8 64 128)` : Timer Prescaler
 - Ex. `T1_PS_1_8` → For every 8 input clock cycles the timer will count once
- `Tx_SYNC_EXT_(ON OFF)` : Synchronous Clock Enable
 - Ex. `T1_SYNC_ON` → Synchronize the changes for an external Clock input
- `Tx_SOURCE_(EXT INT)` : Clock Source
 - Ex. `T1_SOURCE_INT` → Internal clock used to trigger counting

The period input is the value which is used to trigger the interrupt. The timer will count until it hits the period value. The available period is the primary difference between a 16

and 32bit timer. For a 16bit timer the period value is has a range of 0x0 to 0xFFFF or up to 0xFFFFFFFF for a 32bit timer. The final full syntax is would be something like:

```
OpenTimer1(T1_ON & T1_IDLE_CON & T1_PS_1_64 &
T1_SYNC_EXT_OFF & T1_SOURCE_INT);
```

This would turn on the timer, have it continue during idle, divide the internal clock by 64 and make the clock source internal.

Two other functions are important for general use, WriteTimerx and ConfigIntTimerx. WriteTimerx will write a new value to the count register. For general initialization the syntax is:

```
WriteTimerx(0);
```

0 can be replaced with any integer up to the match value, depending on your desired usage. ConfigIntTimerx sets the interrupt priority along with whether or not the timer interrupt is on. It takes two input values

- Tx_INT_PRIOR_(0-7) : Interrupt Priority level.
 - Ex. T1_INT_PRIOR_2 → Timer interrupt priority is 2
- Tx_INT_(ON OFF) : Interrupt Timer on or not
 - T1_INT_ON → Timer interrupt is on

The complete syntax is:

```
ConfigIntTimer1(T1_INT_PRIOR_2 & T1_INT_ON);
```

An important thing to note is that interrupt priority for the processor is setup where a larger number is a higher priority. By default the processor is at a priority level 0. The implication is that if the interrupt priority is set to 0 it will only be capable of waking the processor from idle. If there are any currently running tasks the interrupt will never actually interrupt the microcontroller.

Timer Usage

Once the timer is initialized with a clock source, period, etc. it is going to always count. If the interrupt is turned on then it will trigger an interrupt when the counter hits the period value. In order to use the interrupt that is triggered an interrupt function must be created that is associated with the timer interrupt that is being used. The syntax for that interrupt is:

```
void __attribute__((__interrupt__)) _TxInterrupt(void)
{
/*type your code here */
}
```

The syntax for the interrupt is built into the compiler. Without this function the interrupt will occur and will not have a function or task to run so the processor will hang. The

interrupt function can have any standard function calls in it. Although, if any variables modified or created during the interrupt need to be accessed by other functions they must be global variables.

There are other important considerations for the interrupt function. In order to prevent the function from being repeatedly called one must place the line:

```
IFS0bits.TXIF = 0;
```

at some point in the interrupt function. This line acts to clear the interrupt flag generated by the timer. If the flag is not cleared the timer may be reset but since the flag isn't the processor will return to the interrupt function repeatedly. Depending on the implementation it is also important to reset the timers count value. The timer will continue to count and will loop around to zero when it reaches its maximum. If it isn't reset then the time between interrupts will be much greater from that point. This is easily avoided with one line of code:

```
WriteTimerx(0);
```

Depending on your intended usage there are other functions which may be useful.

```
/*This function will return the current value of the Timer register.*/
```

```
Var = ReadTimerx();
```

```
/*The following two macros will enable or disable the timer interrupt*/
```

```
EnableIntTx;
```

```
DisableIntTx;
```

```
/*This function will set the priority of the Timer interrupt*/
```

```
/*It accepts integer values from 0-7*/
```

```
SetPriorityIntTx(4);
```

Dynamic Wait

In order to create a dynamically changeable wait statement a timer is implemented that will count at some predetermined rate. By setting the match value to the highest possible and utilizing the WriteTimerx function it becomes trivial to have a dynamic wait function. The following code shows how it would be implemented in a dsPIC30F3013.

```

#include <p30f3013.h>
#include <timer.h>

int timer_wait = 0;
/* set the clock to the internal fast RC x16. This will create a 29.48Mhz clock signal*/
_FOSC(CSW_FSCM_ON & FRC_PLL16);

void main(void) {

/* setup the wait timer interrupt */
  ConfigIntTimer1(T1_INT_PRIOR_5 & T1_INT_ON);
  WriteTimer1(0);

/*To call the wait statement*/
  dynhold(10); //will wait for 10ms

/*Insert your own code after this point*/

}

/*This will wait for how ever many ms it's given*/
void dynhold(int wait){
  OpenTimer1(T1_ON & T1_GATE_OFF & T1_IDLE_CON & T1_PS_1_64 &
  T1_SOURCE_INT, 0xFFFF);

  EnableIntT1;
  /*the 460 is chosen to compensate further for the ~30Mhz clock. So the result is that
  for a given number of ms the timer register is set to leave only that much before the
  highest value. This current setup is capable of waiting up to 142ms.*/
  WriteTimer1(65535 - wait*460);

/*The following lines can be put elsewhere to allow the processor to continue and do
other operations. If no other operations are necessary then simply setting the processor to
Idle will save power*/

  while(timer_wait != 1) {
    Idle();
  }
  DisableIntT1;
  timer_wait = 0;
}

/*The actual ISR for the interrupt*/
void __attribute__((__interrupt__)) _T1Interrupt(void) {
  timer_wait = 1;
  WAIT_INTERRUPT_FLAG = 0;
}

```

Conclusion & Recommendations

The given code can wait up to only 142ms, if a longer wait period is desired there are many options such as calling the function more than once, increasing the prescaler, slowing down the system clock, or providing a slower external clock. In particular, by using a slower external clock a timestamp system can be created to count in seconds or to implement a clock. This code will also facilitate power saving as a result of being able to place the processor into idle for the majority of the time. For instance, consider interrupting every 1ms to perform a calculation and then idle. When considering that the processor would be otherwise running at 29.48Mhz constantly then by putting it into idle it may only operate for 3 μ s every 1ms. This is a tremendous difference to constant operation.

References

Language Tool Library Manual for 16-Bit Microchip Microprocessors.

http://www.egr.msu.edu/classes/ece480/goodman/spring/group12/documents/Language_Tool_Library_Manual.pdf

dsPIC30F Reference Manual.

http://www.egr.msu.edu/classes/ece480/goodman/spring/group12/documents/dsPIC30F_Reference_Manual.pdf