

Developing Timer-Based, Bit-Banged Pulses with the Programmable XBee

Even though the Programmable XBee (PXBee) has hardware-based Pulse Width Modulation (PWM) there are times when it is advantageous to develop pulses by bit-banging via a timer-based interrupt. This paper describes such a method and includes a sample program and XML file compatible with the Digi Eclipse extensions to Code Warrior® for the Programmable XBee.

The first step in the process is to create a PXBee project using the GUI. In the Smart Editor you will need to ADD the I/O pins you want to use for the pulses as well as a Timer-based interrupt. Here is a screen shot of my Smart Editor after adding these features:



As you can see, I selected pins 18 and 19 for the pulses and edited the default names (gpio0 and gpio1) so that it is easier in the code to determine which one I am programming.

There are two *#define* statements which determine the pulse period and interrupt period. A third *#define* calculates the number of interrupts required for the pulse period.

**#define** PWM\_PERIOD\_MS 10 // in msec

**#define** TIMER\_PERIOD 100 // in usec

**#define** PWM\_PERIOD\_COUNT (PWM\_PERIOD\_MS\*1000/TIMER\_PERIOD)

// nbr of interrupts for desired period

The \*1000 factor is because the timer interrupt period is in microseconds and the PWM period is in milliseconds.

Having these values as definitions is good programming practice which allows you to modify the operating characteristics without having to find the statements which control them.

One of the “features” of the ISR that may stand out to you is that it uses assembly language statements to create the pulses. Initially the ISR was written completely in C. However, the execution time of the ISR was determined to be too long. This is somewhat subjective but when the ISR execution time was measured as being about 80% of the ISR period, it was decided to change the I/O statements to assembly language. The disassembly window of the ISR showed that the other statements were already as efficient as hand-coded assembly would be so they were left alone. Several macros were defined for the assembly statements in order to make the code easier to read. Here is one of the four macros:

**#define** GPIO18\_SET bset 2, PTBD

The bit & port values for the macros were determined using the PXBee block diagram.

The operation of the ISR is relatively straight forward. Here is a brief description in outline form:

1. Increment the ISR counter
2. If the count is greater than or equal to the value required to generate the desired period
	1. Set the ISR counter to 0
	2. Start the pulses
3. Otherwise
	1. If the ISR count is greater than or equal to the value required for the pin 18 pulse width
		1. Stop the pulse on pin 18
	2. If the ISR count is greater than or equal to the value required for the pin 19 pulse width
		1. Stop the pulse on pin 19
4. Exit the ISR

As you can see by viewing the ISR code, it is very easy to add more I/O pins for PWM. All that needs to be done is to:

1. ADD the I/O pin via the Smart Editor
2. Create a global variable to contain the number of interrupts required for the desired pulse width
3. Define the required macros with the appropriate assembly language statements
4. Insert two statements into the ISR

The main program initializes the ISR counter to 1 so that no pulses are generated for the first PWM period. This allows time for other initialization to execute with no pulses being generated. If you do not need this feature you can set the ISR counter to *PWM\_PERIOD\_COUNT* which will cause the ISR to start generating pulses the first time it is executed. The pulse width values are then initialized and the interrupt is enabled.

If your requirement is such that you want to vary the pulse width, all you need to do is modify the global variable associated with the I/O pin. For example, changing *PWMvalue\_gpio18* from 10 to 15 will change the pulse width from 1msec to 1.5msec because the ISR period is 100usec.

You do need to ensure that the count value inserted into the PWM variable is less than the calculated value *PWM\_PERIOD\_COUNT*. If the pulse width value is greater than or equal to *PWM\_PERIOD\_COUNT*, the I/O pin will go high and never be set low.

There are at least two other features which may be easily added to this method:

1. One is to output a specific number of pulses. This can be accomplished by implementing a counter variable, setting it to the number of desired pulses and decrementing it in the ISR. When it reaches 0 the ISR could be disabled or the ISR could simply be exited. The main program can use the counter value to determine whether or not the pulses have terminated.
2. Another is that of Pulse Position Modulation. A separate set of variables could be implemented that define starting count values for the pulses. The ISR would then be written such that the pulse would be high only when the ISR period counter is between the start and stop values.

There are some downsides to this method of pulse generation:

1. It takes up CPU processing time which the hardware generation facility does not. The amount of time it takes up is dependent on the amount of processing that occurs in the ISR. This paper has already pointed out one modification of the ISR code which was implemented due to this issue.
2. Another is that the resolution for the pulse width and the period is the ISR period. The shorter the period the higher the resolution. However, as the ISR period is shortened the percentage of CPU time consumed by the ISR is increased. This can easily be detrimental to the operation of the main program.