ZBasic

Application Note

AN-216 High Resolution Pulse Width Modulation

Introduction

The ZX series devices have at their core an Atmel ATMega32 or ATMega644 microcontroller. These processors have timers that allow the user to produce hardware-based Pulse Width Modulation. Once all the settings are made, the hardware outputs a square wave signal with a variable duty factor and a variable frequency. This frees the processor for other tasks yet the processor can change the PWM settings whenever necessary. PWM can be used for many purposes. Some of the more common uses are for driving RC-type servos, driving DC motors via an H-bridge, and digital-to-analog conversion.

Timer Hardware

In its essence, a timer is simply a counter that is incremented after a certain number of cycles of the main clock. The number of main CPU clock cycles it takes for each increment of the timer counter is called the prescaler value. Prescaler values for Timer 1 are limited to 1, 8, 64, 256, and 1024 on the current core processors for the ZX hardware. Other timers may have different prescaler settings. With a prescaler value of 1, the timer is incremented for every main clock cycle. When the prescaler is 1024, the timer increments once for every 1024 main clock cycles. The rest of the prescaler values function in a proportionally similar way. The “timing” aspect of this comes about by having a “trigger” value (the actual name of this trigger value depends on the specifics of the timer implementation). The trigger value is continuously compared to the current timer counter value and if the timer is equal to the output compare value, then an action is taken.

The processor’s timers are able to operate in several different modes, of which several are useful for generating a PWM signal. The PWM modes can be further divided into sub-groups. One sub-group is called “Phase Correct PWM” (PC-PWM) and the other is “Fast PWM” (F-PWM). The primary difference between these two modes is that in the Fast PWM mode the counter only counts in the upward direction while in the Phase Correct mode the counter cycles upward and downward between two limits. The F-PWM mode can create pulses at about double the frequency of PC-PWM. Figures 1 and 2 illustrate these modes of timer counting and output.

Phase Correct PWM

For Phase Correct PWM, the counter starts counting up from the Bottom value, which for both F-PWM and PC-PWM is zero. As the timer counts up, it eventually hits a trigger threshold. There are two trigger values set in the registers OCR1A and OCR1B (this stands for **O**utput **C**ompare **R**egister for timer **1** **A** or **B**). It should be between the Bottom and Top values. When the trigger value is reached an output pin is brought either high or low. For the ZX-24, two output pins can be used, called OC1A and OC1B on the microcontroller hardware. These pins are assigned to pin

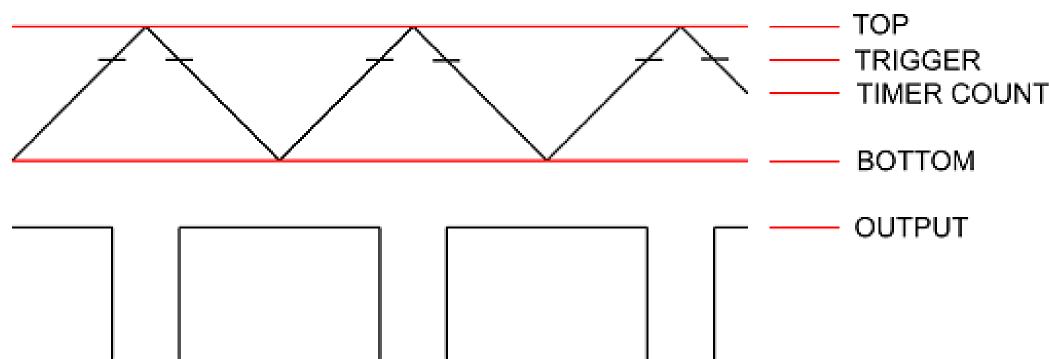


Figure 1 - Phase-correct PWM Timing Diagram

Copyright © 2006 Elba Corp.

26 (green LED) and pin 27 respectively on the ZX-24. This application note focuses on using Timer1 but the other timers in the ZX device are also capable of PWM generation. Timer 0 is dedicated to the RTC function and therefore cannot readily be used for PWM generation. Timer 2, however, may be available for PWM in your application. This is left as an exercise to the reader.

The OC1A pin in controlled by the OCR1A trigger point, and the OC1B is controlled by OCR1B. Other hardware will have the physical pins assigned elsewhere but the pin and register names will remain the same. After reaching the trigger value, and the Output Compare pin is set/cleared, then the timer continues to count upward until the TOP level is reached. The Top value is set by the user in the ICR1 register (called Input Capture Register 1, but is not being used as such in the mode described here) of the microcontroller. The Top value along with the prescaler and main clock frequency (currently always 14.7456 MHz) determine the frequency of the timer. The Timer 1 Top variable is 16 bits (the other timers are 8 bits), and any value that fits in that range is allowed. Note, however, that the Top value must be greater than the Trigger value. Smaller values of the Top give lower resolution to the Trigger. Therefore, the prescaler should be selected to allow the largest Top value for a given timer frequency.

Often several prescaler values will result in a valid Top variable so one must choose the prescaler carefully. See Table 1 for more insight into the relationship between the prescaler value, resolution and maximum frequency. As the table shows, there is considerable overlap between the frequencies and the bit resolutions. To achieve the highest possible bit resolution, one should use the lowest prescaler value that will support the desired combination of resolution and frequency.

Table 1a 16-Bit Output Compare Register values vs. Frequency and Prescaler

Frequency	16-bit Counter: Phase Correct							16-bit Counter: Fast PWM						
	Prescaler Values							Prescaler Values						
	1	8	32	64	128	256	1024	1	8	32	64	128	256	1024
0.1														
0.2							36000							
0.5						57600	14400							28799
1					57600	28800	7200						57599	14399
2				57600	28800	14400	3600					57599	28799	7199
5			46080	23040	11520	5760	1440				46079	23039	11519	2879
10			23040	11520	5760	2880	720			46079	23039	11519	5759	1439
20		46080	11520	5760	2880	1440	360			23039	11519	5759	2879	719
50		18432	4608	2304	1152	576	144		36863	9215	4607	2303	1151	287
100		9216	2304	1152	576	288	72		18431	4607	2303	1151	575	143
200	36864	4608	1152	576	288	144	36		9215	2303	1151	575	287	71
500	14745	1843	460	230	115	57	14	29490	3685	920	459	229	114	27
1000	7372	921	230	115	57	28	7	14744	1842	459	229	114	56	13
2000	3686	460	115	57	28	14	3	7371	920	229	114	56	27	6
5000	1474	184	46	23	11	5	1	2948	367	91	45	22	10	1
10000	737	92	23	11	5	2		1473	183	45	22	10	4	0
20000	368	46	11	5	2	1		736	91	22	10	4	1	
50000	147	18	4	2	1			293	35	8	3	1	0	
100000	73	9	2	1				146	17	3	1	0		
200000	36	4	1					72	8	1	0			
500000	14	1						28	2					
1000000	7							13	0					
2000000	3							6						
5000000	1							1						
Cutoff	112.5	14.06	3.516	1.758	0.879	0.439	0.110	225.0	28.13	7.031	3.516	1.758	0.879	0.220

Copyright © 2006 Elba Corp.

Table 1b 8-Bit Output Compare Register values vs. Frequency and Prescaler

Frequency	8-bit Counter: Phase Correct							8-bit Counter: Fast PWM						
	1	8	32	64	128	256	1024	1	8	32	64	128	256	1024
1														
2														
5														
10														
20														
50							144							
100							72							143
200						144	36							71
500				230	115	57	14					229	114	27
1000			230	115	57	28	7				229	114	56	13
2000			115	57	28	14	3			229	114	56	27	6
5000		184	46	23	11	5	1			91	45	22	10	1
10000		92	23	11	5	2			183	45	22	10	4	0
20000		46	11	5	2	1			91	22	10	4	1	
50000	147	18	4	2	1				35	8	3	1	0	
100000	73	9	2	1				146	17	3	1	0		
200000	36	4	1					72	8	1	0			
500000	14	1						28	2					
1000000	7							13	0					
2000000	3							6						
5000000	1							1						
Cutoff	28913	3614	903.5	451.8	225.9	112.9	28.24	57600	7200	1800	900	450	225	56.25

Once the Top value is reached, the timer counts backward toward zero. When the counter hits the trigger again, the output pin is toggled to the opposite value and the counter continues decrementing to zero. When zero is reached, it begins to count upward, and the sequence repeats.

The reason that this mode is called “phase correct” is that it is technically possible to change the Top and Trigger values on the fly. The ZX implementation may not have the speed that is required to do this properly, but when code is written in Assembler for the raw AVR hardware, for frequencies up to a few hundred KHz, the pulse width can be adjusted on a per-pulse basis. Using PC-PWM, the symmetry of the spacing before and after each pulse is maintained. When filtered properly, it is possible to create a pseudo-sine wave signal. Therefore, within the audio spectrum, it is possible to create a useful complex sound.

Fast PWM

This mode functions the same as PC-PWM except that the counter does not count backward after reaching the TOP value. The timer starts counting at zero. When it reaches the trigger value set in OCR1A or OCR1B it then sets the corresponding output pin in the same manner as PC-PWM. When the counter reaches the TOP value, the output pin is toggled. On the next clock cycle the counter rolls over to zero. Because the timer does not need to count backward to zero, it is able to run at about double the frequency of PC-PWM. It is important to understand that Fast PWM mode does not have resolution equal to TOP steps. Rather, it has TOP+1 steps because it counts zero as a step. For example, if Top is set to five, the F-PWM counter will count 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, etc. This makes 12 steps for two cycles, or TOP+1 steps when counting from zero to Top. In contrast, in PC-PWM mode the counting would be 0, 1, 2, 3, 4, 5, 4, 3, 2, 1. This makes 10 steps for a full cycle, or only TOP steps when counting from 0 to TOP. This difference in counting makes the ratio of the frequencies not exactly 2:1.

Copyright © 2006 Elba Corp.

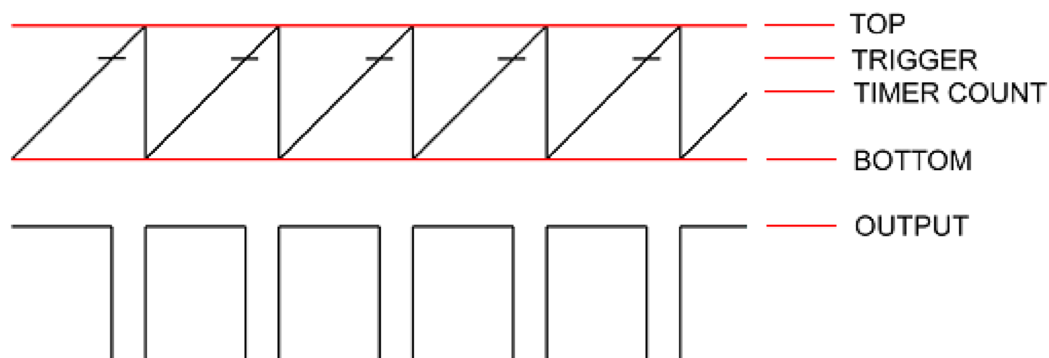


Figure 2 - Fast PWM Timing Diagram

The mathematical equations that apply to the two PWM modes are as follows:

Equation 1) $$\text{Phase Correct PWM Timer Frequency } (f_{timer}) = \frac{f_{clk}}{2 \times \text{Pre} \times \text{Top}}$$

Equation 2) $$\text{Fast PWM Timer Frequency } (f_{timer}) = \frac{f_{clk}}{\text{Pre} \times (\text{Top}+1)}$$

F_{clk} is the clock frequency of the processor (14.7456 MHz), *Pre* is the prescaler that has been chosen, and *Top* is the selected maximum count of the timer.

Commonly, you will decide on a certain PWM frequency and determine the Top value that will provide that frequency. Equations 1 and 2 can be rearranged for this purpose, resulting in Equations 3 and 4.

Equation 3) $$\text{Phase Correct Top} = \frac{f_{clk}}{2 \times \text{Pre} \times f_{timer}}$$

Equation 4) $$\text{Fast PWM Top} = \left(\frac{f_{clk}}{\text{Pre} \times f_{timer}} \right) - 1$$

The optimum prescaler can be determined by iteratively trying prescaler values starting with the lowest until the largest Top value less than 65536 (2^16) is obtained. This iterative technique can be used in a ZBasic subroutine example as illustrated below.

```
Sub Calc_PC_PWM_Settings (ByRef Freq as UnsignedLong, _
                          ByRef Top as UnsignedInteger, _
                          ByRef Pre as UnsignedInteger)

'this subroutine is given Frequency in the range of 1 to 7,373,800 (or MainClock/2)
'it returns Top and Pre with both selected to allow as large a Top value as possible

Dim Temp_UL as UnsignedLong    'use a UL because we need the extra digits
Dim CLK As UnsignedLong    'standard is 14,745,600
CLK = Register.CPUFrequency    'do NOT assume it will always be 14.7Mhz!
Pre = 1                         'start with the lowest prescaler
Temp_UL = CLK\(2*CULng(Pre)*Freq)    'calculate the Top value

If Temp_UL > 2^16 then                'check to see if it is too large
     Pre = 8
```

```
     Temp_Ul = CLK\(2*CULng(Pre)*Freq)   'try the next larger prescaler

     If Temp_UL > 2^16 then
          Pre = 64
          Temp_Ul = CLK\(2*CULng(Pre)*Freq)   'and so-on

          If Temp_UL > 2^16 then
               Pre = 256
               Temp_Ul = CLK\(2*CULng(Pre)*Freq)

               If Temp_UL > 2^16 then
                    Pre = 1024
                    Temp_Ul = CLK\(2*CULng(Pre)*Freq)

                    If Temp_UL > 2^16 then   'ERROR Too high a requested Frequency
                         Temp_UL = 0         'Zero is returned if there is an error
                         Pre = 0
                    End If
               End If
          End If
     End If
End If
TOP = CUInt(Temp_Ul)     'this is tested to be within the magnitude of UI
End Sub
```

Programming

To begin PWM generation, the Timer Mode, Prescaler, Top, Trigger, and starting count values need to be set. These values are set in several CPU registers. Confusingly, two of these registers carry several of these values simultaneously, and the Mode value is split across two of registers. This makes the operation of setting the parameters somewhat non-intuitive, particularly for the Mode.

The timer mode, OC1A/B pin modes, and the prescaler are set in the two Timer/Counter Control Registers (TCCR1A and TCCR1B, which stands for Timer/Counter Control Register 1 A/B). For the modes described in this application note, the Top value will be set in ICR1. The trigger values will be set in the OCR1A and OCR1B (Output Compare Register 1 A/B). The current count for the timer is TCCNT1. While it is not strictly necessary to change TCCNT1, when you are setting ICR1 it is probably best to set the count back to zero as well.

The bits in each register are used as follows:

	BIT FIELD							
Register	7	6	5	4	3	2	1	0
TCCR1A	OC1A Mode	OC1A Mode	OC1B Mode	OC1B Mode	NA	NA	Timer Mode:3	Timer Mode:2
TCCR1B	NA	NA	NA	Timer Mode:1	Timer Mode:0	Prescaler	Prescaler	Prescaler
OCR1A/B	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ICR1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TCCNT1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Table 2 – Timer1 Control Register Bit Assignments

Copyright © 2006 Elba Corp.

Variable	Bit 3	Bit 2	Bit 1	Bit 0
Output Compare Mode: Normal	-	-	1	1
Output Compare Mode: Inverted	-	-	1	0
Output Compare Mode: Disabled	-	-	0	0
Timer Mode: F-PWM	1	1	1	0
Timer Mode: PC-PWM	1	0	1	0
Prescale: Timer Disabled	-	0	0	0
Prescale: 1	-	0	0	1
Prescale: 8	-	0	1	0
Prescale: 64	-	0	1	1
Prescale: 256	-	1	0	0
Prescale:1024	-	1	0	1

Table 3 – Bit Maps of Function Assignments

Exercise: Driving a Servo

Typical RC servos require a control pulse lasting from 1 millisecond to 2 milliseconds, with the center position being defined as about 1.5mS. The frequency should be about 50 Hz.

Equations 2 and 3 can be used to determine that the optimum mode for this application is Fast PWM with a prescaler of 8 and a Top count of 36863. Since the period of 50Hz is 20 milliseconds, the shortest allowed pulse width of 1mS is equal to a duty factor of 0.05 (1/20), which corresponds to a count of 1843. The longest allowed pulse width of 2mS is a duty factor of 0.10 (2/20), corresponding to a count of 3686. By subtracting these two values, the full rotation range of the servo is determined to be about 1843 steps. For a servo capable of 60 degrees of swing each way each step corresponds to approximately 0.065 degrees.

The code fragments below can be used to set the parameters to drive two independent servos. The first fragment defines a group of constants that are useful to manipulate the timer.

```
Public Const pre_mask      as Byte = &B0000_0111
Public Const pre_8         as Byte = &B0000_0010
Public Const oc1a_mask     as Byte = &B1100_0000
Public Const oc1b_mask     as Byte = &B0011_0000
Public Const oc1a_norm     as Byte = &B1000_0000
Public Const oc1a_invert   as Byte = &B1100_0000
Public Const oc1b_norm     as Byte = &B0010_0000
Public Const oc1b_invert   as Byte = &B0011_0000
Public Const t_mode_maska  as Byte = &B0000_0011
Public Const t_mode_maskb  as Byte = &B0001_1000
Public Const f_pwm_modea   as Byte = &B0000_0010
Public Const f_pwm_modeb   as Byte = &B0001_1000
Public Const pc_pwm_modea  as Byte = &B0000_0010
Public Const pc_pwm_modeb  as Byte = &B0001_0000
```

The following code fragment shows how the timer can be prepared to drive the servo using Fast PWM. The PWM duty cycle is initialized for the center position for one servo and the zero position for the other.

```
'clearing the prescaler will stop the timer
Call SetBits(Register.tccr1b, PRE_MASK, &B0000_0000)

'set the timer mode to F-PWM
```

Copyright © 2006 Elba Corp.

```
Call SetBits(Register.tccr1a, t_mode_maska, f_pwm_modea)
Call SetBits(Register.tccr1b, t_mode_maskb, f_pwm_modeb)

'set the output compare modes to normal
Call SetBits(Register.tccr1a, oc1a_mask, oc1a_norm)
Call SetBits(Register.tccr1a, oc1b_mask, oc1b_norm)

'set the top value to give 50 hz when combined with a prescale of 8
Register.icr1 = 36863

'set the timer count to zero
Register.tcnt1 = 0

'set OCR1a to place the servo at the middle position (1.5mS)
'50Hz has a period of 20.0 milliseconds
Register.ocr1a = CUInt(36863.0 * 1.5 / 20.0)

'set OCR1b to place the servo to one extreme side
Register.ocr1b = CUInt(36863.0 * 1.0 / 20.0)

'LAST, set the prescaler to 8, and timing will commence
Call SetBits(Register.tccr1b, PRE_MASK, PRE_8)
```

Once this is done, you can easily change the servo positions by just changing the OCR1A and OCR1B registers.

```
'set OCR1a to place the servo at the 25% position (1.25mS)
Register.ocr1a = CUInt(36863.0 * 1.25 / 20.0)

'set OCR1b to place the servo at the 75% position (1.75mS)
Register.ocr1b = CUInt(36863.0 * 1.75 / 20.0)
```

To achieve the greatest precision, you may need to calculate the ACTUAL frequency at which the timer is working. It is unlikely to be precisely the intended frequency, but most likely is quite close. Using Equation 2, the actual timer frequency in the above exercise is 50.0027 Hz. Therefore, simply using 50Hz in calculations would probably be sufficient, but using the true frequency will give the best possible accuracy in setting the servo positions.

Exercise: DAC Output

Theory

It is possible to create a pseudo-analog output voltage by averaging the digital output. Since it is not possible to vary the **voltage** of the digital output (it is either ON or OFF), we must vary the duration of the ON and OFF times. The duty factor is the ratio of the ON time to the total ON + OFF time, and it determines the average voltage output. For instance if the duty factor is 50% the output pin will spend half its time at zero volts, and the other half at 5 volts and the average voltage is 2.5 volts. When viewed on a sufficiently small time scale, the output is simply either zero or 5 volts, and nothing in between, but with averaging we lose sight of the ON and OFF peaks and valleys.

A capacitor can be used to create the simplest averaging filter. It will store the charge (and therefore the voltage) over time creating the averaging effect. The capacitor can be thought of as a voltage reservoir. When the output is high, the charge flows into the capacitor, gradually increasing its voltage. Then when the output is low, it will draw the charge out of the capacitor, lowering its voltage. If the flow is too fast, it will fill the capacitor instantly and it will instantly rise to maximum voltage. Then when the flow is reversed, it will instantly drain to zero volts. We need a way to make the filling and draining more gradual. This can be achieved by adding a resistor between the output pin and the capacitor. This slows the change in the voltage of the capacitor and allows it to hold the simply average voltage. Because the charge is constantly going in or out, we can never make the voltage perfectly stable. The more stable we try to make the voltage (i.e. less ripple in the voltage), the more slowly the capacitor is able to

Copyright © 2006 Elba Corp.

change its voltage when the user changes the duty factor. Because we live in an imperfect world, we can not have perfectly instantaneous response to changes, and have perfectly stable voltage as well. We must compromise. Exactly where the compromise is made depends on the user's needs. Some requirements are diametrically opposed, for instance accuracy of the averaging vs. the speed that the filter can deliver the correct average voltage. High frequency vs. number of steps in the duty cycle (granularity). Simplicity of the circuit vs. quality of filtering. Size and expense of the components vs. quality of filtering.

As can be seen from this list, one of the major concerns is "quality of filtering". This can be defined in many ways, but one of the more important quality concerns is the ripple of the output voltage. This ripple is the residual effect of the peaks and valleys from the input square wave. It is never possible to eliminate them completely, but they can be reduced to any arbitrarily small amount. How small is small enough? That depends on the particular application, but here are some ideas.

One way we could determine the acceptable limit on ripple would be based on the resolution of any Analog-to-Digital converter we might be using. For example the ZX hardware has 10 bit (0-1023 counts) ADC hardware. Ripple of less than 1 part in 1024 would be invisible. If we are using a 5 volt system, then 1/1024 equates to about 5 millivolts. For a filter, the attenuation is usually measurement in "db", 1/1024 = -60.2 db. [db = 20 Log_{10}(Vout/vin)] If we were willing to accept 50mV of ripple, we would only need 40 db of attenuation. It would probably be a good idea to add a few extra db of attenuation just to be sure that the ripple does not exceed the upper limit. If you had a 16 bit ADC, then you would need to limit the ripple to better than 1/65536. This is –96db. With a few extra db thrown in, you would need a nice round –100db. This might be hard to do with a second order filter, but it is a piece of cake for a fourth order filter. A "quad" op-amp could allow up to an 8th order filter using the one device (plus 12 resistors and 8 capacitors). An 8th order filter will have a very steep drop-off in the voltage above the cut-off frequency (-48db/octave). This allows the use of a higher cut-off. See the discussion below regarding filter design software and yet further below regarding settling time.

What about the other constraints? Many books have been written about the pro's and con's of various designs of filters which will be summarized here. A good compromise in simplicity vs. quality is a "second-order" filter which delivers double the db value of attenuation (square the voltage value of attenuation) compared to the most simple single resistor and capacitor first-order filter. A Bessel filter gives better control of transients than a Butterworth filter. A multiple feedback topology also gives better control than single feedback topology (Sallen-Key). An active filter, using an op-amp as a part of the filtering process, will allow smaller components and still provide a low impedance output.

There are various filter design programs available free from semiconductor manufacturers that allow you to set the constraints, and it will give you the schematic for the active filter circuit. Filter-Pro from Texas Instruments was used as a major source for the following circuits, and can help you with variations in filter design, particularly higher order circuits using more than one op-amp stage.

Now we reach the question of the granularity of the duty factor. In general, the DAC clock should run at the highest possible frequency to allow the smallest possible capacitors. The current ZX devices have a 14.7456 MHz clock. This means that each step in the duty factor must be a minimum of 0.0678 uS. If we want 256 steps (0-255) in the duty cycle, then the period must be 256 clock cycles long. This equates to 17.29uS or 57,600 Hz. See the following table.

	Duty Factor Step Resolution				
	256	1,024	4,096	16,384	65536
Frequency	57,600 Hz	14,400 Hz	3,600 Hz	900 Hz	225Hz

Table 4 – Relationship Between Step Resolution and Frequency

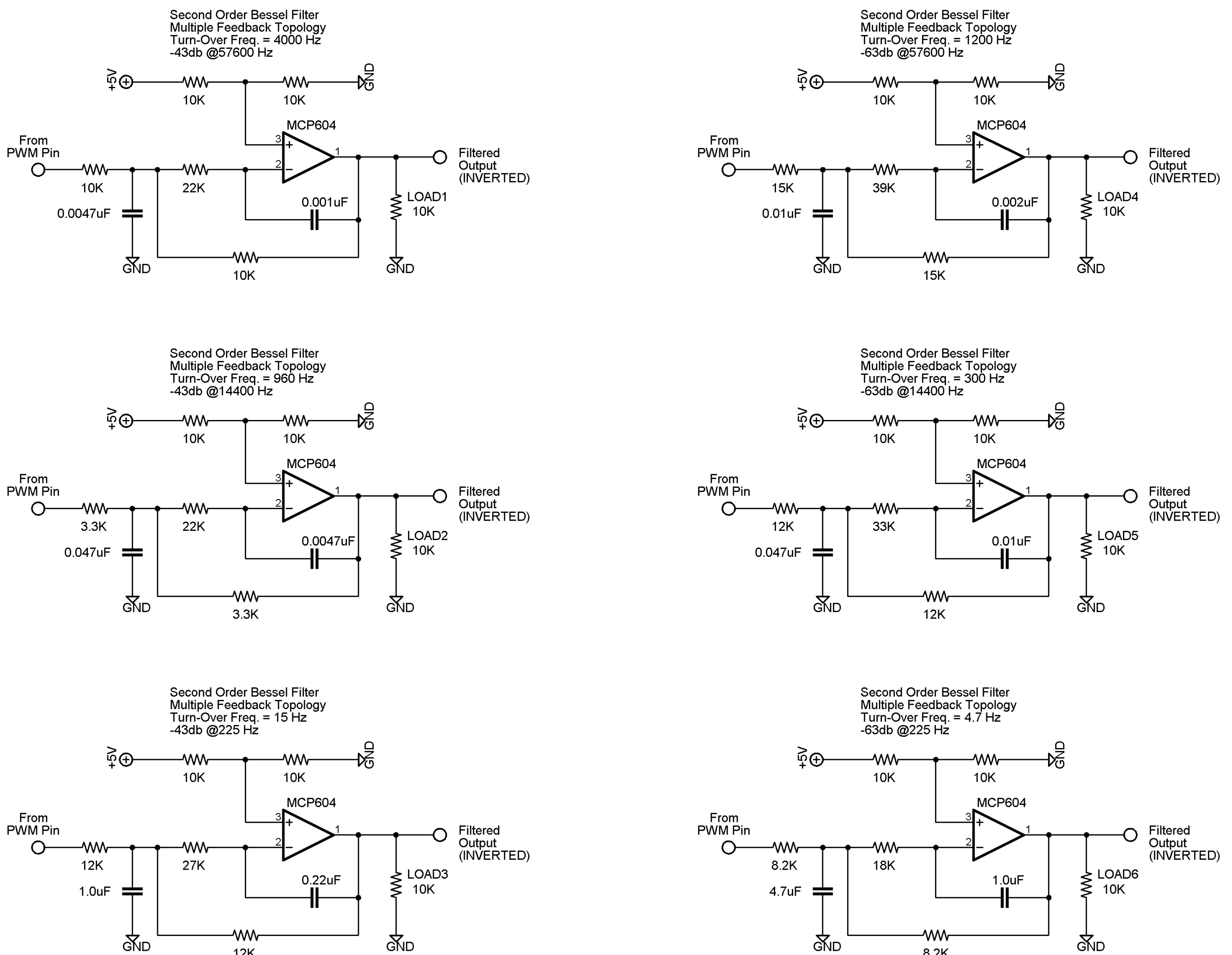
How many steps of in duty factor adjustment should be used? Two numbers that might make sense are either 256 or 65536 because those will fit in a byte or an integer respectively. But, 65536 steps might be an excessively large number considering that the ADC hardware of the ZX can only measure voltages with 10 bits of accuracy! Also, the highest step resolution must run at the rather low frequency of 255 Hz, making filtering more difficult. Nevertheless, some ADC hardware is fully capable of measuring with 16 bits of resolution, so 65535 steps may be necessary. [If one were to apply the Nyquist theorem, it might be best to run the DAC with double the duty cycle resolution as compared to the resolution of the voltage measurement hardware.]

The last issue is the "settling time". This is the time it takes the output to settle to a new pseudo-voltage after changing the duty factor. The settling time is proportional to the turn-over frequency (F_c) [Remember, the turn-over frequency is based on the number of steps in the duty factor as well as the amount of ripple in the output]. The settling time is also proportional to the accuracy required from the measurement of the pseudo-analog signal. The more accuracy that is needed the longer it takes to fully settle. The controlling equation is $0.6933 * N_{bits} / (2 * PI * F_c)$. Table 5 gives the settling time for various combinations. If you are using the ZX hardware to read the voltage, then use the 10-bit column for the settling time.

	Settling Time			
2nd Order Bessel Turn-Over Freq.	8 bits	10 bits	12 bits	16 bits
4.7 Hz (-63 db @ 225 Hz)	188 mS	235 mS	282 mS	376 mS
15 Hz (-43 db @ 225 Hz)	58.8 mS	73.6 mS	88.3 mS	118 mS
300 Hz (-63db @ 14.4 KHz)	2.94 mS	3.68 mS	4.41 mS	5.88 mS
960 Hz (-43 db @ 14.4 KHz)	0.92 mS	1.15 mS	1.38 mS	1.84 mS
1.3 KHz (-63 db @ 57.6 KHz)	0.68 mS	0.85 mS	1.02 mS	1.36 mS
4.0 KHz (-43 db @ 57.6 KHz)	0.221 mS	0.276 mS	331 mS	0.441 mS

Table 5 – Settling Times for Various Configurations

As this table shows, 8 bit resolution at 57.6 KHz has about 550 times faster settling time than 16 bits at 225 Hz. This allows much faster response time if the speed of the response is a critical issue. Allowing ten times as much ripple in the output will allow about 3 times faster response time.



Copyright © 2006 Elba Corp.

Figure 3 - Second Order Bessel Filters Tuned for Various Cut-off Frequencies

Exercise: Driving an H-Bridge

In the simplest view, an H-Bridge is a transistor circuit that has two inputs and can be used to drive a DC electric motor in both directions. Depending on which input is high, the motor will rotate one direction or the other. But, if both inputs are high simultaneously, a short circuit is created and the H-Bridge circuitry may be destroyed. When both inputs are low, there is no power to the motor at all.

In order to avoid the possibly destructive effects of having both inputs high simultaneously special purpose logic is often used. There are several commercially available H-Bridge driver IC's as well as more complex devices on a printed circuit. These will not be discussed in detail here.

Usually the H-Bridge circuits have two direction inputs as well as an "enable" input. There are various techniques used to drive the H-Bridge at less than 100% speed while preserving torque. The simplest technique of using PWM pulsing of the "enable" input, similar to the preceding DAC example, may be adequate in some situations. More often somewhat more complicated techniques are used.

One variant of the DAC-style PWM technique uses a constant frequency with only the duty factor changing for the middle of the speed range. At the higher and lower ends, frequency may also be changed so that the Off time is adjusted instead of the On time. Again, the "enable" input to the H-Bridge is pulsed. Experimentation may be necessary to determine the optimal frequency to give good motor torque.

A third technique that is reported to work well is "Locked Anti-Phase". This does not pulse the "enable" pin on the H-Bridge, but instead drives the motor forward and backward at high frequency so that it averages out. At 50% duty factor, the motor is driven equally forward and backward so it comes to a standstill, with braking. At 100% duty, the motor will turn at full speed forward. At 0% duty, it turns backward at full speed. If the "enable" pin is off, then the motor coasts to a standstill.

This technique seems to have the most promise due to positive reports of its actual operation. To achieve this without any extra external logic, a two-channel timer must be used. On the ATMega32-based ZX hardware, only Timer 1 has two channels. On the ATMega644, all 3 timers have two channels, Timer 0 is used by the RTC, but Timer 2 can be used with 8-bit precision. (Note: this may conflict with the software UART channels which also use Timer 2.)

The two channels are set up with opposite outputs and with opposite modes (Normal and Inverted). When the timer sets and clears the two channels, it will happen simultaneously and they will be locked in "anti-phase". If there is any error in the setup of the two channels, then there is the risk of driving both channels high at the same time, thereby destroying an unprotected H-Bridge.

It is probably better to use a single PWM output and run the signal straight to one H-Bridge input, and through an inverter to the other output. Technically, the inverter causes a slight propagation delay, on the order of 15 to 20nS (depending on the logic family). This means that for 15nS or so, the two H-Bridge inputs will have the same value. Probably this will not do any damage for such a brief interval, but it may be safer to send the PWM output to two 2-input XOR logic gates. The second input of one of the gates is connected to the positive voltage input, and the other gate has its second input connected to ground. In this manner one gate will always output the opposite value of the other for the same PWM input, and there will be equal propagation delay.

My source on this subject had good results with a PWM frequency of a few KHz. Commercial PWM devices frequently use tens of KHz. It is feasible to drive the H-Bridge with locked anti-pahse at 57,600 Hz with 256 steps of resolution (128 in each direction). Again some experimentation may show how motor torque changes across PWM frequencies.

Copyright © 2006 Elba Corp.

Programming

Here is a subroutine using the second PWM technique listed above. It will calculate the PWM frequency given a specific duty factor, based on the constants of minimum pulse width, and the middle-range frequency. There are practical limits to the duty factor. The relationship of duty factor and motor speed is probably not linear particularly at the extremes of the duty factor. It might be worthwhile to create a correction factor look-up table, or just avoid the use of rather high or low duty factors. A working example of this technique is found in the sample code for this application note.

```
'some arbitrary values.  Test others!
Private Const MiddleFreq as Single = 57600.0    'Hz
Private Const MinPulse   as Single = 3.47e-6    'seconds

Public Function CalcPWMFreq ( ByVal Speed as Single ) as Single
   If (Speed < (MiddleFreq * MinPulse)) then
       CalcPWMFreq = Speed / MinPulse
   ElseIf (Speed > (1.0 - (MiddleFreq * MinPulse))) Then
       CalcPWMFreq = (1.0 - Speed) / MinPulse
   Else
       CalcPWMFreq = MiddleFreq
   End If
End Function
```

Summary

In the preceding discussion and examples, it can be seen that PWM techniques have many uses. Programming the Timers to provide a wide variety of frequencies and duty factors is reasonably simple once the use of hardware registers is understood.

It is expected that interested readers will use this material as an overview of the subject matter and use the sample programs to provide only a starting point for further development tailored to specific needs.

Author

Anthony Rhodes is an amateur programmer with a special interest in microcontrollers. He has written several modules to interface support hardware to the ZX platform. Many of them actually work!

e-mail: support@zbasic.net **Web Site:** http://www.zbasic.net

Disclaimer: Elba Corp. makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The Company reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the Company or other parties is expressed or implied.

Copyright © Elba Corporation 2006. All rights reserved. ZBasic, ZX-24, ZX-40, ZX-44 and combinations thereof are trademarks of Elba Corp. or its subsidiaries. Other terms and product names may be trademarks of other parties.