ZBasic

Application Note

AN-221 Interfacing Rotary Encoders

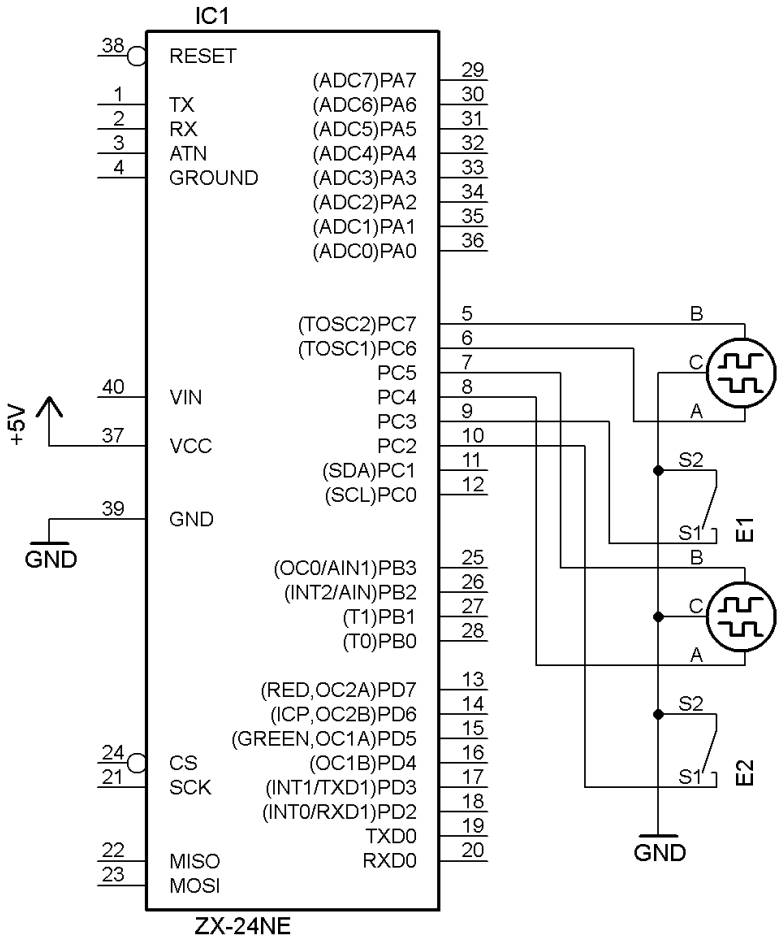
Introduction

This application note describes how to interface ZX devices with one or more rotary encoders. A rotary encoder such as the Bourns PEC12 (http://www.bourns.com/data/global/pdfs/PEC12.pdf) is a continuous rotation mechanical device that outputs a 2-bit Gray code (quadrature) signal that changes as the shaft is rotated. Encoders typically have 12 or 24 detents and may also include a SPST momentary push button.

This application note contains ready to use interface code for rotary encoders including a software switch debouncer and an example of how to use the interface. The code uses some of the more advanced features of the ZBasic language such as structures, based variables, aliased variables, and the "bit" subtype. The use of each feature is explained so that readers can try using these features in their own code.

Hardware Hookup

Three I/O connections are needed to connect a rotary encoder that includes a push button to a microcontroller. The example schematic below shows port C on an Oak Micros ZX-24ne device connected to two encoders named E1 and E2.



The Gray code output is available on pins A and B of the encoder. Pin C is the common and is normally connected to ground. The encoder switch is available on pins S1 and S2 with S2 connected to ground. The internal pull-ups on the ZX-24ne are used thus avoiding the requirement for additional resistors.

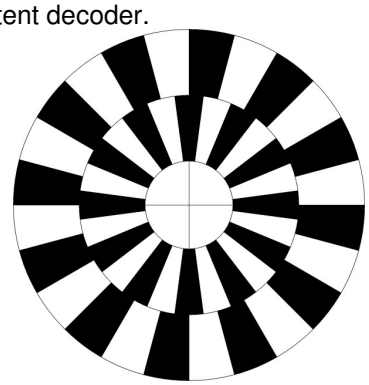
Published January 2009

Reading the Encoder

Encoders output a Gray code as depicted on its schematic symbol above. The two waveforms from the A and B pins result from turning the encoder clockwise or anti-clockwise. The direction determines the order of the 2-bit Gray code as shown below:

Clockwise Rotation ->
00 01 11 10 00
<- Counter Clockwise Rotation

If the output is 00 and goes to 01, then the encoder has moved one “tick” clockwise. If it goes from 00 to 10 then it moved one tick anti-clockwise. If there is contact bounce between two positions then the encoder will read clockwise and anti-clockwise ticks that cancel each other out. If the output goes from 00 to 11 then a position has been skipped and in this case the simplest solution is to simply ignore any ticks. Below is a diagrammatic view of what the tracks might look like on a 24 detent decoder.



With ZX devices there are several ways to read an encoder. One method is to use interrupts whenever the state of an I/O pin changes. A comparison of the previous state and current state can be used to determine what happened. The shortcoming of this approach is that you may be restricted to only one encoder on those ZX devices that have a limited number of interrupt pins or do not support pin change interrupts.

As an alternative the most direct approach is to sample the I/O pins and look for a change in the state. It is sufficient to sample the encoder every 5 milliseconds because the number of changes per second is relatively small. This technique is generally not applicable for optical encoders used for measuring the rotation of a wheel, which usually has a larger number of codes per revolution and spins much faster.

The algorithm to detect changes in the encoder makes use of the XOR (**⊕**) function and comparing the current state with the previous state. Assume states T_n, T_{n+1}, T_{n+2} etc where the inputs A and B change according to the direction of rotation and the state of A and B are labeled A_n, B_n, A_{n+1}, B_{n+1} etc. For clockwise (CW) rotation the following truth table can be written:

State	$\mathbf{A_n}$	$\mathbf{B_n}$	$\mathbf{A_n \oplus B_n}$	$\mathbf{A_{n-1}}$	$\mathbf{A_n \oplus A_{n-1}}$
T_n	0	0	0	0	0
T_{n+1}	1	0	1	0	1
T_{n+2}	1	1	0	1	0
T_{n+3}	0	1	1	1	1

And for counter clockwise (CCW) rotation the following truth table can be written:

State	$\mathbf{A_n}$	$\mathbf{B_n}$	$\mathbf{A_n \oplus B_n}$	$\mathbf{A_{n-1}}$	$\mathbf{A_n \oplus A_{n-1}}$
T_n	0	0	0	1	1
T_{n+1}	0	1	1	0	0
T_{n+2}	1	1	0	0	1
T_{n+3}	1	0	1	1	0

By observation the two XOR columns are the same the clockwise rotation and different for counter-clockwise rotation. This fact can be used to either increment or decrement the tick counter each time there is a change in either A or B. Note that if a clockwise rotation produces a negative tick count then it is likely that pins A and B have been swapped when connected to the ZX device.

Published January 2009

Translating this into what is required for ZX devices, a mechanism is needed to simultaneously read both A and B inputs, determine if there is a change and then increment or decrement the tick counter. The standard routine to read the value of an I/O pin (GetPin) cannot be used to simultaneously read two I/O pins. Fortunately it is possible by using the Register.PinX control program variable where X represents one of the standard AVR ports such as A, B, C etc. The following code illustrates the algorithm using port C where the encoder pins A and B are connected to I/O pins 6 and 7 respectively. The PIN register value is ANDed with a mask (&hC0) so that only the two relevant I/O bits are extracted.

```
Private tickCount as Integer
Private old_value as Byte

Public Sub CheckEncoders()
   Dim new_value as Byte
   Dim old_bits(0 to 7) as Bit Alias old_value
   Dim new_bits(0 to 7) as Bit Alias new_value

   new_value = Register.PinC And &hC0

   ' Check if something changed on the I/O port
   If (old_value Xor new_value) <> 0 Then
       ' update encoder tick count if something changed
       If (new_bits(6) Xor old_bits(6)) = (new_bits(6) Xor new_bits(7)) Then
           tickCount = tickCount + 1
       Else
           tickCount = tickCount - 1
       End If

       ' update the old value
       old_value = new_value
   End If
End Sub
```

The *new_value* byte variable contains the current state of the A and B encoder pins and the *old_value* byte variable contains the previous value. The ZBasic library routine GetBit() could be used to extract the required bit values from each byte value as follows:

```
If (GetBit(new_value, 6) Xor GetBit(old_value, 6)) = _
   (GetBit(new_value, 6) Xor GetBit(new_value, 7)) Then
```

A more readable mechanism is available by using the “bit” subtype and aliasing as shown in code above. A byte can be considered as an array of 8 bits indexed from 0 to 7. The “alias” feature overlays the bit array on top of a byte, thus simplifying access to each bit. The two alias definitions above for the *old_bits* and *new_bits* arrays do not use any additional RAM and the ZBasic compiler provides all of the “magic” to generate the correct code which uses the GetBit() function under the covers.

Encoder Software Interface

The file encoder.bas from the ZIP file accompanying this application note contains the encoder interface and software implementation. The encoder public interface consists of a structure named *Encoder* and the following public routines:

- InitEncoder()
- GetEncoderButton()
- GetEncoderTicks()
- ResetEncoder()
- UpdateEncoder()

The data structure and public functions are described in each of the sections below. Note that a semaphore is used for the last four routines to protect the private data from multiple updates by different tasks. Application note AN210 (Sharing Data Between Tasks) gives more details on using semaphores and other techniques for sharing data.

Published January 2009

Encoder Data Structure

The *Encoder* data structure is used help abstract out how the encoders are connected to the ZX device and remove any hard-coding of I/O ports and pin numbers. The structure is declared Public so it is visible outside of the encoder software interface and is a parameter to the *InitEncoder()* subroutine. Some members of the structure are declared Public and some are declared Private. This means that only the public members are available outside of *Encoder.bas* module whereas both the public and private members are available from within the *Encoder.bas* module. This mechanism provides some degree of protection and encapsulation for the private part of the Encoder data structure. Here is the full declaration of the 15 byte data structure:

```
Public Structure Encoder
   ' Public members help define the configuration of the encoder
   Public encoderPort as UnsignedInteger
   Public pinA as Byte
   Public pinB as Byte
   Public buttonPort as UnsignedInteger
   Public buttonPin as Byte

   ' Private members are mainly used for the encoder current state
   Private tickCount as Integer
   Private encoderMask as Byte
   Private oldEncoderValue as Byte
   Private buttonMask as Byte
   Private buttonCount as Byte
   Private buttonPressed as Boolean
   Private buttonPreviousState as Boolean
End Structure
```

As remarked by the comments in the code above, the public members of the data structure are used to define the configuration of the encoder and the private members are used to hold the current state. The public members are used to define:

- The address of the Register.PinX used for the encoder A and B pins.
- The port bit number used for pin A.
- The port bit number used for pin B.
- The address of the Register.PinX used for the encoder button. It can be a different port to the one used for the encoder A and B pins. A special constant named *NO_ENCODER_BUTTON* can be used to signify that the encoder button is not required.
- The port bit number used for the button.

Below is a typical encoder configuration for two encoders taken from the example code (AN221.bas) provided with this application note. In this example two encoders are connected to Port C on the ZX device but the button on the second encoder is not used.

```
Dim config(1 to 2) as Encoder

config(1).encoderPort = Register.PinC.DataAddress
config(1).pinA = 7
config(1).pinB = 6
config(1).buttonPort = Register.PinC.DataAddress
config(1).buttonPin = 3

config(2).encoderPort = Register.PinC.DataAddress
config(2).pinA = 5
config(2).pinB = 4
config(2).buttonPort = NO_ENCODER_BUTTON
config(2).buttonPin = 2
```

InitEncoder() Subroutine

The InitEncoder public subroutine is used to initialize one or more encoders. It takes two parameters, one is the address of an array of *Encoder* structures and the other is the number of elements in the array. Passing the address of the array means that the structure does not need to be copied over and avoids the requirement of

declaring either a fixed size array or using memory from the heap. Declaring a fixed size array "inside" the Encoder software interface is wasteful of memory as the maximum size is not known in advance.

Below is the relevant code that stores the InitEncoder() parameters and provides array addressing into the encoder array.

```
Private configAddress as UnsignedInteger
Private encoderData() as Encoder Based configAddress
Private encoderCount as Byte

Public Sub InitEncoder(ByVal configAddr as UnsignedInteger, ByVal size as Byte)
   encoderCount = size
   configAddress = configAddr
End Sub
```

The two private variables *configAddress* and *encoderCount* are used to store the parameters to *InitEncoder()*. The variable named *encoderData* provides direct array indexing into the array of *Encoder* data structures pointed to the *configAddress* variable. This is achieved by declaring it as a "based" variable giving the address (configAddress). Notice that the *encoderData* variable contains empty array indices as the size of the array is not known prior to runtime. Another way to look at this is that the *encoderData* declaration is providing the "type" of the data addressed by *configAddress*. No additional memory is used by the *encoderData* variable and the ZBasic compiler provides all of the "magic" to generate the correct code to access the data. Because arrays in ZBasic have a start index of 1 by default, the first *Encoder* data structure is accessed using the expression *encoderData(1)*.

The majority of the InitEncoder() routine is used to initialize the remainder of the *Encoder* data structure and setup the port (or ports) I/O pins used for the encoder(s) as inputs with a pullups. As a recap of the AVR architecture, to setup an input I/O pin with a pullup, the appropriate bit of the corresponding data direction register (DDR) is set to 0 and the bit on the PORT register is set to 1.

The code in the InitEncoder() subroutine makes use of the fact that the PIN, DDR, and PORT registers for a given I/O port are almost always at consecutive RAM addresses and therefore the DDR and PORT registers can be addressed given only the address of the PIN register. There is one known exception to this rule that is PORTF on the mega128 and special code is included for this case.

The InitEncoder() subroutine uses another "based" variable named *reg* to access a given DDR or PORT register. The *reg* variable is based on an address variable named *addr*. By varying the value of *addr*, the appropriate register can be used. Here is the relevant code from InitEncoder(), where *i* is the current index to the array of *Encoder* data structures:

```
Dim addr as UnsignedInteger
Dim reg as Byte Based addr

' calculate register mask
mask = Shl(1, encoderData(i).pinA) Or Shl(1, encoderData(i).pinB)

' set DDR and PORT registers (address based on PIN register)
addr = encoderData(i).encoderPort + 1
Call SetBits(reg, mask, &H0)
addr = addr + 1
Call SetBits(reg, mask, &HFF)
```

Similar code is also used to setup the mask and DDR/PORT registers for the encoder button and can be found in Encoder.bas module in the ZIP file associated with this application note.

The final step of InitEncoder() subroutine is to initialize the encoder tick count by using the UpdateEncoder() and ResetEncoder() routines as shown below:

```
 ' initialize encoder tick counters
Call UpdateEncoder()
For i = 1 to encoderCount
    ResetEncoder(i)
Next i
```


Published January 2009

ResetEncoder() Subroutine

The ResetEncoder() subroutine is used to simply reset the tick count for a given encoder.

UpdateEncoder() Subroutine

The UpdateEncoder() public subroutine is used to update the tick counts for the configured encoders and buttons. It should be called at least every 5 milliseconds. The core algorithm for this routine has already been described in previous section titled “Reading the Encoder”. The main difference in the final implementation as shown below is that the fields in the *Encoder* data structure are used to reference the port PIN register, mask, and I/O bit numbers.

In this case the RAMPeek() system library function is used to read the PIN register. The end result is the same as using a based variable and is shown as an alternative implementation. The PIN register address is taken from the *Encoder* structure rather than using a hard-coded system variable such as Register.PINC.

```
Public Sub UpdateEncoder()
   Dim new_value as Byte, old_value as Byte, i as Byte
   Dim old_bits(0 to 7) as Bit Alias old_value
   Dim new_bits(0 to 7) as Bit Alias new_value

   For i = 1 to encoderCount
       new_value = RamPeek(encoderData(i).encoderPort) And encoderData(i).encoderMask
       old_value = encoderData(i).oldEncoderValue

       ' Check if something changed on the I/O port
       If (old_value Xor new_value) <> 0 Then
           ' Update encoder tick count if something changed
           Dim pinA as Byte
           PinA = encoderData(i).pinA
           If (new_bits(pinA) Xor old_bits(pinA)) = _
              (new_bits(pinA) Xor new_bits(encoderData(i).pinB)) Then
               encoderData(i).tickCount = encoderData(i).tickCount + 1
           Else
               encoderData(i).tickCount = encoderData(i).tickCount - 1
           End If
       End If

       ' Save the old value for next time
       encoderData(i).oldEncoderValue = new_value
   Next i
End Sub
```

GetEncoderTicks() Function

The GetEncoderTicks() function as shown below returns the current value of ticks for the encoder and resets the tick count back to 0. The calling code determines what to do with the tick count and usually adds it to some kind of accumulator.

```
Public Function GetEncoderTicks(ByVal i as Byte) as Integer
   GetEncoderTicks = encoderData(i).tickCount
   encoderData(i).tickCount = 0
End Function
```

The example code for this application note shows how the speed of rotating the encoder can be used to proportionally affect a value i.e. the faster the encoder knob is rotated, the faster the value changes. This can be very useful for quickly changing a value that has a large range but yet still have fine control over the value.

In a given time interval, the faster the encoder is rotated, the higher the tick count. A scaling factor can then be applied to that value depending on its magnitude. Here is an example scaling routine provided in example AN221.bas that applies a simple proportional factor to the encoder tick count:

```
Private Function scaleTickCount(ByVal tickCount as Integer) as Integer
   If abs(tickCount) <= 2 Then
```

```
            scaleTickCount = tickCount
        ElseIf abs(tickCount) <= 3 Then
            scaleTickCount = tickCount * 2
        ElseIf abs(tickCount) <= 4 Then
            scaleTickCount = tickCount * 4
        ElseIf abs(tickCount) <= 6 Then
            scaleTickCount = tickCount * 8
        Else
            scaleTickCount = tickCount * 16
        End If
    End Function
```

And here is an example usage for the scaling routine. The range for the value *v1* is between 0 and 300. The Max() and Min() system library routines are used to keep the value within the range no matter how far the encoder is rotated.

```
    Dim e1 as Integer
    e1 = GetEncoderTicks(1)

    If e1 <> 0 Then
        v1 = Min(Max(scaleTickCount(e1) + v1, 0), 300)
        Debug.Print "Value 1:"; v1
    End If
```

GetEncoderButton() Function

The GetEncoderButton() function as shown below returns the current state of the button in the buttonPressed variable and if the button state changed since the last call to the routine. The current and previous state of the encoder button is stored in the *Encoder* data structure.

```
Public Function GetEncoderButton(ByVal i as Byte, ByRef buttonPressed as Boolean) as Boolean
   GetEncoderButton = FALSE
   buttonPressed = encoderData(i).buttonPressed
   If encoderData(i).buttonPreviousState <> buttonPressed Then
       GetEncoderButton = TRUE
       encoderData(i).buttonPreviousState = buttonPressed
   End If
End Function
```

The current state of the button is decided using a ZBasic implementation of a software debouncing function as described on page 19 of "A Guide to Debouncing" (see http://www.ganssle.com/debouncing.pdf). This function is modified from the original in three main ways as shown in the source code below:

- The debouncing is based on a counter rather than a timer. A counter is used for debouncing to avoid the issues of trying to setup a timer interval in ZBasic. The debounce initial counter values for pressing and releasing a button are configured depending on whether a ZVM or native mode ZX device is used.
- The configuration and status of the encoder button is used from the *Encoder* data structure.
- The changed value variable is not used and is returned by GetEncoderButton().

```
Private Sub debounceEncoderButton(ByVal i as Byte)
   Dim rawState as Boolean
   Dim count as Byte

   rawState = ((RamPeek(encoderData(i).buttonPort) And encoderData(i).buttonMask) = 0)
   If rawState = encoderData(i).buttonPressed Then
       ' set the counter for the given state
       If encoderData(i).buttonPressed Then
           count = DEBOUNCE_RELEASED_COUNT
       Else
           count = DEBOUNCE_PRESSED_COUNT
       End If
   Else
       ' count is finished so record button state change
       count = encoderData(i).buttonCount - 1
       If count = 0 Then
```

```
            encoderData(i).buttonPressed = rawState

            ' reset the counter for the given state
            If rawState Then
                count = DEBOUNCE_RELEASED_COUNT
            Else
                count = DEBOUNCE_PRESSED_COUNT
            End If
        End If
    End If

    ' save the count in the encoder structure for the next invocation
    encoderData(i).buttonCount = count
End Sub
```

The debouncing private routine is called from the UpdateEncoders() function for each available encoder button as shown below:

```
        If encoderData(i).buttonPort <> NO_ENCODER_BUTTON Then
            Call debounceEncoderButton(i)
        End If
```

Using the Encoder Software Interface

The AN221.bas that is part of the associated ZIP file with this application note gives an example of how to use the encoder software interface. For this example two encoders are used and the button on the second encoder is not used. The hardware and corresponding software configuration have already been given in earlier sections of this application note.

To use the encoder a separate task is used to call the UpdateEncoder() function as often as possible as shown below:

```
        Private Sub processEncoders()
            Do
                ' update value of rotary encoders and buttons
                Call UpdateEncoder()

                ' let other tasks do some work
                Call Sleep(0)
            Loop
        End Sub
```

The main task is used to query the two encoder tick counts, button status and process the results. In this example the button is used to switch between two modes 0 and 1. For mode 0, each encoder output is scaled and restricted within a range so that a clockwise rotation cannot exceed a maximum and a counter-clockwise rotation cannot be less than a minimum of zero. For mode 1, only one encoder output is used to continuously rotate within a range of 0 to 4 regardless of the direction of rotation, i.e. that is there is no minimum value and the next value after a 0 is a 4 for a counter-clockwise rotation. Here is part of the code from AN221.bas for the second encoder:

```
          ' process encoder 2 value
        If e2 <> 0 Then
            If mode = 0 Then
                ' update value 2 with a scaled tick count, within limits
                v2 = Min(Max(scaleTickCount(e2) + v2, 0), 200)
                Debug.Print "Value 2: "; v2
            Else
                ' update menu selection value
                m2 = (e2 + m2) mod 5
                ' deal with going negative
                If m2 < 0 Then
                    m2 = m2 + 5
                End If
                Debug.Print "Menu option: "; m2
            End If
        End If
```

An example use of these two modes is for a digital power supply. One encoder is used for voltage setting and the other for current setting. Pressing the encoder button switches into a menu mode and now rotation of the other encoder is used to switch menus. Feedback of the mode, menu selection, voltage, and current is given on a LCD display as shown in the photographs below. In this way a pair of encoders can replace a multitude of buttons and simplify a device user interface.

Power Supply with 2 Rotary Encoders

Loading in a saved Voltage and Current Setting

Default mode with variable Voltage and Current
(Current limit is 150mA, actual is 10mA)

Because the processing in the main task may take some while it is wise every so often to give control back to the processEncoders() task so that no encoder ticks or button changes are missed. The simplest way to do this is to use the ResumeTask system library function. An extract of the code is shown below:

```
#if Option.TargetCode="ZVM"
Private Const taskStackSize as Integer = 70
#else
Private Const taskStackSize as Integer = 100
#endif
Private taskStack(1 to taskStackSize) as Byte

' call task that keeps the encoders updated
CallTask processEncoders(), taskStack

Do
    ' ResumeTask for the processEncoders task can be called
    ' at any time as a way of yielding control to that task
    Call ResumeTask(taskStack)

    ' No more work to do so go to sleep for a while. If there was
    ' more work such as updating a LCD display, that could be done
    ' here, interspersed with calls to ResumeTask as necessary.
    Call Sleep(10)
Loop
```

Here is some sample output from the AN221.bas example program. It starts in mode 0. Encoder 1 is rotated to value 51 and encoder 2 is rotated to its end stop of 200. Then mode 1 is selected using encoder 1 button and encoder 2 is rotated until item 3 is selected. The encoder 1 button is then pressed to return to mode 0.

```
Mode 0 (Input)
Value 1: 2
Value 1: 8
Value 1: 24
Value 1: 40
Value 1: 46
Value 1: 48
Value 1: 49
```


Published January 2009

```
Value 1: 50
Value 1: 51
Value 1: 52
Value 1: 51
Value 2: 78
Value 2: 84
Value 2: 100
Value 2: 140
Value 2: 200
Value 2: 200
Mode 1 (Menu)
Menu option: 1
Menu option: 2
Menu option: 3
Mode 0 (Input)
```

Summary

This application note addresses the problem of interfacing a sophisticated input device to a ZX device. The Encoder.bas module provides a reusable software interface for rotary encoders that is hardware independent and could be used directly in production level code. Using rotary encoders opens up new possibilities for user-friendly hardware devices and tools.

Author

Mike Perks is a professional software engineer who became interested in microcontrollers. Mike has written a number of articles, projects and application notes related to ZBasic, BasicX and AVR microcontrollers. Mike is also the owner of Oak Micros which specializes in AVR-based devices including his own ZX-based products. You may contact Mike at mikep@oakmicros.com or visit his website http://oakmicros.com.