Today it is rare to find much in the way of new ham radio equipment that doesn't have a microprocessor tucked away inside which controls just about every function. This article is intended to introduce you to a simple-to-use processor and provide actual examples of circuits and programs that should tempt you to exercise your imagination in creating your own homebrew projects.

The BasicX-24™

Add An Easy-To-Use Microprocessor To Your Homebrewing Skillset

BY DENNIS NENDZA,* W7KMV

Many of the bells and whistles now standard on radio equipment would not be possible without little computers called *microprocessors* on board. Admittedly, the programs written to orchestrate the latest big-buck transceivers are very complex and generally not open to examination or modification. Few hams have the ability to dig in and successfully change such devices. However, not all processors are difficult to use.

There actually are a few processors, or controllers as they are also known, that were created to be easily understood and used. A popular small controller that has been around for a number of years is the Basic Stamp II™[1] made by Parallax, Inc. It is programmed in a fairly simple English-like language called BASIC.[2] However, there are significant constraints to this particular device, which include limited program size, speed, and a very small workspace for variables, which are the places where information is stored. Another impediment when considering interfacing with analog circuits is this controller's inability to directly read voltage levels.

NetMedia, Inc.[3] addressed these limitations with the advent of its BasicX-24™ (BX-24) controller. With vastly superior performance for nearly the same price, this little gem has not received the same attention as the Basic Stamp II™. It is a remarkable computer system on a chip. After building what I felt was a rather complex application for a messaging APRS tracker[4] using this processor, I felt more hams should be exposed to this very useful tool. I approached NetMedia for this article and received samples of its processor family, which are employed in the examples. Let's see what it takes to fire up one of these devices and how to make it do our bidding.

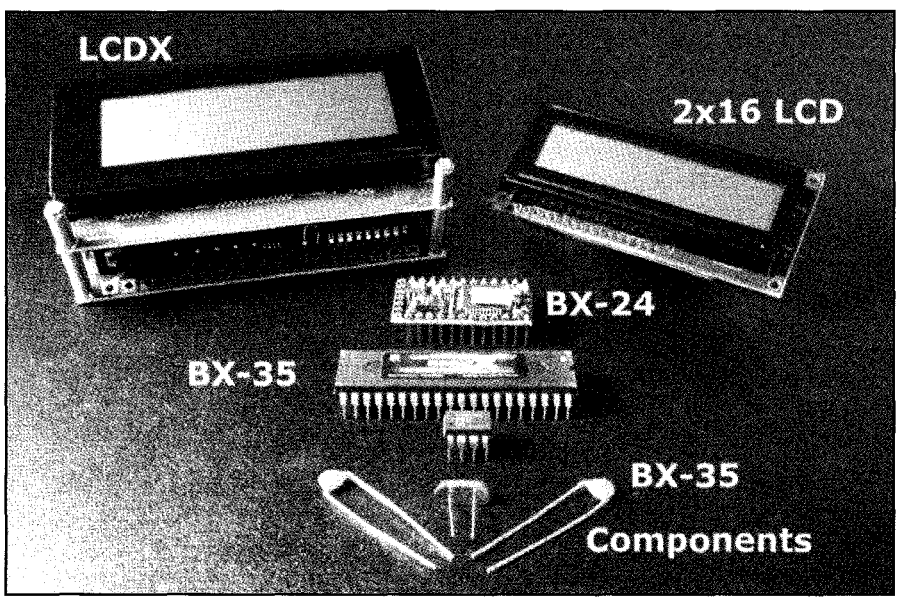

Fig. 1– The BX-24/35 family. The complete BX-24 one-chip system is surrounded by the BX-35, which sports more I/O lines and an off-chip EEPROM and oscillator; the LCDX, a complete computer with 4×20 character display with sounder, 16-key keyboard decoder, relay drivers, and adjustable ADC inputs; and the 2×16 serial data LCD display.

Support System

Most small processors are "spoon-fed" their programs from another computer, typically the ubiquitous personal computer. Any PC running Windows 95™ or better will suffice to develop a program and upload it to the BX-24. The same computer can also be used as an

**34 E. MacIver Place, Tucson, Arizona 85705*
e-mail: <W7KMV@arrl.net>

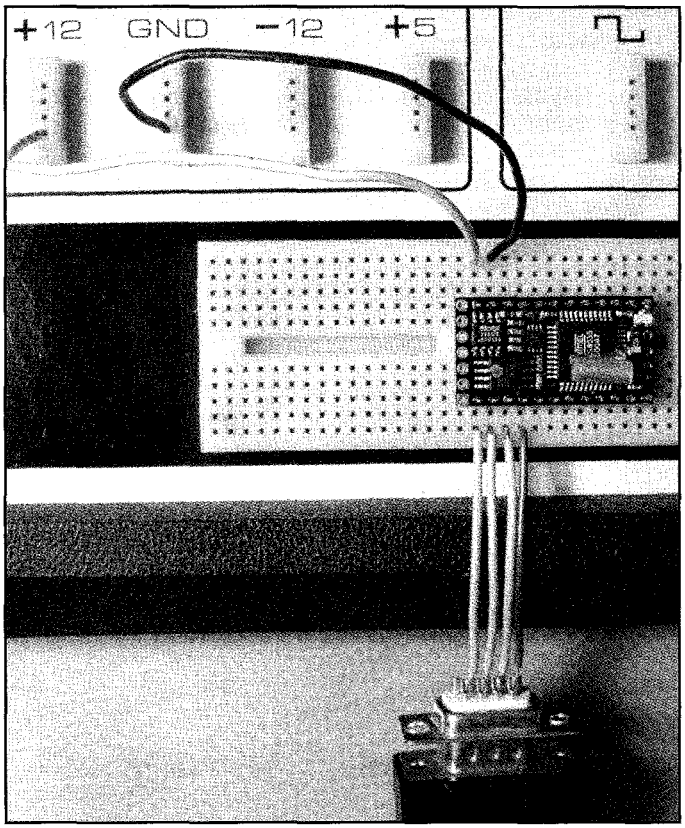

Fig. 2– The BX-24 basic connections; a 5.5–15 VDC source and serial port. Onboard red and green LEDs can be programmed for simple visual I/O tests.

output device for data coming out of the controller if we choose to keep the computers linked after uploading the program. You will need a machine with a serial port for communication. This is worth noting, as some computers are now being sold with only USB ports as the replacement for the once-standard serial connection. USB to serial converters *do* exist[5] should you be fortunate enough to have a late-model machine with no serial port.

The BX-24 computer arrives as a miniature collection of surface-mount components pre-assembled on a 24-pin carrier. It plugs into a solderless breadboard or perf board just like any other 24-pin chip and is ready to go once power and the serial port are connected. It is one of several physical implementations available which share the same programming and development environment. Fig. 1 shows the BX24/35 family of devices currently available. The BX-35 has more I/O pins and uses an outboard crystal oscillator. Fig. 2 shows the basic setup of a BX-24 on a breadboard, and fig. 3 is the schematic. In addition, prepared experimenter boards available from NetMedia and Peter H. Anderson[6] provide room for additional circuitry of your own design. I find that nothing beats a good-size solderless breadboard (or two) for trying out the latest idea.

In fig. 3 you can examine the pin-out diagram of the processor, showing a connection to power and a serial path to the PC . There are two ground pins. Pin 23 is to be used for the negative power connection, and Pin 4 is used as common for the serial I/O connection to the PC. Pin 24 accepts +5.5 to 15 VDC, which is used to power the processor. A small on-board voltage regulator allows for the wide range of input voltage. If you have regulated 5 VDC and wish to use *only* that to power the device, then connect it to Pin 21 in lieu of any connection to Pin 24. Pins 1–3 comprise the processor serial out, serial in, and attention signals, respectively. The additional circuitry in fig. 3 is used and explained later in this article.

Now that the hardware is connected, we need to establish a means of talking to the processor and sending it the programs we are going to develop on the PC. On the BasicX-24™ website (see note 3), there is a download page where both the documentation and the programming environment may be obtained. I leave it to the reader to go through the install process and peruse the documentation.

When starting the BasicX 2.10 development environment, you will notice that it consists of two major sections. The first part that opens a window is the "Downloader." From this window we can set parameters that relate to the specific model of the processor that we are using (there are three variants), the download port, and even stop and start the processor via the serial connection. For now, let's make sure the following menu items are set:

File-Set Starting Directory: Choose a directory where you want your program to reside.

Processor-Processor Type: Set to the processor type you have.

I/O Ports-Download Port: Set to the serial port to use on your computer and **OPEN** it.

At this point we can invoke the editor window by selecting **File-Open Editor**. This new window is the one in which we will build the example applications in this article. Select **File-New Project** to get started and set the project type to "**General Purpose**," project name to "**CQBlink**," the module name to "**CQBlinker**," and click OK. You will notice that the window now has the project name in the title bar and several lines of program code already included inside.

It is important to understand the difference between a project, a module, and subroutines. Briefly stated, a project contains all files and programming that relate to a particular undertaking, such as a data logger, a GPS decoder, a digital thermometer, or a digital SWR meter. A project contains one or more modules. A module name shares the file name in which the module is stored. Within modules you may define and use subprograms which may be made accessible to other modules by declaring them public, or inaccessible by declaring them private. Confusing? Don't be too concerned at this point, as the examples will help to get us past all this. We will employ one module in one file with several subroutines. Now, enter the program in Table I into the editor window or pick up a copy of it from the <members.cox.net/desertlavender/bxprojects.htm> web site[7], where all the example programs can be found. If you download the program, you can just copy it and paste into the editor window. Delete any duplicate lines at the beginning or end of the window following the paste. It is good practice while typing in a program to periodically perform a **File-Save Project** operation to ensure that our work is saved as we progress. Note that complete programs are saved as projects and have several components described by the "projectname.bxp" file. The editor will only open ".bxp" files.

With all the typing complete, it's time to let the BX-24 compiler scan our program and see if there are any errors that it can find. Select the **Compile-Compile** menu item or press the **F4** function key to have the program scanned and compiled. When errors are found, the line number and error type will be displayed. This compiler does not supply a list of errors, since it stops on the first error found. Cleaning up the compile errors is an iterative process in which you fix the line in

question and compile again. If the compile finds no errors, a status line at the bottom of the editor window will indicate success with a "**Compiled OK**" message and show the length of the program as well as how much RAM is used for variable storage.

The next step is to transfer the compiled program to the processor. This is accomplished from the first window we encountered, the Download Window. The **Processor-Download** menu item will initiate the transfer if the download port has been specified and opened and there is a powered-up BX-24 correctly connected at the other end of the serial cable. Following a successful download, it is necessary to restart the processor. This can be done by either selecting **Processor-Execute**, clicking on the green light of the traffic signal icon, momentarily grounding Pin 22 on the processor, or cycling the processor power. If all went well, you should see a red LED blinking out "CQ" on the controller.

FREQOUT!

A variant of this visible code sender adds sound, which requires the addition of a small piezo speaker or simple headphone. Fig. 3 shows how to add sound output to the processor by connecting a sound-generating component to a

```
Option Explicit
'-----------------------------------------------------------
' This program blinks the red LED on the BasicX-24 chip to send a visible CQ.
'-----------------------------------------------------------
' Define useful constants
Const RedLED    As Byte = 25            'Red LED's equivalent pin number
Const LEDon     As Byte = 0             'To set an OFF condition
Const LEDoff    As Byte = 1             'To set an ON condition
Const SpeedConstant  As Single = 1.20   'Constant used to compute element time
' SpeedConstant/Speed(WPM)=element time
Const Speed     As Single = 13.0        'Set speed at 13 wpm
' Define variables
Public TDit     As Single               'Dit time
Public TDah     As Single               'Dah Time
Public Ie       As Single               'Inter-element time (Dit time)
Public Ic       As Single               'Inter-character time (3x Dit time)
Public Iw       As Single               'Inter-word time (7x Dit time)
'-----------------------------------------------------------
Public Sub Main()
Call RedOff                             'Ensure red LED off
Call Delay (1.0)                        'Wait one second
TDit=SpeedConstant/Speed                'Set Dit time
TDah=TDit*3.0                           'Set Dah time
Ie=TDit                                 'Inter-element time
Ic=TDah                                 'Inter-character time
Iw=TDit*7.0                             'Inter-word time
Do
   ' Send a "C" via the red LED
   Call Dah                             'Call the Dah subroutine
   Call Dit                             'Call the Dit subroutine
   Call Dah
   Call Dit
   Call InterChar
   ' Send a "Q" via the red LED
   Call Dah
   Call Dah
   Call Dit
   Call Dah
   ' Send an inter-word delay
   Call InterWord
Loop                                    'Go back and continuously send
End Sub
'-----------------------------------------------------------
' Subroutines follow
'-----------------------------------------------------------
Public Sub RedOn()
   ' Turn red LED on
   Call PutPin(RedLED, LEDon)           'Turn red LED on
End Sub
'-----------------------------------------------------------
Public Sub RedOff()
   ' Turn red LED off
   Call PutPin(RedLED, LEDoff)          'Turn red LED off
End Sub
'-----------------------------------------------------------
Public Sub Dah()
   ' Turn red LED on for TDah time and follow with Ie time off
Call RedOn                              'Light it up
Call Delay (TDah)                       'Wait TDah time
Call RedOff                             'Turn it off
Call Delay (Ie)                         'Follow with inter-element time off
End Sub
'-----------------------------------------------------------
Public Sub Dit()
   ' Turn red LED on for TDit time and follow with Ie time off
Call RedOn                              'Light it up
Call Delay (TDit)                       'Wait TDit time
Call RedOff                             'Turn it off
Call Delay (Ie)                         'Follow with inter-element time off
End Sub
'-----------------------------------------------------------
Public Sub InterChar()
   ' Perform inter-character delay assuming last element was followed by an inter-element delay
Call Delay (Ic-Ie)                      'Wait
End Sub
'-----------------------------------------------------------
Public Sub InterWord()
   ' Perform inter-word delay assuming last element was followed by an inter-element delay
Call Delay (Iw-Ie)                      'Wait
End Sub
```

Table I– CQBlinker program listing.

processor output pin (Pin 15 in this example). The program alterations to create an audible tone in addition to the visible LED are shown in Table II.

The FREQOUT system library routine used in this example allows you to specify a processor output pin and two frequencies to output simultaneously for a given length of time. To produce a single tone, just specify a zero for the unused one. For the forward thinking among you, it may have just become apparent that DTMF (dual tone multi-frequency) audio for phone dialing or control purposes can be created this way, and indeed it can. Referring back to our CW sending example, the time spent sending a tone to the speaker can replace the delay of our LED-only example. If you were to use the processor as an audio-generating device for connection to a transmitter or audio amplifier, it would be a good idea to smooth out the harsh-sounding square-wave with some capacitance in parallel with the output and use a small audio transformer to couple to the external device's input.

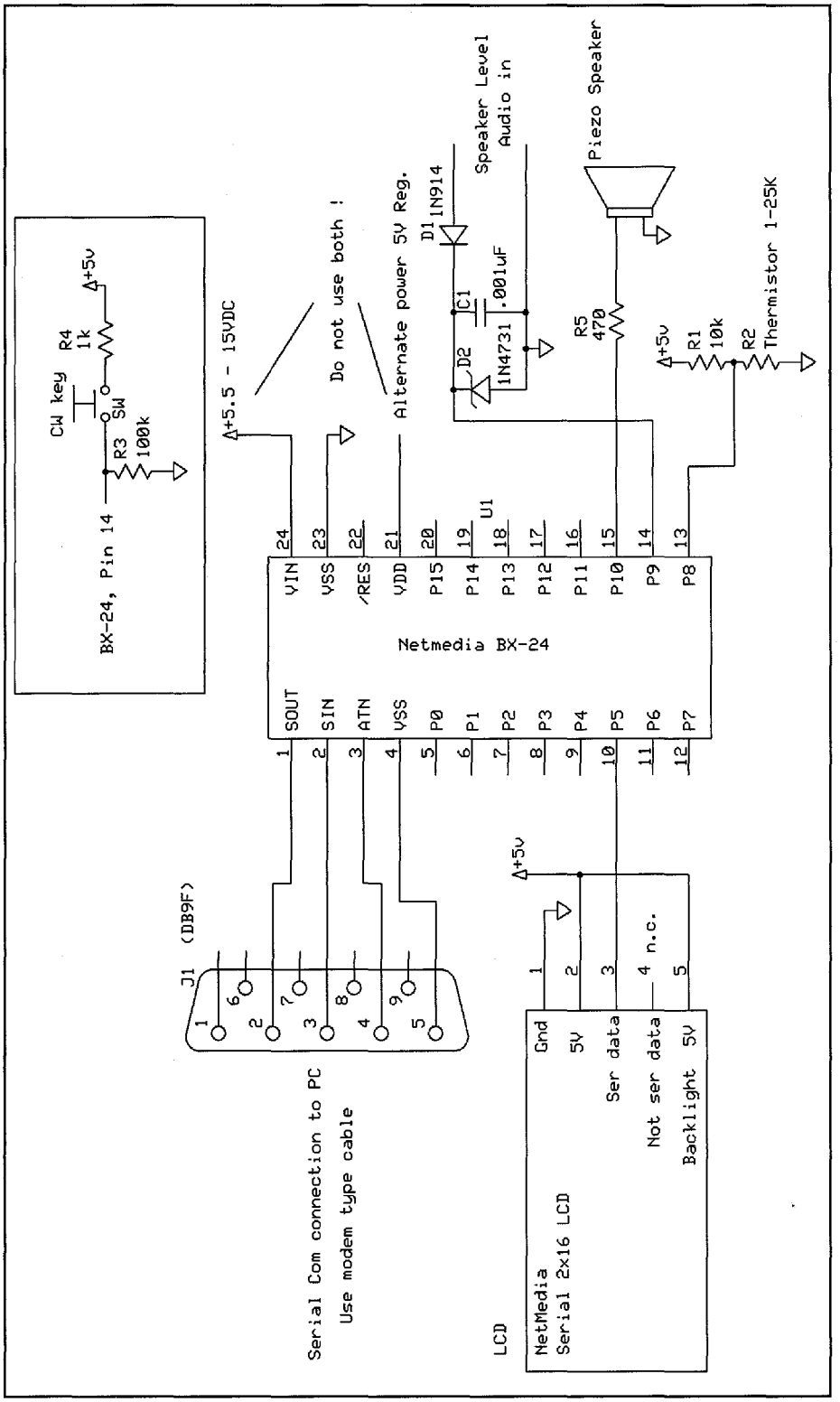

Fig. 3– BX-24 connections referenced in the examples. Zener diode, D2, is 4.3 V. D1 can be any small signal diode. R2 is a RadioShack 271-110A 10K ohm thermistor. The piezo speaker can be a headphone or All Electronics PE-38.

A Simple Morse Decoder

When considering how fast this little processor could copy Morse code, I did some back-of-the-envelope figuring and estimated that it would be lucky to hit 20 to 30 wpm. I thought the timing would become skewed by the program's inability to execute fast enough to keep up with the next element at higher speeds. Boy, was I surprised when it flawlessly copied an audio source timed at 99 wpm! At 99 wpm the dits are about 12 milliseconds long—not much time for a small processor running a tokenized[8] BASIC programming language to keep track of things.

Keep in mind that if you wrote a program to strictly follow all the timing rules for CW, it wouldn't work very well. There is a great deal of variation in CW, as everyone has a slightly different way of sending it or tuning their keyer. With straight keys the timing is most variable. The best solution is to follow the "heuristic rules" that we use in our heads when we copy CW manually. These simple rules are:

1. If an element is about twice as long as or longer than a previous dit, then it is probably a dah.

2. Following each element, recompute a new dit and dah time based on that element.

3. If a space is longer than 2.5 dits, it is probably a character space.

4. If a space is longer than 2 dahs, it is probably a word space.

Rule 1 makes the element decision easy, and best of all, it works. Rule 2 helps the program track changing speeds. Abrupt changes in speed are tough to deal with, but by averaging the last dit or dah with the current one, the program adjusts quickly to speed changes. Rules 3 and 4 seem to work pretty well for a variety of spacing experienced.

To set up the processor to work with the program we need an external keying

source. It can be a straight key, set of paddles for an electronic keyer, or an audio source. It all comes down to the need to provide the processor with a logic 1 and 0 to indicate whether the key is down or up. My first experiment was with a straight key, and it works pretty well to demonstrate the effectiveness of the algorithm or method of decoding CW.

The program to copy CW is too long to print here, but it is available at the download website (see note 7), and a few words are in order to explain how it is organized. Common characters and punctuation are six elements or less. By noting this, I started to develop some ideas on how to copy a character and translate it into a printable ASCII character. My technique may not be original, but I am unaware of another example that uses it. By considering dits as zeros and dahs as ones, I would shift in a 0 or 1 on the right side of a byte, which is 8 bits wide. Here's how the idea works: Following a silence that indicates a character or word break, we establish a new character byte with the

CW Character Element Input Byte Construction Process

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CW example received character is "P" (. _ _ .)
0	0	0	0	0	0	0	1	Immediately after character or word space
0	0	0	0	0	0	1	0	After first dit recognized - insert 0
0	0	0	0	0	1	0	1	After first dah recognized - insert 1
0	0	0	0	1	0	1	1	After second dah recognized - insert 1
0	0	0	1	0	1	1	0	After last dit recognized - insert 0
0	0	0	0	0	0	0	1	Character or word space occurs - lookup character, reset input byte

Resulting Byte: 00010110 binary, or 22 decimal

Resulting value is used to index the code table and pick up the ASCII representation of the received character

22nd entry in table

Printable codes for the ASCII characters

CODETABLE255

94	'	^
69	'	E
84	'	T
73	'	I
65	'	A
78	'	N
77	'	M
83	'	S
85	'	U
82	'	R
87	'	W
68	'	D
75	'	K
71	'	G
79	'	O
72	'	H
86	'	V
70	'	F
94	'	^
76	'	L
94	'	^
80	'	P
74	'	J
66	'	B
88	'	X
67	'	C
89	'	Y
90	'	Z
81	'	Q
94	'	^
94	'	^

(continues)

Fig. 4– The steps that occur as a CW character is received are outlined in the process shown here. CODETABLE255 is constructed so that a "received" CW character can be directly interpreted as an equivalent ASCII character by adding its value to the base of the table to perform the lookup.

Changes to CQBlinker: Existing lines(bold type) are shown with action such as delete or insert after.

a visible CQ
Delete, then replace with
a visible and audible CQ.

Const LEDoff	**As Byte = 1**	**To set an ON condition**
Insert after—		
Const Tone	As Integer = 1000	Cw tone is 1000 Hz
Const TonePin	As Byte = 15	Send tone on pin 15

Call Delay (TDah)	**Wait TDah time**
Delete, then replace with	
Call FreqOut (TonePin, Tone,0,TDah)	Wait TDah time while sending tone

Call Delay (TDit)	**Wait TDit time**
Delete, then replace with	
Call FreqOut (TonePin, Tone,0,TDit)	Wait TDah time while sending tone

Table II– Modifications to CQBlinker to add sound.

value "1" in it, a binary "00000001." This 1 will continually be shifted to the left in the byte each time a dit or dah is detected and the respective "0" or "1" is shoved into the right side of the byte. Look at fig. 4 to see the steps to building the character byte using the CW character "P" (·– –·) as an example.

To keep the program simple and fast, I broke the decoding technique into several parts. The first part is determining when a change of state occurs. Either the key has closed or it has opened. A change of state makes it easy to time the length of the previous state. When the state changes, we read the timer of the processor that keeps time in approximately 2-millisecond ticks and compute the length of the previous state. This leads to the second part, which looks at key-up or key-down intervals. If the current state is key up, then the previous must have been key down and it is time to compare the key-down time to the current dit length. This yields a dit or dah and the character byte is modified accordingly. Conversely, if the current state is key down, then the previous was key up and we check to see if that represented the space between an element, character, or word. If it's a character or word space, we look up the ASCII[9] representation of the received CW character we've built and send it to the display device. This lookup requires a table built to allow the numerical representation of the received CW character to be used as an index which, when added to the table's beginning, points to the corresponding ASCII printable character.

To move from decoding CW sent by a straight key to off-the-air signals requires some extra work outside the processor. The simplest method, given enough audio power, is to rectify the audio and provide a bit of filtering with a capacitor. To protect the processor we have to ensure that the level of the filtered audio does not rise above the 5V logic level. Fig. 3 also includes the simple audio rectification filter and Zener diode input protection circuit. I used this to "listen" to 5 – 99 wpm CW audio (from powered speakers) that was created on the PC by the Wavgen program[10], which creates a playable audio file. This rudimentary circuit will work on quiet receiver passbands with no adjacent signals audible. It does require speaker-level audio to generate enough voltage for the detector. A more sophisticated circuit employing narrow audio filtering and automatic gain control is necessary for better copy under less-than-ideal conditions.

An additional note on the above example is the use of a 2×16 character LCD display. Fig. 3 shows how simple it is to employ such a device for output. If we wanted to incorporate the processor and display in one ready-to-use package, the LCDX pictured in fig. 1 would be an ideal choice. It also contains relay drivers and scalable voltage measuring interfaces.

I hope the above examples demonstrate how easy it is to become familiar with small programmable processors. Their speed and flexibility allow them to replace boards full of digital logic and many analog circuits while allowing the designer/builder freedom to make changes and updates without lifting a soldering iron.

A final note: You may have noticed that fig. 3 shows a thermistor connected to Pin 13 of the processor. Curious? Go to the website <members.cox.net/desertlavender/bxprojects.htm> for more ideas on using these computers on a chip. ■

Notes

1. <www.parallax.com>
2. BASIC—the Beginner's All-purpose Symbolic Instruction Code—was developed at Dartmouth College. For more historical information see: <en.wikipedia.org/wiki/Dartmouth_BASIC>.
3. <www.netmedia.com> and <www.basicx.com>
4. D. Nendza, "Anatomy of a Homebrew Messaging APRS Tracker," *QEX*, January 2005, pp. 16–28.
5. One source of USB to serial converters is *CQ* advertiser West Mountain Radio. See the ad for more information.
6. Peter H. Anderson: <www.phanderson.com/>.
7. Examples mentioned in this article, and more, are available at: <members.cox.net/desertlavender/bxprojects.htm>.
8. Tokenize—see: <en.wikipedia.org/wiki/Tokenize>.
9. ASCII definition: <en.wikipedia.org/wiki/Ascii>.
10. Wavgen is available at <ah0a.org/AH0A.html>.