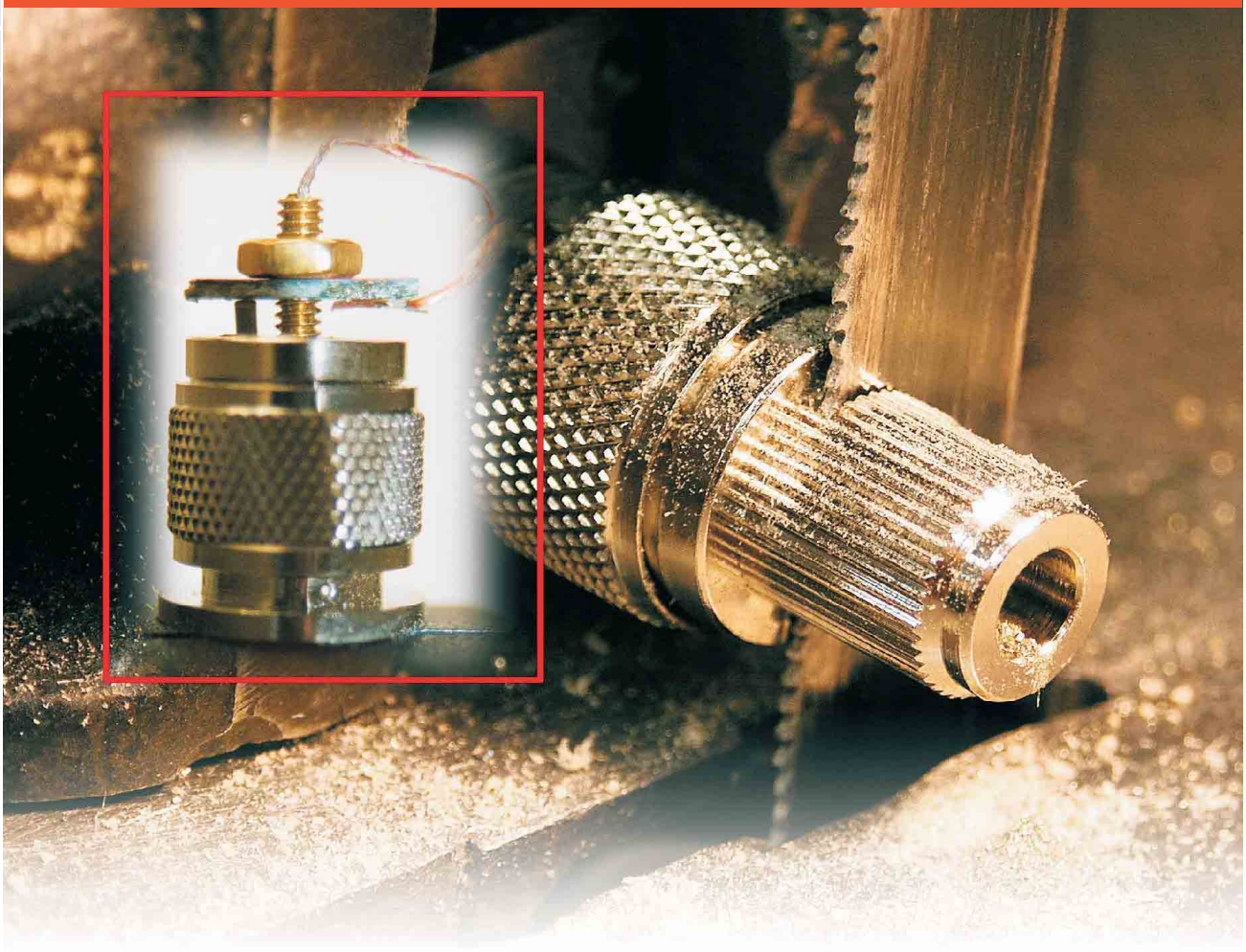
$5

# QEX

INCLUDING:
COMMUNICATIONS QUARTERLY

Forum for Communications Experimenters

September/October 2002

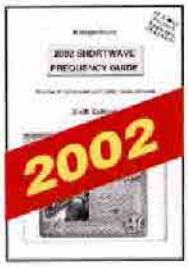


See **W6LSN's** SurCapAdapt in Tech Notes

**ARRL** *The national association for AMATEUR RADIO*
225 Main Street
Newington, CT USA 06111-1494

# ARRL Marketplace!

## These publications have been added to the ARRL Library...
## *so you can add them to yours!*




**2002 Shortwave Frequency Guide**

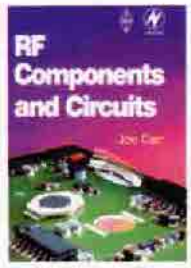
Schedules of clandestine, domestic, and international broadcast stations worldwide! Quickly find frequencies and a superb alphabetical list of stations. Includes another 10,078 entries for utility stations (Red Cross, United Nations and more).

**ARRL Order No. 8663—$34.95**


**2002 Super Frequency List on CD-ROM**

Includes all shortwave broadcast stations worldwide, plus all utility stations from 0 to 30 MHz. Nearly 40,000 entries! Find the latest schedules of clandestine, domestic and international broadcasting services compiled by top experts in this field.

**ARRL Order No. 8671—$24.95**




**RF Components and Circuits**

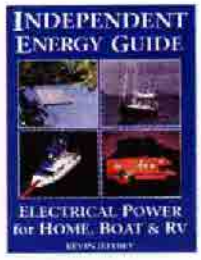
A comprehensive introduction to understanding, designing and building RF circuits, including fault-finding and use of test equipment.

**ARRL Order No. 8759—$37.99**


**Transmission Line Transformers —4th Edition**

Tremendous coverage of the subject of broadband transmission line transformers. Guanella and Ruthroff as well as hundreds of real transformers.

**ARRL Order No. TLT4—$39**




**Independent Energy Guide —Electrical Power for Home, Boat & RV**

Covers fixed, portable, and mobile energy systems; DC charging sources and AC power systems, solar, wind and water power, battery chargers, inverters, and more.

**ARRL Order No. 8601—$19.95**


**Radio-Electronic Transmission Fundamentals**

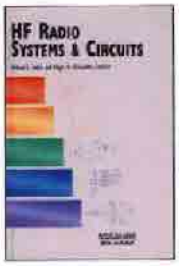
Clear, concise explanations of antennas, transmission lines, and RF networks in the framework of electromagnetic field theory.

**ARRL Order No. RETF—$75**


**The Mobile DXer**

A practical guide to successful mobile DXing. Learn how to select and install mobile gear and pick antennas. Understand propagation, mobile DX operating tips, and making the most of portable operating.

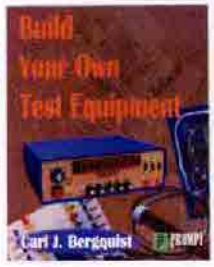
**ARRL Order No. TMDX—$12.95**




**HF Radio Systems & Circuits**

Includes Software! Comprehensive coverage of system definition and performance requirements down to the individual circuit elements that make up radio transmitters and receivers. Thorough attention is given to key circuits like oscillators, synthesizers, filters and amplifiers, speech processing, AGC systems, high linearity amplifiers, and solid state power amplifiers.

**ARRL Order No. 7253—$75**




**Build Your Own Test Equipment**

Build practical devices with commonly-available components. Multi-output test bench power supply, signal generator and tester, IC tester, multimeter, frequency counter, and others.

**ARRL Order No. 8604—$310.95**

**Electronic Applications of the Smith Chart**

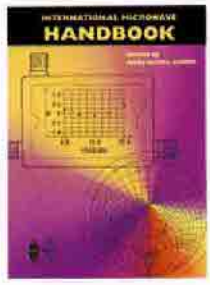
How the chart is used for designing lumped element (inductors and capacitors) and transmission line circuits (coaxial, waveguide, stripline or microstrip lines). Includes tutorial material on transmission line theory and behavior, circuit representation on the chart, matching networks, network transformations and broadband matching.

**ARRL Order No. 7261—$59**


***win*SMITH 2.0**

An easy-to-use, flexible computerized Smith Chart. Accelerate your RF and microwave designs! Unlock a greater understanding of transmission lines and simple matching problems. 3.5-inch installation diskette. Requires Microsoft *Windows*.

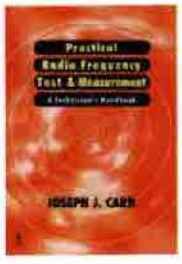
**ARRL Order No. 7946—$80**




**International Microwave Handbook**

Operating techniques, equipment, system anaysis, antennas, propagation, transmission lines, components, semiconductors and valves, construction techniques, and more.

**ARRL Order No. 8739—$39.95**




**Practical Radio Frequency Test & Measurement**

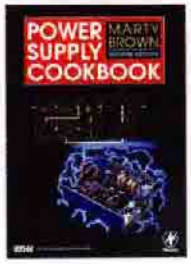
Learn the basics of performing tests and measurements used in radio-frequency systems installation, proof of performance, maintenance, and troubleshooting. Provides immediate applications, test set-ups, procedures, and interpretation of results.

**ARRL Order No. 7954—$39.95**


**33 Simple Weekend Projects**

A wide-ranging collection of do-it-yourself electronics projects. Useful accessories for VHF FMing, projects for satellite communications, CW, simple antennas, and a complete HF station you can build for around $100!

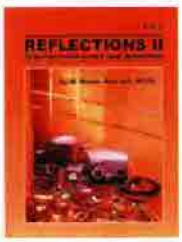
**ARRL Order No. 7628—$15.95**




**Power Supply Cookbook**

A useful resource for amateurs interested in circuit design, and with a basic knowledge of electronics. A step-by-step design framework for power supplies, many of which can be designed in less than a day.

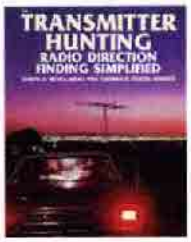
**ARRL Order No. 8599—$39.99**




**Reflections II —Transmission Lines and Antennas**

by M. Walter Maxwell, W2DU. An in-depth treatment of transmission lines, standing waves, antenna matching, reflected power and antenna tuners. Second edition.

**ARRL Order No. REF2—$19.95**




**Transmitter Hunting —Radio Direction Finding Simplified**

Covers equipment and techniques for HF and VHF radio direction finding. Locate jammers and other sources of malicious interference, engage in sport hunting, even help search-and-rescue groups!

**ARRL Order No. 2701—$24.95**

**Order Toll Free**

**1-888-277-5289**

**www.arrl.org/shop**

**Shipping and Handling instructions:** US orders add $5 for one item, plus $1 for each additional item ($10 max.). US orders are shipped via UPS. International orders add $2.00 to the US shipping rate ($12.00 max.). Orders are shipped via surface mail. Other shipping options are available. Please call or write for information.

**Sales Tax** is required for shipments to CT (including S/H), VA 4.5% (excluding S/H), CA (add applicable tax, excluding S/H) and Canada (excluding S/H).


**ARRL** *The national association for* ***AMATEUR RADIO***

225 Main Street, Newington, CT 06111-1494 USA tel: 860-594-0355
fax: 860-594-0303 e-mail: pubsales@arrl.org World Wide Web: http://www.arrl.org/shop

QEX

INCLUDING: COMMUNICATIONS QUARTERLY

*QEX* (ISSN: 0886-8093) is published bimonthly in January, March, May, July, September, and November by the American Radio Relay League, 225 Main Street, Newington CT 06111-1494. Yearly subscription rate to ARRL members is $24; nonmembers $36. Other rates are listed below. Periodicals postage paid at Hartford, CT and at additional mailing offices.

POSTMASTER: Send address changes to: QEX, 225 Main St, Newington, CT 06111-1494
Issue No 214



**Offices**

225 Main St, Newington, CT 06111-1494 USA
Telephone: 860-594-0200
Telex: 650215-5052 MCI
Fax: 860-594-0259 (24 hour direct line)
e-mail: **qex@arrl.org**

Subscription rate for 6 issues:

In the US: ARRL Member $24, nonmember $36;

US by First Class Mail:
ARRL member $37, nonmember $49;

Elsewhere by Surface Mail (4-8 week delivery):
ARRL member $31, nonmember $43;

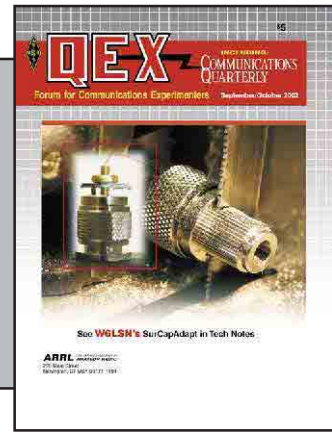
Canada by Airmail: ARRL member $40, nonmember $52;

Elsewhere by Airmail: ARRL member $59, nonmember $71.

Members are asked to include their membership control number or a label from their *QST* wrapper when applying.

In order to ensure prompt delivery, we ask that you periodically check the address information on your mailing label. If you find any inaccuracies, please contact the Circulation Department immediately. Thank you for your assistance.






**About the Cover**

**A SurCapAdapt fixture helps measure chip-component values—the story begins on p 51.**

# Features



# Columns



# Sept/Oct 2002 *QEX* Advertising Index

THE AMERICAN RADIO RELAY LEAGUE

The American Radio Relay League, Inc, is a noncommercial association of radio amateurs, organized for the promotion of interests in Amateur Radio communication and experimentation, for the establishment of networks to provide communications in the event of disasters or other emergencies, for the advancement of radio art and of the public welfare, for the representation of the radio amateur in legislative matters, and for the maintenance of fraternalism and a high standard of conduct.

ARRL is an incorporated association without capital stock chartered under the laws of the state of Connecticut, and is an exempt organization under Section 501(c)(3) of the Internal Revenue Code of 1986. Its affairs are governed by a Board of Directors, whose voting members are elected every two years by the general membership. The officers are elected or appointed by the Directors. The League is noncommercial, and no one who could gain financially from the shaping of its affairs is eligible for membership on its Board.

"Of, by, and for the radio amateur, "ARRL numbers within its ranks the vast majority of active amateurs in the nation and has a proud history of achievement as the standard-bearer in amateur affairs.

A bona fide interest in Amateur Radio is the only essential qualification of membership; an Amateur Radio license is not a prerequisite, although full voting membership is granted only to licensed amateurs in the US.

Membership inquiries and general correspondence should be addressed to the administrative headquarters at 225 Main Street, Newington, CT 06111 USA.

Telephone: 860-594-0200
Telex: 650215-5052 MCI
MCIMAIL (electronic mail system) ID: 215-5052
FAX: 860-594-0259 (24-hour direct line)

**Officers**

***President:*** JIM D. HAYNIE, W5JBP
3226 Newcastle Dr, Dallas, TX 75220-1640
***Executive Vice President:*** DAVID SUMNER, K1ZZ

**The purpose of *QEX*** is to:
1) provide a medium for the exchange of ideas and information among Amateur Radio experimenters,
2) document advanced technical work in the Amateur Radio field, and
3) support efforts to advance the state of the Amateur Radio art.

All correspondence concerning *QEX* should be addressed to the American Radio Relay League, 225 Main Street, Newington, CT 06111 USA. Envelopes containing manuscripts and letters for publication in *QEX* should be marked Editor, *QEX*.

Both theoretical and practical technical articles are welcomed. Manuscripts should be submitted on IBM or Mac format 3.5-inch diskette in word-processor format, if possible. We can redraw any figures as long as their content is clear. Photos should be glossy, color or black-and-white prints of at least the size they are to appear in *QEX*. Further information for authors can be found on the Web at **www.arrl.org/qex/** or by e-mail to **qex@arrl.org**.

Any opinions expressed in *QEX* are those of the authors, not necessarily those of the Editor or the League. While we strive to ensure all material is technically correct, authors are expected to defend their own assertions. Products mentioned are included for your information only; no endorsement is implied. Readers are cautioned to verify the availability of products before sending money to vendors.

# *Empirical Outlook*

## It All Comes Down to Marketing

That old saw "Build a better mousetrap and folks will beat a path to your door" is ridiculous. Mousetraps are still necessary, but it is unlikely anyone will better the combination of three pieces of steel, a spring and some wood. That invention fills a need by being simple, inexpensive and effective. It is reusable, too.

In telecommunications, we seek to fill needs to exchange information efficiently over long distances. Anything that reduces the cost or increases the speed of that is potentially outstanding. More than anything else, digital technology has given us the tools to have a good whack at it. In keeping with Amateur Radio's legacy, we're concentrating on those things that seem most promising and contributing what we know to the general knowledge base. We're not redesigning mousetraps, but does anyone outside Amateur Radio know that?

Like some of our correspondents, we feel the technical facets of Amateur Radio are not getting the press they deserve. Emergency preparedness is apparently perceived as the most valuable purpose of our service. Perhaps that is because the public associates the word "service" with something providing ready and tangible benefits. Most do not think about the other reasons that we are a service; for many of us however, those other reasons are at least as valuable as emergency preparedness.

In fact, emergency services we routinely provide would not be possible had it not been for certain technical innovations, much of the praise for which is due to amateurs. However, the average Joe doesn't know that. All he knows is that ham radio is sort of like CB except you need a license. We have this perceptual problem, but it is no use sitting around and moaning about it—do something!

Most of the writers we know are or were involved professionally in communications. They write about things of interest to them and others like them. We suspect that League technical publications, including *QEX*, could have a broader appeal than they currently enjoy. We are taking steps to see what we can do about that.

We'd like to see *QEX* sustain further growth. Sometimes it's not enough to focus entirely on content; the rest of the job is marketing. One thing's for sure: Potential supporters will not jump on our bandwagon unless they know we exist. It is the same with any product or service, really. We encourage you to stand up for your avocation by helping us get the word out about what we're doing. Entice your colleagues to look at Amateur Radio as a vehicle for future growth. Identify needs and bottlenecks to progress. Attack them with vigor because this is your service, and its fortune depends entirely on what you do with it.

## In This Issue

We received good feedback about the Jul/Aug issue, so the discussion of software controlled and software defined radios (SDR) continues. Mark Erbaugh, N8ME, contributes a piece on software to control the Ten-Tec Pegasus. It is applicable to other transceivers, and it concentrates on software development by example.

Gerald Youngblood, AC5OG, returns with Part 2 of his SDR series. Gerald focuses on how to interface a sound card under *Visual Basic* to acquire data. Jim Scarlett, KD7O's Part 2 has been delayed. Stephen Gradijan, WB5KIA, discusses development environments for high-level languages.

Ray Mack, WD5IFS, gives us an introduction to switching power supplies—something we have been wanting for a while. His subsequent parts will include details of circuit design, magnetics and semiconductors. Tadeusz Raczek, SP7HT, explores the "DX Prowess of Receivers." He includes some specific examples using test data from ARRL and others. John Stephensen, KD6OZH, describes programmable logic devices and their application to SDRs.

In Tech Notes, Dan Hinz, W6LSN, describes a fixture for measuring the value of surface-mount capacitors. In RF, Zack Lau, W1VT, describes a 2-meter Yagi antenna.—*73, Doug Smith, KF6DX,* **kf6dx@arrl.org.** □□

# Customize the Ten-Tec Pegasus —Without Soldering

*Software driven radios? Let's make some software!*

By Mark E. Erbaugh, N8ME

This article is a brief introduction to my favorite *Windows* programming tool, Borland *Delphi*. Instead of a simple "hello, world" introduction, I'll work through developing a minimal control program for the Ten-Tec Pegasus transceiver. The control program will only support receive and won't have all the bells and whistles. Those will be left, as they say, as an exercise for the reader.

However, the tools needed to flesh out this control program will be provided. It is up to you to decide how you want your program to work. You may even come up with an entirely new operating paradigm that advances Amateur Radio, or at least the Pegasus, to the next level.

### *Delphi*

Since its release in 1995, I have been using Borland *Delphi* in my professional work as a *Windows* software developer. I have found that it is every bit as powerful as more sophisticated languages, such as C++ and Java. There is very little that I need to do in my programs that can't be done quickly and easily with *Delphi*. On the other hand, it is easy to learn and you need not swallow the proverbial elephant in one bite.

Programming in *Delphi* is done in the Pascal language. Pascal was developed in the 1970s as a teaching language and was designed to demonstrate sound programming principles. Yes, there is a GOTO statement, but I seldom use it. With just a brief introduction to the language, you should be able to read a piece of code and understand what it is doing.

*Delphi* has been through six major releases. The initial release (*Delphi 1*) was for 16-bit Windows (*Windows* 3.11). The later releases have been for 32-bit Windows. The current version, *Delphi 6*, was released in the spring of 2001. However, I still use the previous version, *Delphi 5*, as it is very solid and I have not found it lacking in the areas where I work. This tutorial will be based on *Delphi 5*, but any version other than 1 or 2 should work as well.

Each release of *Delphi* has come in three editions. For *Delphi 6*, these editions are labeled Personal, Professional and Enterprise in increasing order of sophistication and cost. The Personal edition is suitable for the work described in this article and sells for around $50. The other two editions, with significantly higher prices are geared for professional developers. Essentially, they have more "bells and whistles."

The computer requirements for *Delphi 6* are minimal by today's standards so that shouldn't be an issue. Here they are for the Personal edition:

"Intel Pentium 166 MHz or higher (P2 400 MHz recommended); Microsoft *Windows 2000*, *Windows ME*, *Windows 98* or *Windows NT* 4.0 with Service Pack 5 or later; 32 MB RAM (128 MB recommended); 75 MB hard disk space (compact install), 160 MB hard disk space (full install); CD-ROM drive; VGA or higher-resolution monitor; mouse or other pointing device."

Check out the Borland Web site

3105 Big Plain-Circleville Rd
London, OH 43140
n8me@arrl.net

(**www.borland.com**) for more information.

### Pegasus

The Pegasus has been reviewed in *QST*.[1] It is a very solid ham transceiver worthy of its Ten-Tec lineage. Except for the **ON/OFF** switch, it is a completely computer-controlled radio. All functionality of the radio is provided via a control program running on a computer connected to the radio via a serial cable. The Pegasus is supplied with a control program, and there are other control programs available as shareware. (See the *QST* review of the N4PY software.[2]) The only problem with these control programs is that they do things the way the author thinks best. If you want it to work differently, you can ask the author to make a change, or you can write your own software.

Ten-Tec has published a *Programmers' Reference Guide* that details the protocol of the commands to and responses from the Pegasus. It is available on their Web site (**www.rfsquared.com**). If you look at this guide, you will see that while controlling the Pegasus is not difficult, it is not trivial and certainly is not a candidate for an introductory tutorial.

### *Components*

So how can a general-purpose program-development tool, such as *Delphi*, make developing a Pegasus control program simple enough for this tutorial? Controlling the Pegasus needs to be much more straightforward, such as a simple assignment:

```
RXFREQUENCY := 14230000;
```

The answer is one of the most powerful features of *Delphi*: components. A component is a software "black box" that can encapsulate complexity. *Delphi* comes with dozens of components that are used to simplify *Windows* programming. In addition, you can develop your own components or add additional components developed by others. There are thousands of components available for *Delphi* and many may be downloaded from the Web free of charge.

On the ARRL Web site is the *mePegasus* component.[3] Once this component has been added to your version of *Delphi*, it can be used in programs that you develop just like the components that come with *Delphi*.

### *Getting Started*

Install *Delphi* following the instructions supplied. Install the *mePegasus* component into *Delphi* using the instructions in the help file supplied with it. In this tutorial, we will be working with the four main *Delphi* windows shown in Fig 1.

This screen shot was from my development system and is from *Delphi 5*. The windows may be moved and sized as needed, so your screen may appear slightly different.

[1]Notes appear on page 8.

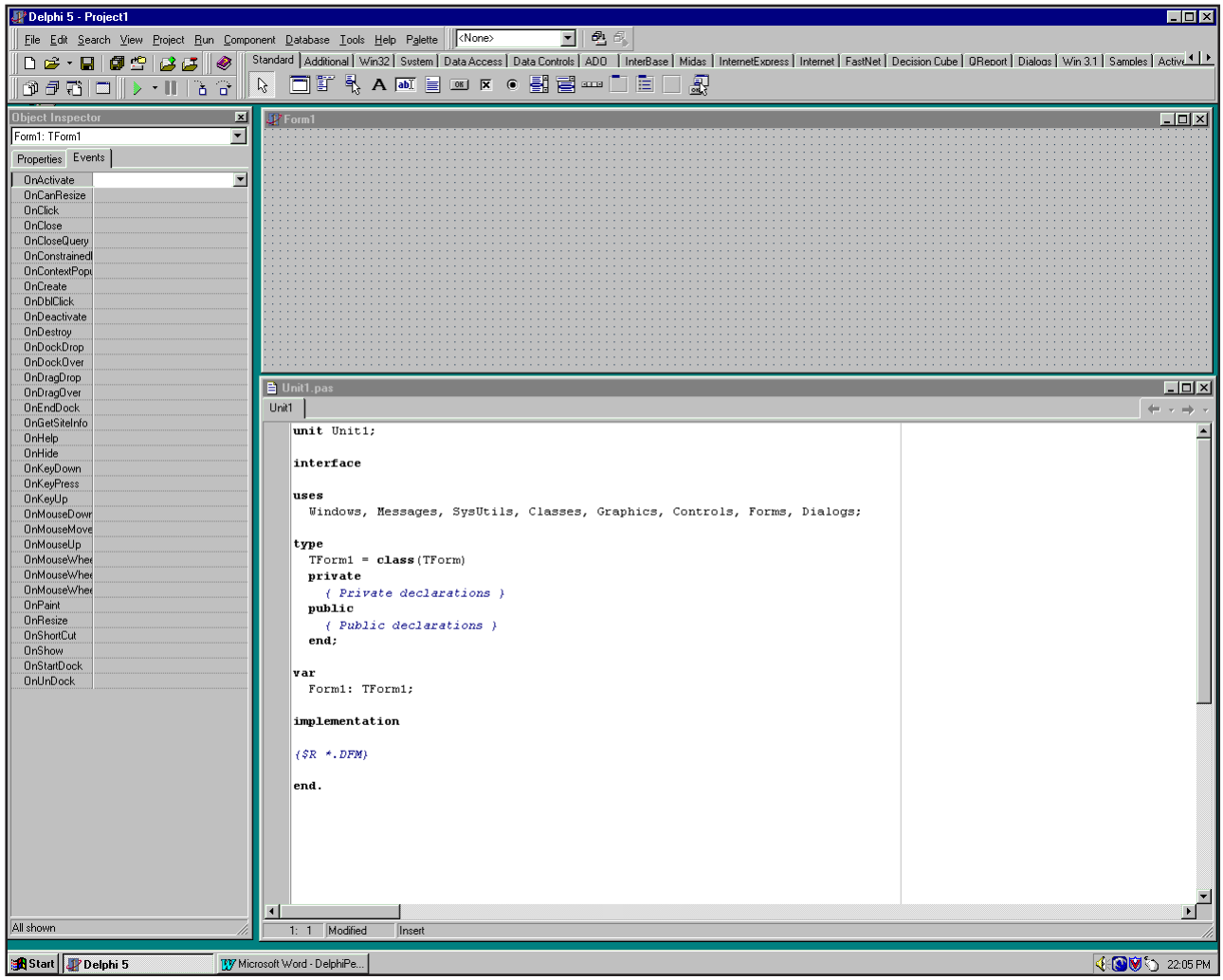



**Fig 1—The *Delphi* programming environment. These are the four main windows.**

The top window is the main *Delphi* window. It has the typical Windows menu bar at the top. On the left are speed buttons for common tasks. More importantly, on the right side is the component palette. This is where you select components to add to your program.

The window on the left side is the Object Inspector. This is where you edit the properties and events of the various components. The window under the main window on the right is where you place the components that form the appearance of your program. The bottom-right window is the code editor, where you enter Pascal program code.

Fig 2 is a screen shot of my version of the finished program. However, the appearance of the program is up to you. You can place the components in any position you like.

From the File menu, select New Application. Then select Save Project As, create a new folder to hold your work and name it *QSTPegasus*. Navigate to this folder and save the unit file as main and the project file as *QSTPegasus*.

As is standard in Windows, you can resize windows by dragging the sides or corners with the mouse, and you can move windows by dragging the title bars with the mouse. Size the main form (Form1) window to the size you want for your control program.

Click in the middle of the form window to display properties of the form in the Object Inspector window. There are two tabs. The left tab lists the properties, the right the events. On each of these tabs, the name of the property or event is listed in the left column and the right column is where you edit the value. Scroll to the name property and change it from "Form1" to "frmQSTPegasus." Change the caption property to "*QST* Pegasus."

On the component palette, select the *QST* tab and click on the mePegasus component (the square with the flying horse). Click anywhere on the *QST* Pegasus form to drop the component. This adds the component to the program. The mePegasus component is a nonvisual component. While it shows up as the flying-horse box in design mode, nothing will display when running the application. In the properties tab for this component, change the name property value to "Pegasus."

Next, add some components for setting a few of the properties of the Pegasus and for indicating signal strength. In this tutorial, we will not be supporting all the functionality of the mePegasus component, but we'll touch on enough to give you an idea of how to add the rest. (The mePegasus help file should be of use here.) The mePegasus component sets up default values for properties that are not used. For example, the tutorial will not have an adjustment for RF gain; the default value is *max* and that will be fine.

From the Win32 palette tab, drop a TrackBar component onto the form. This will be the AF gain control. Position it wherever you like on the form and make it whatever size you like. Note: If you pause the mouse cursor over a component on the palette, Delphi will pop up a hint that gives you the name of the component. Change the following properties from their initial values in Table 1.

Now, select the events tab in the Object Inspector. Locate the OnChange event (it should be at the top of the list) and double click in the right (value) column. Delphi automatically inserts the value "tbAFGainChange" and creates a skeleton method in the code editor window, where you can fill in the code for the event-handler method. For this method, enter the lines under "Code for tbAFGainChange" from the Code sidebar.

This event handler will run every time the user changes the position of the slider. We have already set the properties to allow the position of the track bar to range from 0 to 255, which are the values of the AF gain setting on the Pegasus. This line of code instructs the mePegasusControl component that we named Pegasus to set the AF gain of the Pegasus to the position of the track bar.

Now, from the Standard palette tab, drop a label component (with the letter A) onto the form. This will be the caption for the AF gain control, and it will display the numeric value of the AF-gain setting. Move the label wherever you like, but it should probably be near the AF-gain track bar. Change the properties to those shown in Table 2.

The ampersand (&) in the caption property causes Alt-A to be a shortcut for this label component. Since we set the FocusControl property to tbAFGain (our AF gain track bar), this component

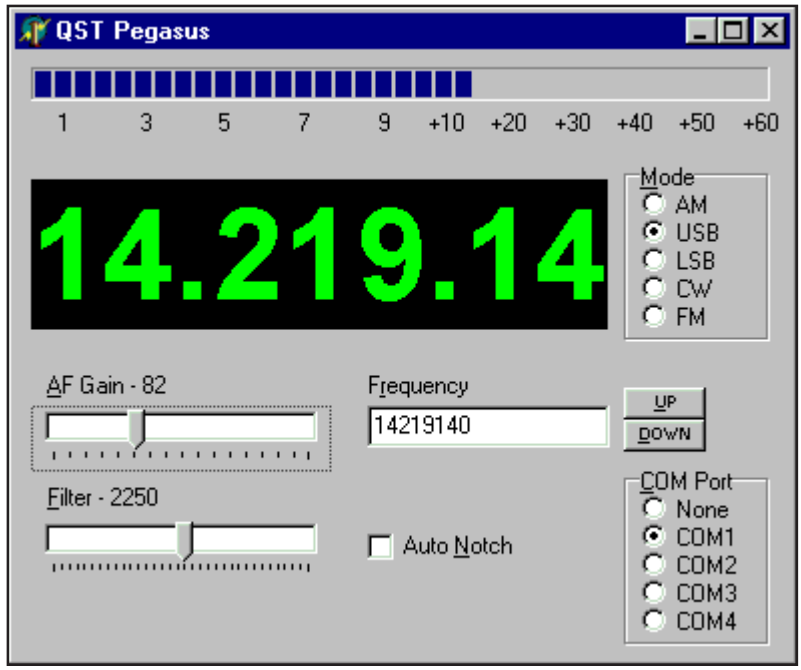



**Fig 2—A screen capture of the finished program.**

**Table 1—AFGain TrackBar Properties**

| *Name* | *tbAFGain* |
|---|---|
| Caption | AF Gain |
| Max | 255 |
| Frequency | 16 |
| PageSize | 16 |

**Table 2—AFGain Label Properties**

| *Property* | *Value* |
|---|---|
| Name | lblAFGain |
| Caption | &AF Gain |
| FocusControl | tbAFGain |

will get the windows focus when alt-A is pressed. Notice that when you click in the value column for the FocusControl property, a drop-down menu button appears on the right side of the column. You can use the drop-down list and select tbAFGain from it instead of typing if you choose.

Next, click on the mePegasus component and select the event tab in the Object Inspector. You will notice that this component has many events. That is because events are the primary way the component communicates with the program. Double-click in the value column for the OnAFGain event to create an event handler skeleton. For this event handler type the lines in the Code sidebar for tbAFGain EventHandler Code.

This event handler is called whenever the mePegasus component's AFGain value is changed. In this tutorial, we don't need to set the track-bar position as it has just been set, and we could set the lblAFGain caption in the tbAFGain.OnChange event handler. This illustrates an important feature of the mePegasus component. It is possible that the AF gain could be set by other actions. If you didn't use the OnAFGain event, you would have to remember to update the track bar position and label caption. The OnAFGain event handler can centralize this code. We'll see the true value of this when we get around to setting the receiver frequency.

You are now ready to see the first fruits of your labor. We are going to compile and run this program. This demonstrates the incremental development that is possible with *Delphi*. From the File menu, select Save All. We don't want to lose all the work! Then from the Run menu, select Run. This will compile and, if there are no errors, run your program. If the compiler reports errors, recheck your work and resolve the errors. In most cases, the compiler error message will be helpful in locating and fixing the error. If you double-click on the error message, the error location will be displayed in the code window. If you click on the error message and press F1, the help system will display details about the type of error.

Once the program compiles successfully, the program will run. You will see the track bar. As you move it with the mouse, the caption label will display the current value. To quit the program and return to designing, click on the close box (X) at the top right of the window. At this point, we are not actually communicating with the Pegasus so it doesn't matter if the Pegasus is connected or powered up.

### Adding More Controls

Next, we will add a *radio group* component for mode selection. The component is called a radio group because it behaves like the old push button radios once common in cars. When you push in one button, all the other buttons pop out. Similarly, when you click on one item, the dot moves to that item; only one item may have the dot at a time.

**Table 3—Radio Group Properties**

| *Property* | *Value* |
|---|---|
| Name | rgMode |
| Caption | &Mode |
| Items | AM USB LSB CW FM |
| ItemIndex | 0 |
| Columns | 1 or 5 |

**Table 4—Filter-Selection TrackBar Properties**

| *Property* | *Value* |
|---|---|
| Name | tbFilter |
| Max | 33 |
| PageSize | 1 |

**Table 5—Filter TrackBar Label Properties**

| | |
|---|---|
| Name | lblFilter |
| Caption | &Filter |
| FocusControl | tbFilter |

**Table 6—Autonotch Checkbox Properties**

| | |
|---|---|
| Name | ckbAutoNotch |
| Caption | Auto &Notch |

**Table 7—Frequency Caption Label Properties**

| *Property* | *Value* |
|---|---|
| Name | lblFrequency |
| Caption | F&requency |
| FocusControl | edFrequency |

**Table 8—Frequency-Control Button Properties**

| | |
|---|---|
| Name | btnUp |
| Caption | &Up |
| | |
| Name | btnDown |
| Caption | &Down |

From the Standard palette page, drop a RadioGroup component on the form. Set its property values as shown in Table 3.

If you want the choices to be in a horizontal row, set the Columns property to "5." If you want them to be in a vertical row, set Columns to "1." Then size the component as appropriate. To enter the Items value, click in the values column: an ellipsis button is displayed. Click on it and a window pops up. Enter the values in the order given, one to a line. Add the code lines from the Code sidebar to the rgMode.OnClick event handler. Then add the code for the PegasusControl.OnRxMode event handler.

Next, add another track bar for filter selection. There are 34 possible filters on the Pegasus ranging from 300 Hz to 8000 Hz. Set the property values as shown in Table 4. Add the tbFilter.OnChange event-handler code. Add a caption label as shown in Table 5, and add the Pegasus.OnRxFilter event-handler code.

From the Standard palette page, add a CheckBox component with the properties shown in Table 6. Then add the ckbAutoNotch.OnClick event-handler code and the Pegasus.OnAutoNotch event-handler code. Save your work and run the program to see how it behaves.

Now we need a way to set the frequency. The tutorial will have two or three ways of doing it, which are interchangeable. You can type the frequency into an edit box, click up or down buttons or adjust the frequency using the remote tuning knob on the Pegasus if that is available.

From the Standard palette page, add an Edit component. Size this to handle eight characters as we will enter and display the frequency in hertz. Set the Name property to "edFrequency" and add the edFrequency.OnKeyPress event-handler code.

Note that "end" at the end is in addition to the end provided automatically in the event-handler skeleton. This event handler is a little more complex than previous event handlers. Its purpose is to send the new frequency to the radio when the user pressed the **ENTER** key. The "try ... except" handling is to gracefully handle the case when the user types a non-numeric value. Add a caption label and set its properties as shown in Table 7.

Add the Pegasus.OnRxFrequency event-handler code. Then add two Button components and set their properties as shown in Table 8.

Enter the code for the btnUp.OnClick and btnDown.OnClick event handlers. Now add the Pegasus.OnEncoder event-handler code. Save your work and run the program. Since we have not established communication with the Pegasus, the remote-tuning pod will not be working. The next step is to actually establish communication with the Pegasus.

**Communicating with Pegasus**

From the Standard palette page, drop a RadioGroup component on the form and set its property values as shown in Table 9. Add the and the Pegasus.OnComPort event-handler code.

Run the program. Select the COM port connected to the transceiver in the software and you should have your very own custom receiver. Exit the program (and *Delphi*) and look at the program icon that *Delphi* generated for you in the QSTPegasus folder. You have a functioning Windows program that is less than 500 kb in size, and this is the only file needed to run your program. You could transfer just this file to another machine with a Pegasus connected to the same serial port and it would work.

Let's add some more controls. Reopen *Delphi* and the source file. From the Win32 palette, add a ProgressBar component that will become the S-meter. Its properties are in Table 10. Add the Pegasus.OnRxMeter event-handler code.

The maximum value of 19 corresponds to a maximum S-meter reading of about 60 over S9. The *shr 8* (shift right by 8) divides the value by 256. We do this because the Pegasus reports signal strength in S units and fractional S units; we want only the integer value.

To "calibrate" the S-meter display, set the Position property to "1" and notice how much is displayed. Place a label with the caption of "1" at this point. Repeat with positions and labels for "3," "5," "7" and "9." To get the calibration points over S9, first set the position to "14." This corresponds to five S units over S9, or +30 dB. Place a label with a caption of "+30" at this point. Then place labels of "+10" and "+20" in between the "9" and "+30." Set the position to "19." This is the +60-dB point. Add the "+40" and "+50" labels in a similar fashion.

You may notice that every time you start the program, the radio parameters return to their default values. The Pegasus has no memory for such storage. All parameters must be restored each time the Pegasus is started. A control program needs to save the last used settings and restore them the next time it is run. Three properties of the mePegasus component can be used to save and restore the settings: ComPort, CurrentVFO and CurrentParams. The ComPort property is an integer value. The other two are record structures that contain multiple fields.

These data could be saved to a file, such as an INI file, but for 32-bit Windows programs, the preferred storage location is the Windows Registry. Create an event handler for the frmQSTPegasus.OnCreateEvent, and for the frmQSTPegasus.OnDestroy event.

Note that the full event handlers are shown. The *var* and lines below it need to be added above the begin line. These lines declare local variables for the event handlers.

You also need to manually add a unit name to the "uses" clause at the top of the file. Up until now, *Delphi* has added files to the "uses" clause as you add components, but since Tregistry is not a component, you have to add its unit, which is "Registry." The completed "uses" clause should look as shown in the Code sidebar. The explanation of the code in these event handlers is beyond the scope of this tutorial. Look in the *Delphi* help file for TRegistry to see how to use the Windows registry.

As a final touch, we will add a large frequency display. Add another label component to the screen. Change the name property to lblFrequencyDisplay. Click on the Font property value and then on the ellipsis button to bring up the font dialog. Select a large font and a bright color, such as green. Set the label color to black. Change the caption to "00.000.00" and make the label big enough to display all of the text. Add a new line to the Pegasus.OnRxFrequency event handler after the line that is already there.

**Conclusion**

This completes the tutorial. While we have only examined a handful of the properties available in the mePegasus component, we have examined a representative group. All the remaining properties behave similarly to the ones that were used in the tutorial. Consult the mePegasus help file for details on all the features.

The tutorial presents a very basic control program. It is not intended to be complete and does not exploit even all the receive features of the Pegasus. The processes of adding support for the remaining features are simple and similar to the ones already shown. The difficult task is designing the way those features should look and work when added.

All of the control programs that I have seen tend to create a virtual radio front panel on the computer screen. They show a picture of the radio, complete with images of knobs and buttons. The user manipulates these virtual knobs and buttons with the computer mouse, just as a user would manipulate the knobs and buttons of a real radio with their hands.

For those of us who have been around radios for a while, this virtual interface is familiar. This interface represents a model or paradigm that has been around for a long time. In fact, when you look at a modern radio you can see how persistent is the current paradigm. Modern radios have a knob or knobs that manipulate a VFO. In old radios, VFO stood for variable frequency oscillator and when you turned the VFO knob, you were changing the value of a variable capacitor that changed the frequency of this oscillator. In today's radios, you

**Table 9—Communication Radio Group Properties**

| *Property* | *Value* |
|---|---|
| Name | rgComPort |
| Caption | &COM Port |
| Items | NONE COM1 COM2 COM3 COM4 |
| ItemIndex | 0 |
| Columns | 1 or 5 |

**Table 10—S-Meter Progress Bar Properties**

| *Property* | *Value* |
|---|---|
| Name | pbSMeter |
| Max | 19 |

are merely turning the knob of an optical encoder that sends pulses to a microprocessor. The microprocessor interprets these pulses as a request to change the parameters to a digital frequency synthesizer.

Perhaps the persistence of this paradigm indicates that it is still the best available for the task. Yet now that the interface is a computer program, there may be a previously undiscovered paradigm that revolutionizes the way we use our radios.

On the other extreme, with a custom control program, you could emulate the interface of your favorite radio, contemporary or from the past. A *QST* article showed a slide-rule tuning dial for the Yaesu FT-1000 radios.[4] Something like this is certainly doable.

If your computer has a sound card, the control program can speak to the operator by playing prerecorded wave files. This could be a benefit to vision-impaired operators.

The number of modifications that can be made is virtually unlimited. The nice thing about these modifications is that they are entirely reversible. Assume that after you have worked on a modification, you decide that it really isn't going to work out the way you planned and you decide to scrap it. All you have to do is delete some source code files and revert to the previous version. All your unwanted changes are automatically undone. Try that with a chassis full of new holes and additional circuitry!

*Mark earned a two-year nonrenewable novice ticket in high school, but let that license expire without ever getting on the air. He got back into Amateur Radio in 1982. He has a BS in Chemical Engineering from Rose-Hulman Institute of Technology and works as a self-employed programmer developing business support software for* Windows *(using* Delphi*).*

**Notes**

[1]P. Danzer, N1II, "Ten-Tec Pegasus HF Transceiver" (Product Review), *QST*, Feb 2000, pp 63-67.

[2]A. Gavenas, WA6IQD, "N4PY Pegasus Control Program, version 1.45" (Short Takes), *QST,* May 2001, p 65.

[3]You can download this package from the ARRL Web **http://www.arrl.org/qexfiles**. Look for 0209ERBAUGH.ZIP.

[4]B. Wood, W0DZ, "The Return of the Slide Rule Dial," *QST*, Feb 2002, pp 33-35.

**Code**

**tbAFGainChange Event-Handler Code**
```
if not Pegasus.InUpdateNotify then
Pegasus.AFGain := tbAFGain.Position;
```

**tbAFGain Event-Handler Code**
```
tbAFGain.Position := Sender.AFGain;
lblAFGain.Caption := '&AF Gain -'
+ IntToStr( Sender.AFGain );
```

**rgMode.OnClick Event-Handler Code**
```
if not Pegasus.InUpdateNotify then
Pegasus.RxMode := rgMode.ItemIndex;
```

**PegasusControl.OnRxMode Event-Handler Code**
```
rgMode.ItemIndex := Sender.RxMode;
```

**tbFilter.OnChange Event-Handler Code**
```
if not Pegasus.InUpdateNotify then
Pegasus.RxFilter := 33 - tbFilter.Position;
```

**Pegasus.OnRxFilter Event-Handler Code**
```
tbFilter.Position := 33 - Sender.RxFilter;
lblFilter.Caption := '&Filter - '
+ IntToStr( FILTER_WIDTH[Sender.RxFilter] );
```

**ckbAutoNotch.OnClick Event-Handler Code**
```
if not Pegasus.InUpdateNotify then
Pegasus.AutoNotch := ckbAutoNotch.checked;
```

**Pegasus.OnAutoNotch Event-Handler Code**
```
ckbAutoNotch.Checked := Sender.AutoNotch;
```

**edFrequency.OnKeyPress Event-Handler Code**
```
if Key = #13 then
   try
    Pegasus.RxFrequency := IntToStr( edFrequency.Text );
    Key := #0;
   except
   end;
```

**EdFrequency.Text Event-Handler Code**
```
edFrequency.Text := IntToStr( sender.RxFrequency );
lblFrequencyDisplay.Caption := format( '%.2d.%.3d.%.2d',
  [sender.RxFrequency div 1000000,
  ( sender.RxFrequency mod 1000000 ) div 1000,
  (sender.RxFrequency mod 1000) div 10 ] );
```

**btnUp.OnClick Event-Handler Code**
```
Pegasus.RxFrequency := Pegasus.RxFrequency + 10;
```

**btnDown.OnClick Event-Handler Code**
```
Pegasus.RxFrequency := Pegasus.RxFrequency - 10;
```

**Pegasus.OnEncoder Event-Handler Code**
```
Pegasus.RxFrequency := Pegasus.RxFrequency
+ 10 * Pegasus.LastEncoder;
```

**rgMode.OnClick Event-Handler Code**
```
if not Pegasus.InUpdateNotify then
Pegasus.ComPort := rgComPort.ItemIndex;
```

**Pegasus.OnComPort Event-Handler Code**
```
rgMode.ItemIndex := Sender.ComPort;
```

**Pegasus.OnRxMeter Event-Handler Code**
```
pbSMeter.Position := Sender.LastSMeter shr 8;
```

**frmQSTPegasus.OnCreateEvent Event-Handler Code**

```
   procedure TfrmQSTPegasus.FormCreate(Sender: TObject);
   var
   Reg: TRegistry;
   ComPort : byte;
   VFORec : TVFORec;
   Params : TPegasusControlRec;
begin
   Reg := TRegistry.Create;
   try
    Reg.RootKey := HKEY_CURRENT_USER;
    if Reg.OpenKey( '\Software\Pegasus', false ) then
    begin
     if Reg.ReadBinaryData( 'VFO', VFORec, sizeof( TVFORec )) > 0
      then Pegasus.CurrentVFO := VFORec;
     if Reg.ReadBinaryData( 'PARAMS', Params, sizeof( TPegasusControlRec )) > 0
      then Pegasus.CurrentParams := Params;
     try
      ComPort := Reg.ReadInteger( 'COM_PORT' );
      Pegasus.ComPort := ComPort
     except
     end;
    end;
   finally
    Reg.CloseKey;
    Reg.Free;
   end;
end;
```

**frmQSTPegasus.OnDestroy Event-Handler Code**

```
procedure TfrmQSTPegasus.FormDestroy(Sender: TObject);
var
   Reg: TRegistry;
   ComPort : byte;
   VFORec : TVFORec;
   Params : TPegasusControlRec;
begin
   Reg := TRegistry.Create;
   try
    Reg.RootKey := HKEY_CURRENT_USER;
    if Reg.OpenKey( '\Software\Pegasus', true ) then
    begin
     VFORec := Pegasus.CurrentVFO;
     Params := Pegasus.CurrentParams;
     ComPort := Pegasus.ComPort;
     Reg.WriteBinaryData( 'VFO', VFORec, sizeof( TVFORec ));
     Reg.WriteBinaryData( 'PARAMS', Params, sizeof( TPegasusControlRec ));
     Reg.WriteInteger( 'COM_PORT', ComPort );
    end;
   finally
    Reg.CloseKey;
    Reg.Free;
   end;
end;
```

**A Proper "Uses" Clause**

```
uses
   Registry,
   Windows, Messages, SysUtils, Classes, Graphics, Controls,
   Forms, Dialogs, StdCtrls, ComCtrls, PegasusControl,
   ExtCtrls;
```

**New Line for the Pegasus.OnRxFrequency Event-Handler Code**

```
lblFrequencyDisplay.Caption := format( '%.2d.%.3d.%.2d',
   [sender.RxFrequency div 1000000,
   ( sender.RxFrequency mod 1000000 ) div 1000,
   (sender.RxFrequency mod 1000) div 10 ] );
```

# A Software-Defined Radio for the Masses, Part 2

*Come learn how to use a PC sound card to enter the wonderful world of digital signal processing.*

By Gerald Youngblood, AC5OG

Part 1 gave a general description of digital signal processing (DSP) in software-defined radios (SDRs).[1] It also provided an overview of a full-featured radio that uses a personal computer to perform all DSP functions. This article begins design implementation with a complete description of software that provides a full-duplex interface to a standard PC sound card.

To perform the magic of digital signal processing, we must be able to convert a signal from analog to digital and back to analog again. Most amateur experimenters already have this capability in their shacks and many have used it for slow-scan television or the new digital modes like PSK31.

Part 1 discussed the power of quadrature signal processing using *in-phase* (*I*) and *quadrature* (*Q*) signals to receive or transmit using virtually any modulation method. Fortunately, all modern PC sound cards offer the perfect method for digitizing the *I* and *Q* signals. Since virtually all cards today provide 16-bit stereo at 44-kHz sampling rates, we have exactly what we need capture and process the signals in software. Fig 1 illustrates a direct quadrature-conversion mixer connection to a PC sound card.

This article discusses complete source code for a DirectX sound-card interface in Microsoft *Visual Basic*. Consequently, the discussion assumes that the reader has some fundamental knowledge of high-level language programming.

### Sound Card and PC Capabilities

Very early PC sound cards were low-performance, 8-bit mono versions. Today, virtually all PCs come with 16-bit stereo cards of sufficient quality to be used in a software-defined radio. Such a card will allow us to demodulate, filter and display up to approximately a 44-kHz bandwidth, assuming a 44-kHz sampling rate. (The bandwidth is 44 kHz, rather than 22 kHz, because the use of two channels effectively doubles the sampling rate—*Ed.*) For high-performance applications, it is important to select a card that offers a high dynamic range—on the order of 90 dB. If you are just getting started, most PC sound cards will allow you to begin experimentation, although they

[1]Notes appear on page 18.

8900 Marybank Dr
Austin, TX 78750
gerald@sixthmarket.com

may offer lower performance.

The best 16-bit price-to-performance ratio I have found at the time of this article is the Santa Cruz 6-channel DSP Audio Accelerator from Turtle Beach Inc (**www.tbeach.com**). It offers four 18-bit internal analog-to-digital (A/D) input channels and six 20-bit digital-to-analog (D/A) output channels with sampling rates up to 48 kHz. The manufacturer specifies a 96-dB signal-to-noise ratio (SNR) and better than –91 dB total harmonic distortion plus noise (THD+N). Crosstalk is stated to be –105 dB at 100 Hz. The Santa Cruz card can be purchased from online retailers for under $70.

Each bit on an A/D or D/A converter represents 6 dB of dynamic range, so a 16-bit converter has a theoretical limit of 96 dB. A very good converter with low-noise design is required to achieve this level of performance. Many 16-bit sound cards provide no more than 12-14 effective bits of dynamic range. To help achieve higher performance, the Santa Cruz card uses an 18-bit A/D converter to deliver the 96 dB dynamic range (16-bit) specification.

A SoundBlaster 64 also provides reasonable performance on the order of 76 dB SNR according to PC AV Tech at **www.pcavtech.com**. I have used this card with good results, but I much prefer the Santa Cruz card.

The processing power needed from the PC depends greatly on the signal processing required by the application. Since I am using very-high-performance filters and large fast-Fourier transforms (FFTs), my applications require at least a 400-MHz Pentium II processor with a minimum of 128 MB of RAM. If you require less performance from the software, you can get by with a much slower machine. Since the entry level for new PCs is now 1 GHz, many amateurs have ample processing power available.

### Microsoft DirectX versus Windows Multimedia

Digital signal processing using a PC sound card requires that we be able to capture blocks of digitized *I* and *Q* data through the stereo inputs, process those signals and return them to the sound-card outputs in pseudo real time. This is called *full duplex*. Unfortunately, there is no high-level software interface that offers the capabilities we need for the SDR application.

Microsoft now provides two application programming interfaces[2] (APIs) that allow direct access to the sound card under *C++* and *Visual Basic*. The original interface is the Windows Multimedia system using the Waveform Audio API. While my early work was done with the Waveform Audio API, I later abandoned it for the higher performance and simpler interface DirectX offers. The only limitation I have found with DirectX is that it does not currently support sound cards with more than 16-bits of resolution. For 24-bit cards, Windows Multimedia is required. While the Santa Cruz card supports 18-bits internally, it presents only 16-bits to the interface. For information on where to download the DirectX software development kit (SDK) see Note 2.

### Circular Buffer Concepts

A typical full-duplex PC sound card allows the simultaneous capture and playback of two or more audio channels (stereo). Unfortunately, there is no high-level code in *Visual Basic* or C++ to directly support full duplex as required in an SDR. We will therefore have to write code to directly control the card through the DirectX API.

DirectX internally manages all low-level buffers and their respective interfaces to the sound-card hardware. Our code will have to manage the high-level DirectX buffers (called DirectSoundBuffer and DirectSoundCaptureBuffer) to provide uninterrupted operation in a multitasking system. The DirectSoundCaptureBuffer stores the digitized signals from the stereo

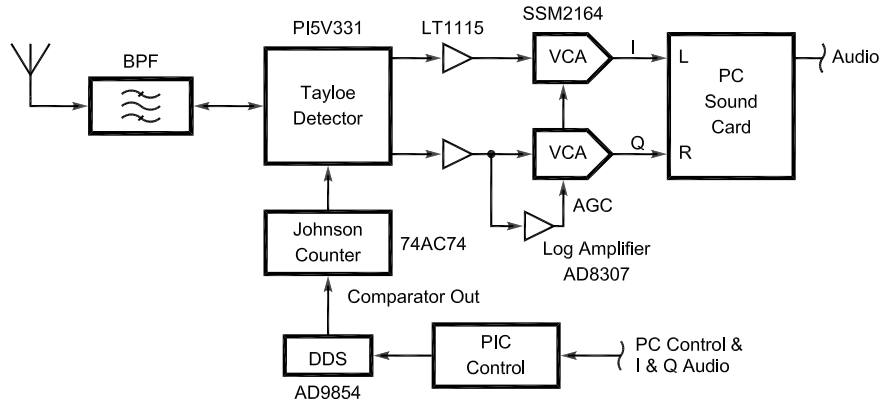



**Fig 1—Direct quadrature conversion mixer to sound-card interface used in the author's prototype.**

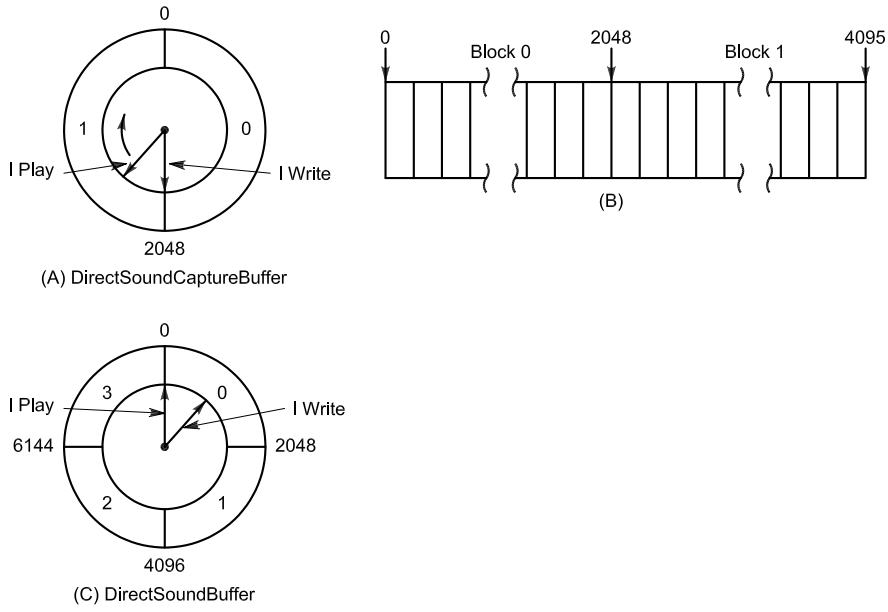



**Fig 2—DirectSoundCaptureBuffer and DirectSoundBuffer circular buffer layout.**

A/D converter in a circular buffer and notifies the application upon the occurrence of predefined events. Once captured in the buffer, we can read the data, perform the necessary modulation or demodulation functions using DSP and send the data to the DirectSoundBuffer for D/A conversion and output to the speakers or transmitter.

To provide smooth operation in a multitracking system without audio popping or interruption, it will be necessary to provide a multilevel buffer for both capture and playback. You may have heard the term double buffering. We will use double buffering in the DirectSoundCaptureBuffer and quadruple buffering in the DirectSoundBuffer. I found that the quad buffer with overwrite detection was required on the output to prevent overwriting problems when the system is heavily loaded with other applications. Figs 2A and 2B illustrate the concept of a circular double buffer, which is used for the DirectSoundCaptureBuffer. Although the buffer is really a linear array in memory, as shown in Fig 2B, we can visualize it as circular, as illustrated in Fig 2A. This is so because DirectX manages the buffer so that as soon as each cursor reaches the end of the array, the driver resets the cursor to the beginning of the buffer.

The DirectSoundCaptureBuffer is broken into two blocks, each equal in size to the amount of data to be captured and processed between each event. Note that an event is much like an interrupt. In our case, we will use a block size of 2048 samples. Since we are using a stereo (two-channel) board with 16 bits per channel, we will be capturing 8192 bytes per block (2048 samples × 2 channels × 2 bytes). Therefore, the DirectSoundCaptureBuffer will be twice as large (16,384 bytes).

Since the DirectSoundCapture Buffer is divided into two data blocks, we will need to send an event notification to the application after each block has been captured. The DirectX driver maintains cursors that track the position of the capture operation at all times. The driver provides the means of setting specific locations within the buffer that cause an event to trigger, thereby telling the application to retrieve the data. We may then read the correct block directly from the DirectSoundCaptureBuffer segment that has been completed.

Referring again to Fig 2A, the two cursors resemble the hands on a clock face rotating in a clockwise direction. The capture cursor, lPlay, represents the point at which data are currently being captured. (I know that sounds backward, but that is how Microsoft defined it.) The read cursor, lWrite, trails the capture cursor and indicates the point up to which data can safely be read. The data after lWrite and up to and including lPlay are not necessarily good data because of hardware buffering. We can use the lWrite cursor to trigger an event that tells the software to read each respective block of data, as will be discussed later in the article. We will therefore receive two events per revolution of the circular buffer. Data can be captured into one half of the buffer while data are being read from the other half.

Fig 2C illustrates the DirectSoundBuffer, which is used to output data to the D/A converters. In this case, we will use a quadruple buffer to allow plenty of room between the currently playing segment and the segment being written. The play cursor, lPlay, always points to the next byte of data to be played. The write cursor, lWrite, is the point after which it is safe to write data into the buffer. The cursors may be thought of as rotating in a clockwise motion just as the capture cursors do. We must monitor the location of the cursors before writing to buffer locations between the cursors to prevent overwriting data that have already been committed to the hardware for playback.

Now let's consider how the data maps from the DirectSoundCaptureBuffer to the DirectSoundBuffer. To prevent gaps or pops in the sound due to processor loading, we will want to fill the entire quadruple buffer before starting the playback looping. DirectX allows the application to set the starting point for the lPlay cursor and to start the playback at any time. Fig 3 shows how the data blocks map sequentially from the DirectSoundCaptureBuffer to the DirectSoundBuffer. Block 0 from the DirectSoundCaptureBuffer is transferred to Block 0 of the DirectSoundBuffer. Block 1 of the DirectSoundCaptureBuffer is next transferred to Block 1 of the DirectSoundBuffer and so forth. The subsequent source-code examples show how control of the buffers is accomplished.

### Full Duplex, Step-by-Step

The following sections provide a detailed discussion of full-duplex DirectX implementation. The example code captures and plays back a stereo audio signal that is delayed by four

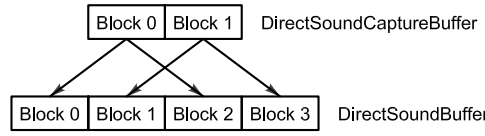



**Fig 3—Method for mapping the DirectSoundCaptureBuffer to the DirectSoundBuffer.**

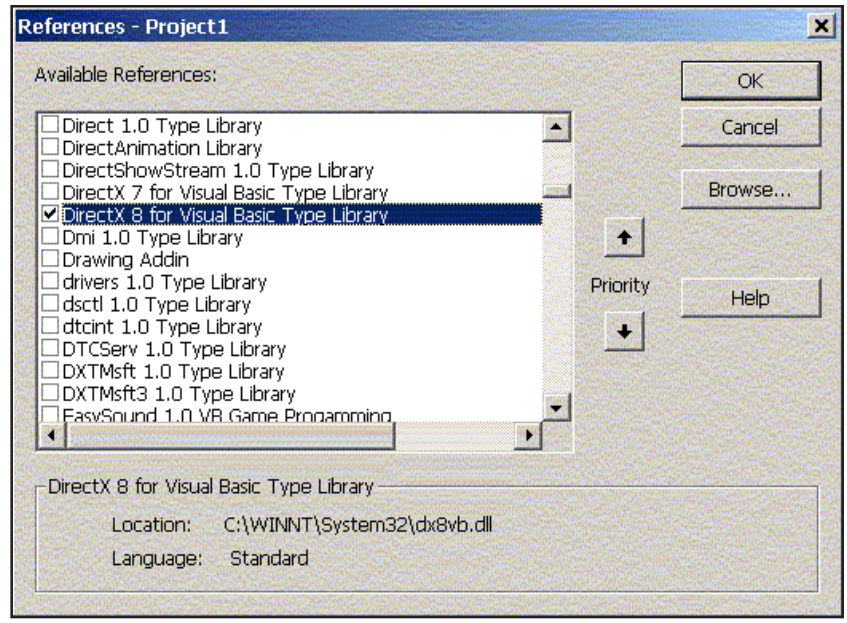



**Fig 4—Registration of the DirectX8 for *Visual Basic* Type Library in the *Visual Basic* IDE.**

capture periods through buffering. You should refer to the "DirectX Audio" section of the *DirectX 8.0 Programmers Reference* that is installed with the DirectX software developer's kit (SDK) throughout this discussion. The DSP code will be discussed in the next article of this series, which will discuss the modulation and demodulation of quadrature signals in the SDR. Here are the steps involved in creating the DirectX interface:

- Install DirectX runtime and SDK.
- Add a reference to DirectX8 for *Visual Basic* Type Library.
- Define Variables, I/O buffers and DirectX objects.
- Implement DirectX8 events and event handles.
- Create the audio devices.
- Create the DirectX events.
- Start and stop capture and play buffers.
- Process the DirectXEvent8.
- Fill the play buffer before starting playback.
- Detect and correct overwrite errors.
- Parse the stereo buffer into *I* and *Q* signals.
- Destroy objects and events on exit.

Complete functional source code for the DirectX driver written in Microsoft *Visual Basic* is provided for download from the *QEX* Web site.[3]

### *Install DirectX and Register it within* Visual Basic

The first step is to download the DirectX driver and the DirectX SDK

```
Option Explicit

'Define Constants
Const Fs As Long = 44100          'Sampling frequency Hz
Const NFFT As Long = 4096         'Number of FFT bins
Const BLKSIZE As Long = 2048      'Capture/play block size
Const CAPTURESIZE As Long = 4096        'Capture Buffer size

'Define DirectX Objects
Dim dx As New DirectX8            'DirectX object
Dim ds As DirectSound8            'DirectSound object
Dim dspb As DirectSoundPrimaryBuffer8   'Primary buffer object
Dim dsc As DirectSoundCapture8    'Capture object
Dim dsb As DirectSoundSecondaryBuffer8 'Output Buffer object
Dim dscb As DirectSoundCaptureBuffer8   'Capture Buffer object

'Define Type Definitions
Dim dscbd As DSCBUFFERDESC        'Capture buffer description
Dim dsbd As DSBUFFERDESC          'DirectSound buffer description
Dim dspbd As WAVEFORMATEX         'Primary buffer description
Dim CapCurs As DSCURSORS          'DirectSound Capture Cursor
Dim PlyCurs As DSCURSORS          'DirectSound Play Cursor

'Create I/O Sound Buffers
Dim inBuffer(CAPTURESIZE) As Integer    'Demodulator Input Buffer
Dim outBuffer(CAPTURESIZE) As Integer   'Demodulator Output Buffer

'Define pointers and counters
Dim Pass As Long                  'Number of capture passes
Dim InPtr As Long                 'Capture Buffer block pointer
Dim OutPtr As Long                'Output Buffer block pointer
Dim StartAddr As Long             'Buffer block starting address
Dim EndAddr As Long               'Ending buffer block address
Dim CaptureBytes As Long          'Capture bytes to read

'Define loop counter variables for timing the capture event cycle
Dim TimeStart As Double           'Start time for DirectX8Event loop
Dim TimeEnd As Double             'Ending time for DirectX8Event loop
Dim AvgCtr As Long                'Counts number of events to average
Dim AvgTime As Double             'Stores the average event cycle time

'Set up Event variables for the Capture Buffer
Implements DirectXEvent8          'Allows DirectX Events
Dim hEvent(1) As Long             'Handle for DirectX Event
Dim EVNT(1) As DSBPOSITIONNOTIFY        'Notify position array
Dim Receiving As Boolean          'In Receive mode if true
Dim FirstPass As Boolean          'Denotes first pass from Start
```

**Fig 5—Declaration of variables, buffers, events and objects. This code is located in the General section of the module or form.**

from the Microsoft Web site (see Note 3). Once the driver and SDK are installed, you will need to register the DirectX8 for *Visual Basic* Type Library within the *Visual Basic* development environment.

If you are building the project from scratch, first create a *Visual Basic* project and name it "Sound." When the project loads, go to the Project Menu/References, which loads the form shown in Fig 4. Scroll through Available References until you locate the DirectX8 for *Visual Basic* Type Library and check the box. When you press "OK," the library is registered.

### *Define Variables, Buffers and DirectX Objects*

Name the form in the Sound project frmSound. In the General section of frmSound, you will need to declare all of the variables, buffers and DirectX objects that will be used in the driver interface. Fig 5 provides the code that is to be copied into the General section. All definitions are commented in the code and should be self-explanatory when viewed in conjunction with the subroutine code.

### *Create the Audio Devices*

We are now ready to create the DirectSound objects and set up the format of the capture and play buffers. Refer to the source code in Fig 6 during the following discussion.

The first step is to create the DirectSound and DirectSoundCapture

```
'Set up the DirectSound Objects and the Capture and Play Buffers
Sub CreateDevices()

    On Local Error Resume Next

    Set ds = dx.DirectSoundCreate(vbNullString)      'DirectSound object
    Set dsc = dx.DirectSoundCaptureCreate(vbNullString)    'DirectSound Capture

    'Check to se if Sound Card is properly installed
    If Err.Number <> 0 Then
        MsgBox "Unable to start DirectSound. Check proper sound card installation"
       End
    End If

    'Set the cooperative level to allow the Primary Buffer format to be set
    ds.SetCooperativeLevel Me.hWnd, DSSCL_PRIORITY

    'Set up format for capture buffer
    With dscbd
        With .fxFormat
            .nFormatTag = WAVE_FORMAT_PCM
            .nChannels = 2                      'Stereo
            .lSamplesPerSec = Fs                'Sampling rate in Hz
            .nBitsPerSample = 16                '16 bit samples
            .nBlockAlign = .nBitsPerSample / 8 * .nChannels
            .lAvgBytesPerSec = .lSamplesPerSec * .nBlockAlign
        End With
        .lFlags = DSCBCAPS_DEFAULT
        .lBufferBytes = (dscbd.fxFormat.nBlockAlign * CAPTURESIZE) 'Buffer Size
        CaptureBytes = .lBufferBytes \ 2      'Bytes for 1/2 of capture buffer
    End With

    Set dscb = dsc.CreateCaptureBuffer(dscbd)       'Create the capture buffer

    ' Set up format for secondary playback buffer
    With dsbd
        .fxFormat = dscbd.fxFormat
        .lBufferBytes = dscbd.lBufferBytes * 2      'Play is 2X Capture Buffer Size
        .lFlags = DSBCAPS_GLOBALFOCUS Or DSBCAPS_GETCURRENTPOSITION2
    End With

    dspbd = dsbd.fxFormat                       'Set Primary Buffer format
    dspb.SetFormat dspbd                        'to same as Secondary Buffer

    Set dsb = ds.CreateSoundBuffer(dsbd)       'Create the secondary buffer

End Sub
```

**Fig 6—Create the DirectX capture and playback devices.**

objects. We then check for an error to see if we have a compatible sound card installed. If not, an error message would be displayed to the user. Next, we set the cooperative level DSSCL_ PRIORITY to allow the Primary Buffer format to be set to the same as that of the Secondary Buffer. The code that follows sets up the DirectSoundCaptureBufferDescription format and creates the DirectSoundCaptureBuffer object. The format is set to 16-bit stereo at the sampling rate set by the constant *F*s.

Next, the DirectSoundBufferDescription is set to the same format as the DirectSoundCaptureBufferDescription. We then set the Primary Buffer format to that of the Secondary Buffer before creating the DirectSoundBuffer object.

### *Set the DirectX Events*

As discussed earlier, the DirectSoundCaptureBuffer is divided into two blocks so that we can read from one block while capturing to the other. To do so, we must know when

```
'Set events for capture buffer notification at 0 and 1/2
Sub SetEvents()

    hEvent(0) = dx.CreateEvent(Me)          'Event handle for first half of buffer
    hEvent(1) = dx.CreateEvent(Me)          'Event handle for second half of buffer

    'Buffer Event 0 sets Write at 50% of buffer
    EVNT(0).hEventNotify = hEvent(0)
    EVNT(0).lOffset = (dscbd.lBufferBytes \ 2) - 1 'Set event to first half of capture buffer

    'Buffer Event 1 Write at 100% of buffer
    EVNT(1).hEventNotify = hEvent(1)
    EVNT(1).lOffset = dscbd.lBufferBytes - 1       'Set Event to second half of capture buffer

    dscb.SetNotificationPositions 2, EVNT()        'Set number of notification positions to 2

End Sub
```

**Fig 7—Create the DirectX events.**

```
'Create Devices and Set the DirectX8Events
Private Sub Form_Load()
    CreateDevices                       'Create DirectSound devices
    SetEvents                           'Set up DirectX events
End Sub

'Shut everything down and close application
Private Sub Form_Unload(Cancel As Integer)

    If Receiving = True Then
        dsb.Stop                        'Stop Playback
        dscb.Stop                       'Stop Capture
    End If

    Dim i As Integer
    For i = 0 To UBound(hEvent)         'Kill DirectX Events
        DoEvents
        If hEvent(i) Then dx.DestroyEvent hEvent(i)
   Next

    Set dx = Nothing                    'Destroy DirectX objects
    Set ds = Nothing
    Set dsc = Nothing
    Set dsb = Nothing
    Set dscb = Nothing

    Unload Me

End Sub
```

**Fig 8—Create and destroy the DirectSound Devices and events.**

DirectX has finished writing to a block. This is accomplished using the DirectXEvent8. Fig 7 provides the code necessary to set up the two events that occur when the lWrite cursor has reached 50% and 100% of the DirectSoundCaptureBuffer.

We begin by creating the two event handles hEvent(0) and hEvent(1). The code that follows creates a handle for each of the respective events and sets them to trigger after each half of the DirectSoundCaptureBuffer is filled. Finally, we set the number of notification positions to two and pass the name of the EVNT() event handle array to DirectX.

The CreateDevices and SetEvents subroutines should be called from the Form_Load() subroutine. The Form_Unload subroutine must stop capture and playback and destroy all of the DirectX objects before shutting down. The code for loading and unloading is shown in Fig 8.

### *Starting and Stopping Capture/Playback*

Fig 9 illustrates how to start and stop the DirectSoundCaptureBuffer. The dscb.Start DSCBSTART_ LOOPING command starts the DirectSoundCaptureBuffer in a continuous circular loop. When it fills the first half of the buffer, it triggers the DirectX Event8 subroutine so that the data can be read, processed and sent to the DirectSoundBuffer. Note that the DirectSoundBuffer has not yet been started since we will quadruple buffer the output to prevent processor loading from causing gaps in the output. The FirstPass flag tells the event to start filling the DirectSoundBuffer for the first time before starting the buffer looping.

### *Processing the Direct-XEvent8*

Once we have started the DirectSoundCaptureBuffer looping, the completion of each block will cause the DirectX Event8 code in Fig 10 to be executed. As we have noted, the events will occur when 50% and 100% of the buffer has been filled with data. Since the buffer is circular, it will begin again at the 0 location when the buffer is full to start the cycle all over again. Given a sampling rate of 44,100 Hz and 2048 samples per capture block, the block rate is calculated to be 44,100/2048 = 21.53 blocks/s or one block every 46.4 ms. Since the quad buffer is filled before starting playback the total delay from input to output is $4 \times 46.4$ ms = 185.6 ms.

The DirectX Event8_DXCallback event passes the eventid as a variable. The case statement at the beginning of the code determines from the eventid, which half of the DirectSoundCaptureBuffer has just been filled. With that information, we can calculate the starting address for reading each block from the DirectSoundCaptureBuffer to the inBuffer() array with the dscb. ReadBuffer command. Next, we simply pass the inBuffer() to the external DSP subroutine, which returns the processed data in the outBuffer() array.

Then we calculate the StartAddr and EndAddr for the next write location in the DirectSoundBuffer. Before writing to the buffer, we first check to make sure that we are not writing between the lWrite and lPlay cursors, which will cause portions of the buffer to be overwritten that have already been committed to the output. This will result in noise and distortion in the audio output. If an error occurs, the FirstPass flag is set to true and the pointers are reset to zero so that we flush the DirectSoundBuffer and start over. This effectively performs an automatic reset when the processor is overloaded, typically because of graphics intensive applications running alongside the SDR application.

If there are no overwrite errors, we write the outBuffer() array that was returned from the DSP routine to the next StartAddr to EndAddr in the DirectSoundBuffer. Important note: In the sample code, the DSP subroutine call is commented out and the inBuffer() array is passed directly to the DirectSoundBuffer for testing of the code. When the FirstPass flag is set to True, we capture and write four data blocks before starting playback looping with the .SetCurrentPosition 0 and .Play DSBPLAY_LOOPING commands.

The subroutine calls to StartTimer and StopTimer allow the average computational time of the event loop to be displayed in the immediate window. This is useful in measuring the efficiency of the DSP subroutine code that is called from the event. In normal operation, these subroutine calls should be commented out.

### *Parsing the Stereo Buffer into I and Q Signals*

One more step that is required to use the captured signal in the DSP subroutine is to separate or parse the left and right channel data into the *I* and *Q* signals, respectively. This can be accomplished using the code in Fig 11. In 16-bit stereo, the left and right channels are interleaved in the inBuffer() and outBuffer(). The code simply copies the alternating 16-bit integer values to the RealIn()), (same as *I*) and ImagIn(), (same as *Q*) buffers respectively. Now we are ready to perform the magic of digital signal processing that we will discuss in the next article of the series.

### Testing the Driver

To test the driver, connect an audio generator—or any other audio device, such as a receiver—to the line input of the sound card. Be sure to mute line-in on the mixer control panel so that you will not hear the audio directly through the operating system. You can open the mixer by double clicking on the speaker icon in the lower right corner of your Windows screen. It is also accessible through the Control Panel.

Now run the Sound application and press the On button. You should hear the audio playing through the driver. It will be delayed about 185 ms from the incoming audio because of the quadruple buffering. You can turn the mute control on the line-in mixer on and off to test the delay. It should sound like an echo. If so, you know that everything is operating properly.

### Coming Up Next

In the next article, we will discuss in detail the DSP code that provides

```
'Turn Capture/Playback On
Private Sub cmdOn_Click()
    dscb.Start DSCBSTART_LOOPING     'Start Capture Looping
    Receiving = True                 'Set flag to receive mode
    FirstPass = True                 'This is the first pass after
Start
    OutPtr = 0                       'Starts writing to first buffer
End Sub

'Turn Capture/Playback Off
Private Sub cmdOff_Click()
    Receiving = False                'Reset Receiving flag
    FirstPass = False                'Reset FirstPass flag
    dscb.Stop                        'Stop Capture Loop
    dsb.Stop                         'Stop Playback Loop
End Sub
```

**Fig 9—Start and stop the capture/playback buffers.**

```
'Process the Capture events, call DSP routines, and output to Secondary Play Buffer
Private Sub DirectXEvent8_DXCallback (ByVal eventid As Long)

    StartTimer                              'Save loop start time

    Select Case eventid                     'Determine which Capture Block is ready
        Case hEvent(0)
            InPtr = 0                       'First half of Capture Buffer
        Case hEvent(1)
            InPtr = 1                       'Second half of Capture Buffer
    End Select

    StartAddr = InPtr * CaptureBytes        'Capture buffer starting address

    'Read from DirectX circular Capture Buffer to inBuffer
    dscb.ReadBuffer StartAddr, CaptureBytes, inBuffer(0), DSCBLOCK_DEFAULT

    'DSP Modulation/Demodulation - NOTE: THIS IS WHERE THE DSP CODE IS CALLED
'    DSP inBuffer, outBuffer

    StartAddr = OutPtr * CaptureBytes       'Play buffer starting address
    EndAddr = OutPtr + CaptureBytes - 1     'Play buffer ending address

    With dsb                                'Reference DirectSoundBuffer

            .GetCurrentPosition PlyCurs     'Get current Play position

            'If true the write is overlapping the lWrite cursor due to processor loading
            If PlyCurs.lWrite >= StartAddr _
                And PlyCurs.lWrite <= EndAddr Then
                    FirstPass = True        'Restart play buffer
                    OutPtr = 0
                    StartAddr = 0
            End If

            'If true the write is overlapping the lPlay cursor due to processor loading
            If PlyCurs.lPlay >= StartAddr _
                And PlyCurs.lPlay <= EndAddr Then
                    FirstPass = True        'Restart play buffer
                    OutPtr = 0
                    StartAddr = 0
            End If

        'Write outBuffer to DirectX circular Secondary Buffer. NOTE: writing inBuffer causes
direct pass through. Replace
        'with outBuffer below to when using DSP subroutine for modulation/demodulation
        .WriteBuffer StartAddr, CaptureBytes, inBuffer(0), DSBLOCK_DEFAULT

        OutPtr = IIf(OutPtr >= 3, 0, OutPtr + 1)       'Counts 0 to 3

        If FirstPass = True Then            'On FirstPass wait 4 counts before starting
            Pass = Pass + 1                 'the Secondary Play buffer looping at 0
            If Pass = 3 Then                'This puts the Play buffer three Capture cycles
                FirstPass = False           'after the current one
                Pass = 0                    'Reset the Pass counter
                .SetCurrentPosition 0       'Set playback position to zero
                .Play DSBPLAY_LOOPING       'Start playback looping
            End If
        End If

    End With

    StopTimer                               'Display average loop time in immediate window

End Sub
```

**Fig 10—Process the DirectXEvent8 event. Note that the example code passes the inBuffer() directly to the DirectSoundBuffer without processing. The DSP subroutine call has been commented out for this illustration so that the audio input to the sound card will be passed directly to the audio output with a 185 ms delay. Destroy objects and events on exit.**

```
Erase RealIn, ImagIn

    For S = 0 To CAPTURESIZE - 1 Step 2        'Copy I to RealIn and Q to ImagIn
        RealIn(S \ 2) = inBuffer(S)
        ImagIn(S \ 2) = inBuffer(S + 1)
    Next S
```

**Fig 11—Code for parsing the stereo inBuffer() into in-phase and quadrature signals. This code must be imbedded into the DSP subroutine.**

modulation and demodulation of SSB signals. Included will be source code for implementing ultra-high-performance variable band-pass filtering in the frequency domain, offset baseband IF processing and digital AGC.

**Notes**

[1]G. Youngblood, AC5OG, "A Software-Defined Radio for the Masses: Part 1," *QEX*, July/Aug 2002, pp 13-21.

[2]Information on both DirectX and Windows Multimedia programming can be accessed on the Microsoft Developer Network (MSDN) Web site at **www.msdn. microsoft.com/library**. To download the DirectX Software Development Kit go to **msdn.microsoft.com/downloads/** and click on "Graphics and Multimedia" in the left-hand navigation window. Next click on "DirectX" and then "DirectX 8.1" (or a later version if available). The DirectX runtime driver may be downloaded from **www.microsoft.com/windows/directx/downloads/default.asp**.

[3]You can download this package from the ARRL Web **www.arrl.org/qexfiles/**. Look for 0902Youngblood.zip. □□

# *Amateur Radio Software: It Keeps Getting Better*

*The tools to develop Amateur Radio applications steadily improve.*

By Stephen J. Gradijan, WB5KIA

My first effort in programming software for Amateur Radio was a program written 20 years ago in *BASIC*. It provided accurate calculations but it was not very pretty, as its graphic capabilities were limited. Today, amateurs are creating commercial and near-commercial-quality Amateur Radio software to do all kinds of things and are making it free to the amateur community. Not all hams programming are professional programmers. High-level programming is easier to do than ever, and personal computers are permitting execution of mathematical routines that would have brought the PCs of 10 years ago to their knees.

Sophisticated graphics are within the capabilities of the novice programmer. Many ham-developed computer programs are available at no cost to the amateur—they track satellites, log contacts, predict radio propagation and support every digital mode available. Until recently, ham-oriented freeware and shareware could be found on the Internet, but without its source code (the code required to modify the program). Most of the programs came with only the executable code. State-of-the-art programming-code examples of ham-radio processes, with a few exceptions, were difficult to find. Today, some amateurs are freely sharing source code with others and even sharing the program-development experience.

The purposes of this article are to show that it is relatively easy to write code with modern programming packages and point out some options for obtaining and using software for your projects.

Internet sites like N1MM's logging project, WA0TTN and AE4YJ's pages for development of PSK, SV2AGW's AGWPacket Engine site for development of packet communications and JE3HHT's MMTTY RTTY pages feature ham programming at its best. Such sites let radio amateurs, worldwide, get involved with the program development, have access to the source code or to a control or DLL to use with your program. Most of what is available is free of charge and usually subject to very generous fair-use terms. Information on the sites describes how the material may be used. The source material available at these sites can shorten the program-development time of your project or help a beginning programmer discover how to do something that otherwise might seem impossible.

1902 Middle Glen Dr
Carrollton, TX 75007
**wb5kia@arrl.net**

### Software on the Web

The *N1MM Logger* (Fig 1) Web site is at **n1mm.com**. On the N1MM Yahoo Discussion Group, one can request new features to the current version of the versatile N1MM logging program and discuss problems (called bugs) with the current logging program. Several hams help Tom Walker, the program's principal developer, develop new code and documentation. Tom and his crew are a cooperative development venture, programming in *Visual Basic 6*. *N1MM Logger* uses Microsoft *Access* files for the logging database. However, most of the code can be read using a simple text editor such as *Notepad,* which comes with the various versions of Windows. Routines can be used in various projects that you might program in another version of *Visual Basic* or that could be translated into another language.

JE3HHT has made the dynamic-link library he programmed for

*MMTTY* available for use by radio amateurs. The *MMTTY* page is at **www.qsl.net/mmhamsoft/mmtty/**. It has a copy of the RTTY engine and examples for *Visual Basic 6* and Borland *C++* developers. Dynamic link libraries (DLLs) are program modules that contain code, data or resources that can be shared among many Windows applications. They can be thought of as compact programs that can be accessed by various Windows programs to provide specific functions. Makato's DLL allows others to incorporate his RTTY engine in their own programs.

Moe Wheatley, AE4YJ, has made his *PSK Core* DLL available for amateur use. Dave Cook, WA0TTN, took Moe's idea farther and created a *WinPSK* ActiveX control (Fig 2) that can be used with a variety of programming platforms. Moe has also made available the original code on which the *PSK Core* DLL is based. His links to a *Visual Basic 6* demonstration by Eric Sundstrup, VK7AAB, have disappeared but a Delphi demonstration by Julian Moss, G4ILO, is still linked at his site **www.qsl.net/ae4jy/pskcoredll.htm**. G4ILO has his own site, **www.qsl.net/g4ilo/main.html** with additional code and programs. WA0TTN's site is **www.netdave.com/wa0ttn**. The effort that went into programming the DLL and ActiveX control was awesome. All these tools are free for amateur use.

You can write your own programs too. It takes time to become familiar with the programming languages, but once you get started, programming can be fun. Hamming and computer programming do go together!

### History

For decades, radio amateurs have made use of computers to enhance their operating with programs to calculate engineering values, log contacts and keep track of awards and QSLs, operate packet, RTTY or other digital modes, predict radio propagation conditions and track Amateur Radio satellites.

Efforts in the 1960s involved mainframe computers, the FORTRAN language and punch cards, but few hams were able to get access to the institutional computers or do the necessary work. Beginning in the 1970s and with the advent of personal computers and the spread of the *BASIC* language in its various forms, hams began to program in earnest.

It was practical in the early days to publish *BASIC* code listings in magazines. In 1981, Tom Clark, W3IWI, wrote a satellite tracking program described in an article called "*BASIC* Orbits" (*Orbit*, March/April 1981, pp 10-11, 19-20, 29). It was very popular and became the basis for many later Windows programs. *QST* contained the *MINIMUF* propagation program and the *Super Duper* logging program in 1982 and 1985, respectively. Keyboarding of long programs was a chore and error-prone. After 1986, program listings disappeared from most journals, as shareware and freeware programs became available on telephone and packet bulletin-board systems (BBSs), disks and CD ROMs. This was a huge improvement in the ability to share software code.

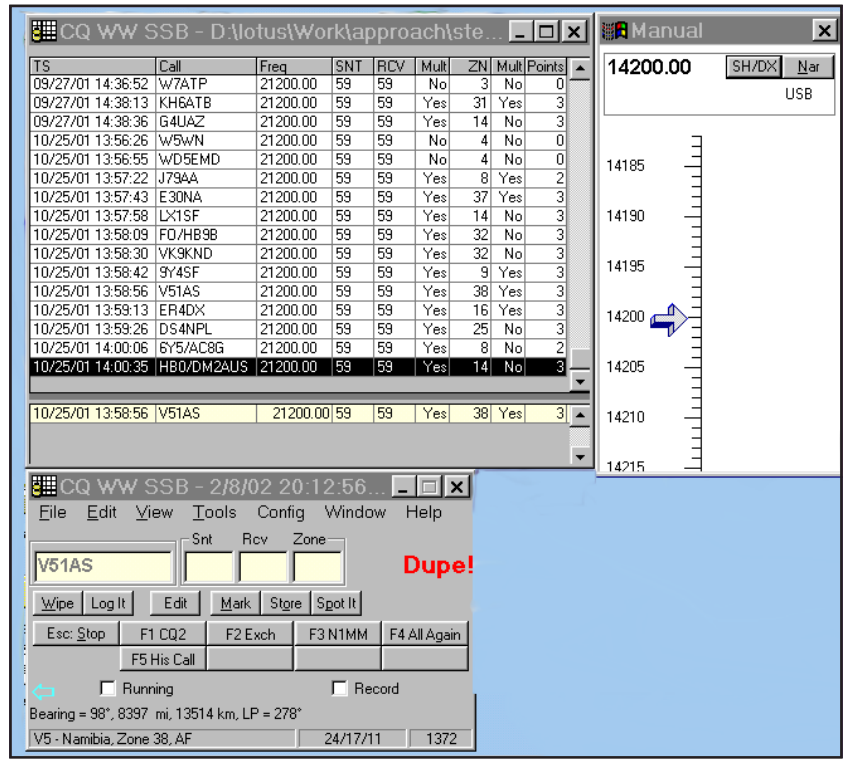



**Fig 1—The *N1MM Logger* is being developed by ham users through exchange of comments and ideas at an Internet discussion Web site. *Visual Basic 6* is the main tool for development of the project.**

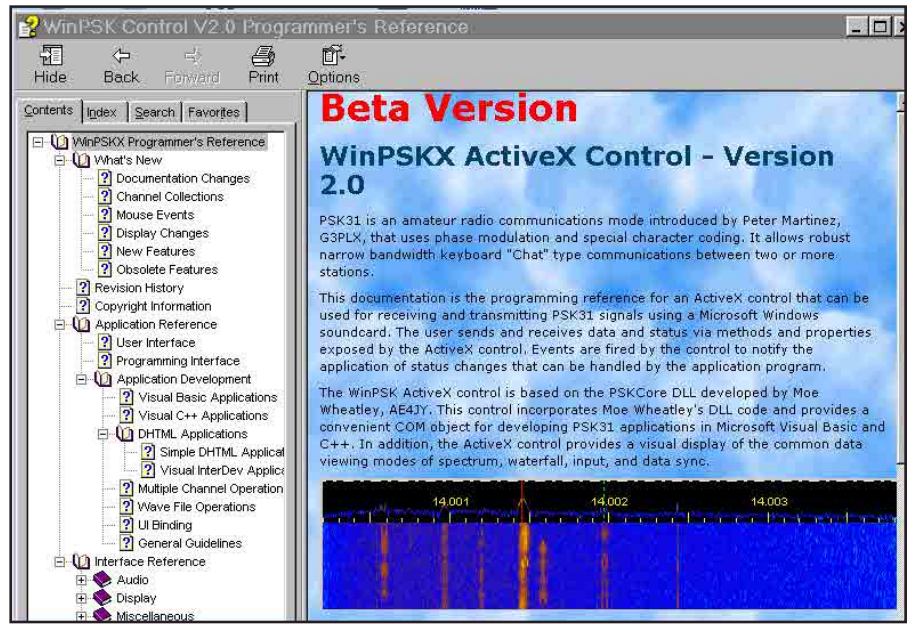



**Fig 2—WA0TNN's documentation for the *WinPSK* ActiveX control he wrote based on AE4JY's PSK Core DLL is very thorough. It works with *Visual Basic*, *Delphi*, *C++* and such.**

*BASIC* was the primary tool for nonprofessional programmers. Professional programmers used other high-level languages like Pascal and FORTRAN (in its various forms) on PCs, along with assembly language.

Early Windows-environment programmers used C and C+ followed by *Visual Basic* in about 1991. It was very difficult for the amateur to write a program for Windows. *Visual Basic* changed that as it became relatively easy to program, so Windows-based ham programs began to appear. It is difficult to provide Windows program listings because it is impractical to print a listing of the graphical properties that are attached to specific controls. Consequently it became difficult to print the source code listings in magazines.

Since 1995 and the creation of *Windows 95*, tools like *Visual Basic* and *Delphi* have created 32-bit programs that can execute at lightning speed on today's fast PCs.

### My Experience with Programming Projects

I have developed numerous programs through the years for my personal enjoyment using interpreted *BASIC*, Computer Associates *Realizer*, *Visual Basic*, *C++, Delphi 1* and *Delphi 5*. Today, I program almost exclusively with *Delphi 5 Professional*.

Many of my DOS and Windows programs were near commercial quality at the time they were written, but many were only partially fleshed out or documented. I knew how to run them and did not require a polished finished product, but only the functionality. Similar programs were available commercially but my own programs cost me only my time and resulted in a better understanding of the basis for the programs (their underlying mathematics and principles that allowed execution).

I programmed antenna pointing, satellite tracking and logging programs, a program to control my PK-232 Multimode controller, Technician and General Class License study guides, and so forth. My most recent effort is a PSK program with an attached logger (see Fig 3). Although I may not be typical, I believe it is possible for you to write programs for yourself too!

*KIApsk Logger* is a PSK program I developed recently because the transmit and receive frequencies of my aging ICOM IC-740 are off by 16 Hz (or there may be a problem with input versus output frequencies of my clone sound card on my six-year-old computer). Using any of the excellent available PSK software programs, I constantly received complaints of slightly off-frequency operation or "you must have left your RIT control on." I do not get such reports any more.

*KIApsk Logger* uses the *WinPSK* ActiveX control (copyrighted in 2001 by Dave Cook, WA0TTN and Moe Wheatley, AE4JY) based on the *PSKCore* DLL developed by Moe Wheatley. The ActiveX control incorporates Moe Wheatley's DLL code. It provides a convenient COM object for developing PSK31 applications in Microsoft *Visual Basic*, *Delphi* and *C++*. The ActiveX control provides a visual display of the common data viewing modes of spectrum, waterfall, input and data sync. The program is similar to most of the available PSK programs today; the difference is that I coded it

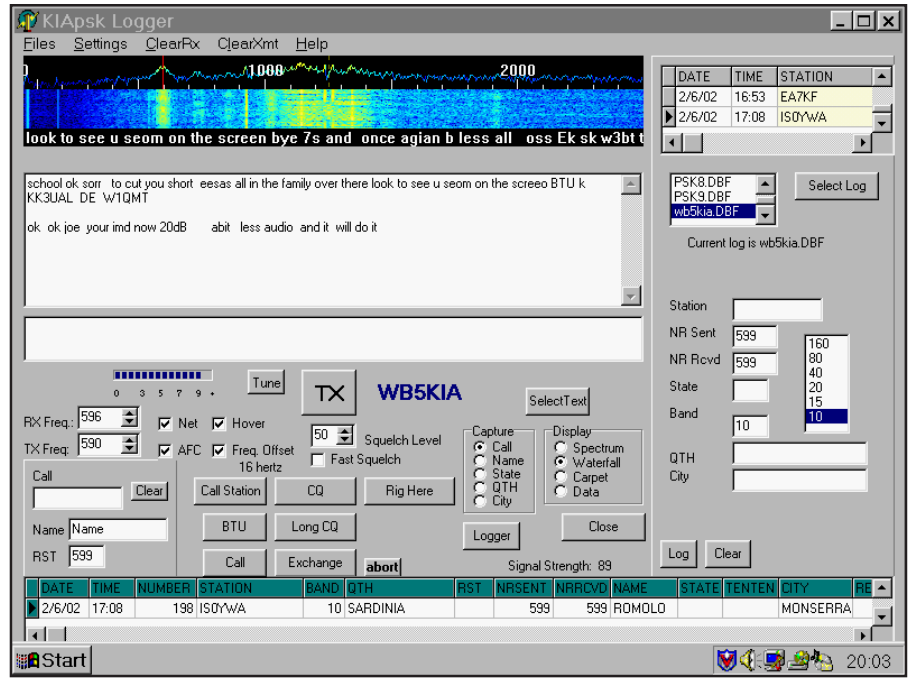



**Fig 3—This is the main screen of the *KIApsk Logger* that was programmed with *Delphi 5* and using the ActiveX control by WA0TTN described in the text.**

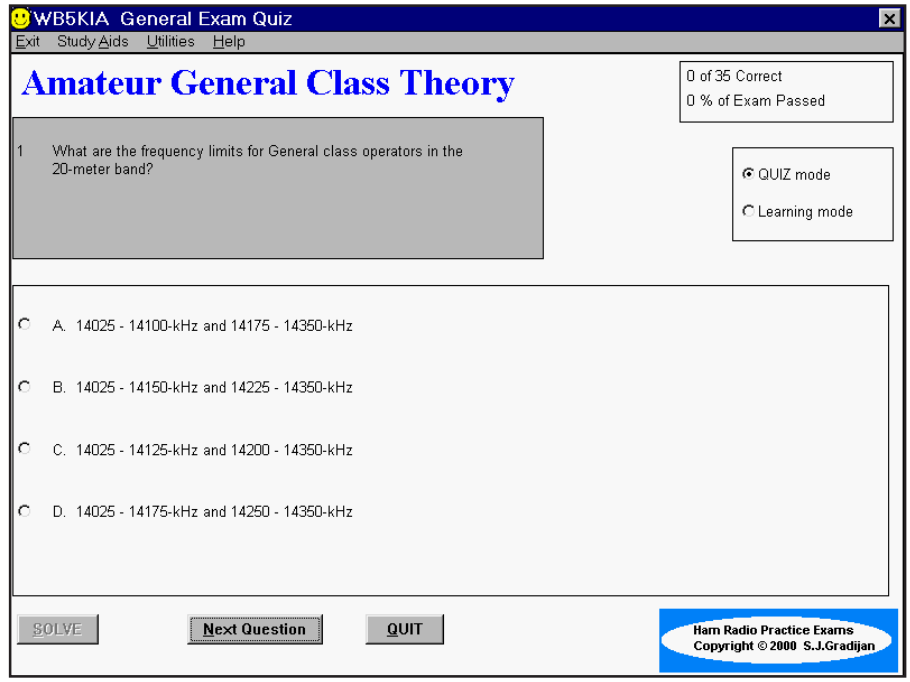



**Fig 4—A General class license quiz generator and study guide was easily programmed with *Realizer*.**

and it does what I want it to.

My son Francis, KD5HTB, found it relatively easy to pass his Technician Plus and General exams using software learning programs I developed using the ARRL/VEC question pool. I created a trial exam generating program and study guide using *Realizer* (see Fig 4). This is one of the simplest types of program. Plenty of material was available on the Internet with source files showing various ways of creating generic test generating programs for educators. Potential program developers of educational ham programs have a wealth of example information and code available.

Putting together various pieces of code can result in new and useful programs. You might need to modify the code for your version of the language or even for a different programming environment. For several years, I used a propagation forecast/beam-heading program based on a modification of the *BASIC* code in the *MINIMUF QST* article, code modified from various other magazines and independently developed code (see Fig 5). The initial code came from the article; the rest came through curiosity and a desire to have additional features.

### Programming Tools

What is needed to get a start in programming for Amateur Radio may be as simple as acquiring a copy of *QBasic*, *QuickBasic*, *PowerBasic*, *Visual Basic* or *Turbo Pascal* for DOS programming. *BASIC* interpreters come in various forms such as *GWBasic*, *BasicA* and so on. Microsoft *Qbasic* is an interpreted version of its compiled *QuickBasic*. The main difference is that *Qbasic* requires the presence of the *Qbasic* interpreter to run the program code. *QuickBasic* can create an executable file that is self-contained. It also has more capabilities. *PowerBasic* by Powerbasic is offering similar to *QuickBasic*. The original *Visual Basic* was a DOS programming tool with expanded graphics capabilities. *Turbo Pascal* is a Borland product using the Pascal programming language. Pascal and *BASIC* have both similarities and differences. Noncommercial versions of both the original *BASIC* and Pascal languages have been developed and can be found for downloading on the Internet. While interpreters can be used to code complex problems, they probably should be avoided today because everyone using the program has to have a copy of the interpreter that was used to develop it. On the other hand, the actual *BASIC* program s readable by anyone who has a copy of *BASIC* or a text editor to read the code. Compiled languages are preferred because they create an executable file and process code faster.

Although some versions of the DOS programming languages have enhanced graphics abilities, DOS programming for ham radio purposes is limited to text output and very simple graphic displays (Fig 6). Most programs that you would code using DOS will run under Windows in a DOS Window, so do not exclude the DOS compiled languages just because they are old technology. If you do not require lightning speed or fancy graphics, you should be quite happy with DOS.

However, Windows programming may be easier to learn. This was not always true. Early Windows programs were coded in *C* and *C++*; development was difficult and slow. The visual programming tools developed for *Windows 3.1* that made programming simpler include *Visual Basic 3*, *Delphi 1* and *Realizer*. These languages are based on the concept of visual objects. Components that provide basic functions can be dragged onto a form in the programming environment. One can modify

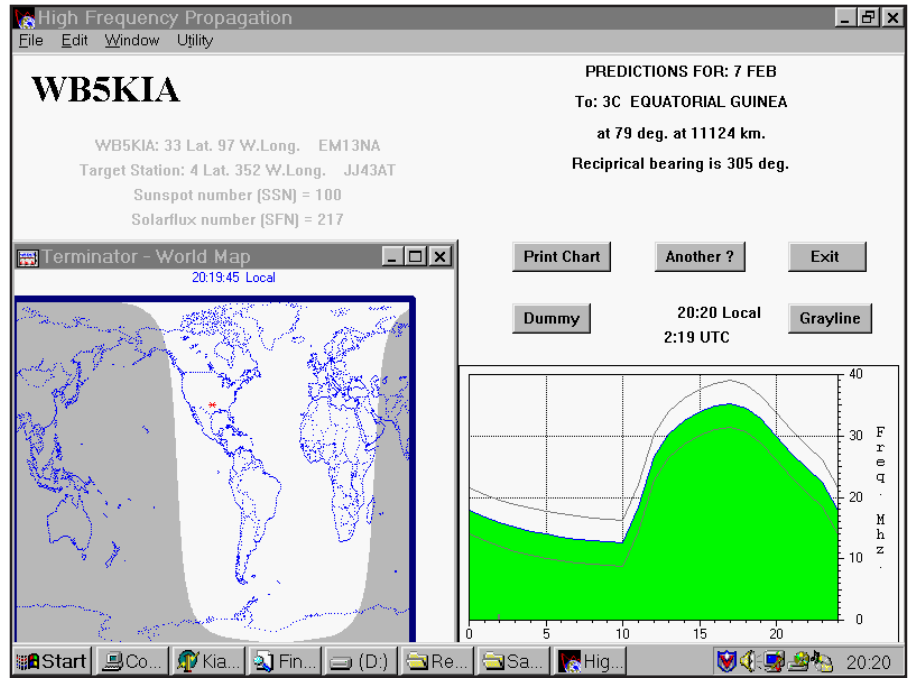



**Fig 5—This highly updated and improved *MINIMUF* HF propagation program was coded with *Realizer*.**

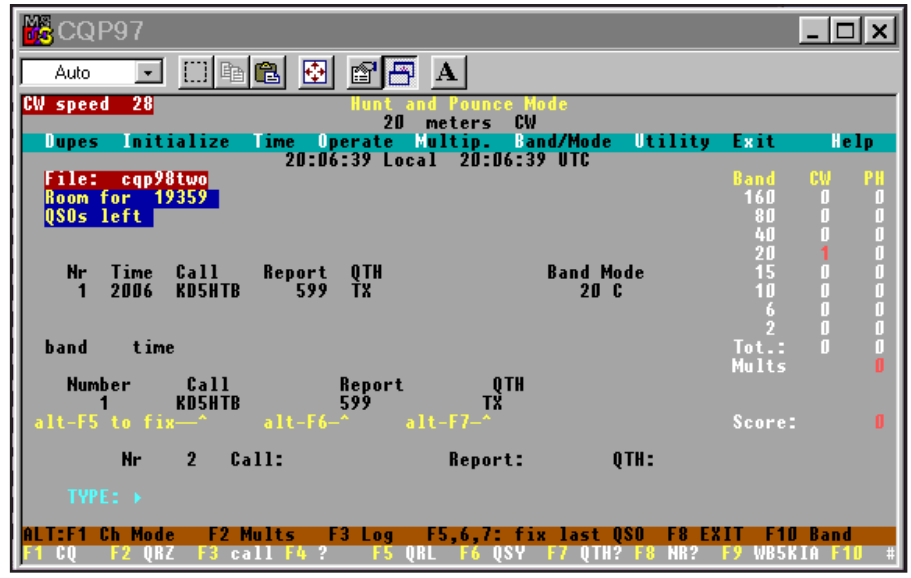



**Fig 6—A DOS contest logger running in a Windows DOS window was programmed with *QuickBasic 4.5*.**

their characteristics from a table and avoid detailed coding for many usual tasks. *Visual Basic 3* and *Realizer* are based on the *BASIC* language. *Delphi 1* is based on Pascal.

Tools developed for *Windows 95* and later operating systems running on 32-bit processors include *Visual Basic 4-6*, *Delphi 2-6* and *Visual C++*. The 16-bit tools can run on 32-bit processors but cannot take advantage of all the increases in processing speed and features of the modern processors and operating systems. Both 16-bit and 32-bit programming systems have enhanced graphics capabilities; they can be used to produce very good-looking programs (see Fig 7).

Tools for Apple users are more difficult to come by and are beyond the scope of this article. A *BASIC* language was available at one time for Apple users. *Linux* users might consider *Kylix* by Borland. It is a *Delphi*-like clone developed by Borland for the *Linux* operating system.

Table 1 describes some of the tools that you might use for your programming activities and a few can be seen in Figs 8, 9, 10.

These programming tools may be available free of charge or used at reasonable prices, especially if the software is designed for 16-bit Windows or DOS programming. Check the terms of the individual software license. Licenses may be transferred in most instances if the media is exchanged and the programs have been removed from the original owner's computers. Copies of *Delphi 1* were made available at no charge in the United Kingdom a few years ago but with the stipulation that programs generated were not for distribution, that is, only for personal use. Copies of *Turbo Pascal* and *Turbo C++* are available free for personal use from the Borland Museum (**www.community.Borland.com**) if you register. Whatever programming language you choose or find available, make sure the software license terms allow you to either sell or give away the resulting program before you start sharing your programs.

Used bookstores frequently have copies of software available in its original packaging at discounted prices. Various E-vendors have last year' s version of some programming software at discounted or reduced prices. Recently, I have found "Professional" editions of *Visual Basic 4* at under $50 and of *Visual Basic 3* and *Realizer* under $20 in a national used-book chain. They were sealed in their original packaging. New "starter" versions of the latest editions of *Visual Basic* and *Delphi* are available for around $100. These are very usable versions of the larger professional or enterprise programs but without all of their parent-program features. The license terms vary with the various versions but most do not permit commercial distribution of the developed programs. The good news is that it is possible to upgrade to the full version of many of these products after you have developed confidence in your programming skills. Also, free or shareware tools are available on the Internet that augment the functionality of these cut-down versions of the professional software packages.

Table 2 is a compilation of Internet sites that have information and resources that can help you with your programming experience. The Internet site addresses frequently change, so be resourceful if they do. If you cannot find what is described at the indicated sites, use a search engine to find what you need.

What is the best language for you? Most of the DOS-based languages are very easy to learn and may be practical if you are comfortable with DOS or have an ancient machine. DOS programs, with a few exceptions, do run on computers having a Windows operating system. Although it is possible to use graphics, it is not as easy as with the Windows programming tools. DOS is fine for "number crunching" applications but consider the arithmetic precision possible with the language and/or program you select.

Windows programming tools come in several flavors. Some are more user friendly to beginners than others. Some tools provide executable files that have a significant speed advantage. Programs designed with 16-bit pro-

**Table 1—Some Programming Tools**

| *Language* | *Description* | *Source* | *Type* | *Ease of Use* |
|---|---|---|---|---|
| ***For DOS*** | | | | |
| *GWBasic** | Simple *BASIC* | Microsoft | Interpreter | Easy |
| *Qbasic** | Advanced *BASIC* | Windows disks | Interpreter | Easy |
| *QuickBasic** | Structured *BASIC* | Microsoft | Compiler | Relatively Easy |
| *PowerBasic* | Structured *BASIC* | PowerBASIC Inc | Compiler | Relatively Easy |
| *Turbo Pascal** | Pascal language | Borland | Compiler | Intermediate difficulty |
| ***For Microsoft Windows*** | | | | |
| *Visual Basic* | Object-oriented structured *BASIC* | Microsoft | Compiler with runtime DLL | Relatively Easy |
| *Realizer** | Object-oriented structured *BASIC* | Computer Associates | Compiler with runtime DLL | Easy |
| *Delphi 1* | 16-bit Visual Pascal | Borland | Compiler | Relatively Easy |
| *Delphi 2-6* | 32-bit Visual Pascal | Borland | Compiler | Relatively Easy |
| *C++* | various flavors | Borland or Microsoft | Compiler | Difficult |
| *Visual C++* | various flavors | Borland or Microsoft | Compiler | Relatively Difficult |
| ***Linux*** | | | | |
| *Kylix* | 32-bit Visual Pascal | Borland | Compiler | Relatively Easy |
| *Visual C++* | | Various | Compiler | Difficult |

*These languages are no longer supported by the companies that originated them, but there is some community support on the Web. They are no longer sold at retail.

**Table 2—Internet Sites with Programming Information and/or Source Code**

***General Amateur Radio Resources***

N1MM Logger
**www.n1mm.com**
Development of contest logging program. Specific source code is available by request. A great site/discussion group

WA0TTN Web page
**www.netdave.com/wa0ttn**
PSK ActiveX control for *Visual Basic*, *Delphi* and *C++*, other controls to make developing PSK programs easier. A remarkable free tool for working with a homemade PSK program

PSK Core DLL
**www.qsl.net/ae4jy/pskcoredll.htm**
PSK Core DLL and example code for *Delphi*. Free DLL works fine.

MMTTY
**www.qsl.net./mmhamsoft**
RTTY DLL by Makoto Mori, JE3HHT—amazing RTTY

MMTTY Programmer's Page
**www.qsl.net/mmhamsoft/programmer/p-download.htm**
Examples of how to use the DLL

SV2AGW Sound-Card Packet
**www.elcom.gr/sv2agw/agwsc.htm**
Packet engine (Packet without TNC hardware)—difficult to use.

HB9JNX/AE4WA Multplatform Sound-Card Packet
**www.baycom.org/~tom/ham/soundmodem**
Another sound-card packet implementation—untested.

Official PSK Web site
**www.aintel.bi.ehu.es/psk31.html**
Code snippets and partial program showing how to use AE4JYs PSK Core DLL with *Visual Basic* and *Delphi*. Many examples of quality programs coded by hams

DX Atlas (shareware)
**www.dxatlas.com**
Advice on how to interface your project to the DX Atlas via COM or OLE automation is in the program's help file. Two *Delphi 5* example programs with code. Example of techniques to interface your project with another project. One of the numerous shareware/ commercial sites that provide information for developers to interface with their products.

Ten-Tec Programers' Page
**tentec.com/rfsquared**
Follow the *update* links to the *Ten-Tec Programmer's Reference Guide* for programming Pegasus and Jupiter radios. General public license source code (Microsoft *Visual C++*, 16-bit) for control of Pegasus/Jupiter and reference guide.

ICOM America
**icomamerica.com**

Kenwood*
**Kenwood.net/ama_page.cfm**
In downloads section, "Software" has several free rig-memory programming applications. Amateur/RCPSoftware section has a free TS-570 control program that also works with many TS-2000 features.

Yaesu*
**Yaesu.com/amateur/amateur.html** or **soft.html**
Only commercial software listed.

The Plicht Brothers
**www.plicht.de/ekki/**
*Delphi* code snippets for ICOM CT-17 level control by DF4OR. Lots of information about controlling radios with PCs in the software and ICOM CI-V sections.

AA6YQ / Ambersoft
**Ambersoft.com/Amateur_Radio/Index.htm**
*Visual Basic* code fragments for programmers controlling radios with PCs

Commander
**www.qsl.net/civ_commander**
Free radio frequency-control program for ICOM, Kenwood and Yaesu. Those developing this program solicit your input

*There is no online developer support from any of these manufacturers.

**Programming Resources**

Mapping site
**www.versamap.com/webdoc03.htm**
CIA public domain map of the world in digital form

Voice of America/Department of Commerce
**elbert.its.bldrdoc.gov/**
*VOCAP*—predicts performance of HF broadcast systems. May no longer be available, contains both *Visual Basic* and *BASIC* code.

Intel
**www.intel.com/software/products/perflib/index.htm**
Intel signal-processing library—free signal processing DLLs, controls and so on. Examples with *Delphi* and *Visual Basic*

CIA—The World Fact Book
**www.cia.gov/cia/publications/factbook/**
Amazing public domain data regarding countries of the world including maps.

CIA World DataBank II
**www.evl.uic.edu/pape/data/Earth/**
Public domain information about countries of the world, up-to-date map information. Dave Pape has digital map data at different resolutions and bitmaps for download.

QRZ.com
**www.qrz.com**

Specifications for format of Keplerian elements in file kep_fmt.txt. Explanation of satellite orbital elements available from NASA, AMSAT and so on.

***Delphi* Resources**

Yahoo Discussion Forum
**Yahoo.com**
*Delphi* programming forum, general code examples and answers to programming questions

Borland
**Borland.com**
General code examples and downloads for *Delphi* and *C++ Builder*. The main *Delphi* site. A free download of *Delphi 6 Standard* might still be available on the site. It requires registration. There are restrictions on the use of programs developed with the free download.

*Delphi* Super Page
**delphi.icm.edu.pl/**
Freeware, shareware controls and code examples.

Torry's *Delphi* Pages
**www.torry.net/**
Freeware, shareware controls and code examples.

Efg's Reference Library *Delphi*
**www.efg2.com/lab/library/index.html**
*Extensive source for programming code for graphics, math routines and such. Numerous links and code for various programming platforms.*

**General *Visual Basic* Resources**

*Visual Basic* World
**www.vb-world.net**
Source code shareware, freeware, tutorials

**General *QBasic, Quick Basic PowerBasic* Resources**

ABC Basic
**www.allbasiccode.com**
Over 2000 pieces of free source code—lots of example code and complete nonham projects.

Qbasic.com
**www.qbasic.com**
300 program examples (not associated with Microsoft)

**Table 3—Suggested Reading**

***Beginners***

G. Perry, *The Complete Idiot's Guide to* Qbasic, (Indianapolis, Indiana: Alpha Books, 1994).
N. Rubenking, Delphi *Programming for Dummies,* (New York: IDG Books Worldwide, 1995).
D. Stivison, *Introduction to Turbo Pascal,* (Alameda, California: Sybex, 1987).
B. Watson, Delphi *By Example,* (New York: Que, 1995).
K. Reisdorph, *Sams' Teach Yourself Borland C++ Builder 3 in 21 Days,* (Indianapolis, Indiana: Sams Publishing, 1998).

***Advanced***

M. Cantu', *Mastering* Delphi *4,* (Alameda, California: Sybex, 1998).
S. Teixeira and X. Pacheco, Delphi *5 Developer's Guide,* (Indianapolis, Indiana: Sams Publishing, 2000).
M. Waite and others, *Microsoft* QuickBasic *Bible,* (Redmond, Washington: Microsoft Press, 1990).
C. Calvert, *Charlie Calvert's Borland C++ Builder,* (Indianapolis, Indiana: Sams Publishing, 1997).

gramming tools (*Visual Basic 3*, *Delphi 1*, *Realizer*) execute more slowly than those designed with 32-bit tools (*Visual Basic 4-6*, *Delphi 2-6* and such).

I have learned to program *C++* and *Visual C++*, but I find that it takes a special kind of person to become proficient with those languages. Generally, development time is longer with these tools than with *Visual Basic* or *Delphi* but project execution, if you followed best programming practices, is faster, especially execution of mathematical routines. *Delphi 5* and *Delphi 6*, in many programming situations are almost as fast as *C++* implementations versus the much slower *Visual Basic*.

My personal preference is the *Delphi 5 Professional* that I acquired new about a year ago. My learning curve was very steep. Programming in DOS is still practical and enjoyable, but for the time employed and for a few more dollars to buy the programming software, you should be programming Windows. If you do not like *Delphi*, try *Visual Basic,* and if you are a perfectionist try *Visual C++,* but be prepared to work hard.

## Start to Program

"Just do it!" exhorts a sign in the office of one of my friends. Depending on what tool one is using, this may work for most of you. For the rest, learning to program is not necessarily any more difficult than getting your amateur license provided you arm yourself with some of the programmers' study guides. If you can find them, several beginning books might be useful. "Suggested Reading" contains some you might wish to consider.

Some of these books are quite expensive, but it is possible to find earlier, used or new editions in secondhand bookstores. Many of these include a CD-ROM with coding examples. Your local library may be an excellent resource. If the book covers a version not too far removed from the language version you are using, a book purchase might be worthwhile.

While programming today is less difficult than it once was, most programs contain hundreds of lines of code that someone else spent considerable time to develop. While it does take some skill to program, modifying existing programs can be considerably easier than developing a new program. That is when the material available at Websites like those made available by N1MM, WA0TTN, AE4JY and others become important to beginning and accomplished software programmers.

Other sites provide full source code (see Table 2). If you are lucky, the code will be usable directly with your programming software. If it is not compatible with the language you are using, aptitude in several programming languages may permit you to translate what you can find into the form you require. Sites on the Internet may

help you with translations.

If you find a program that interests you, look at the source code and compile it using your programming software. If the program runs, you are in luck. If not, the source code might have been programmed in an earlier or later version of the software you have. Do not give up hope. Most probably, only small changes are needed to get the source code to compile with your compiler. Some compilers, like *Delphi*, give you hints as to what the difficulty might be.

I am a great believer in learning by example. There are lots of ham radio source code examples out there—and even more general examples. Hams do a lot to help one another. If you decide to write your own programs, you will find that the programming fraternity is similar; however, many programmers rightfully consider their code proprietary. When you use someone else's code, give credit where it is due. If it is copyrighted, follow the terms of the license agreement. If you have questions about how to do something program-

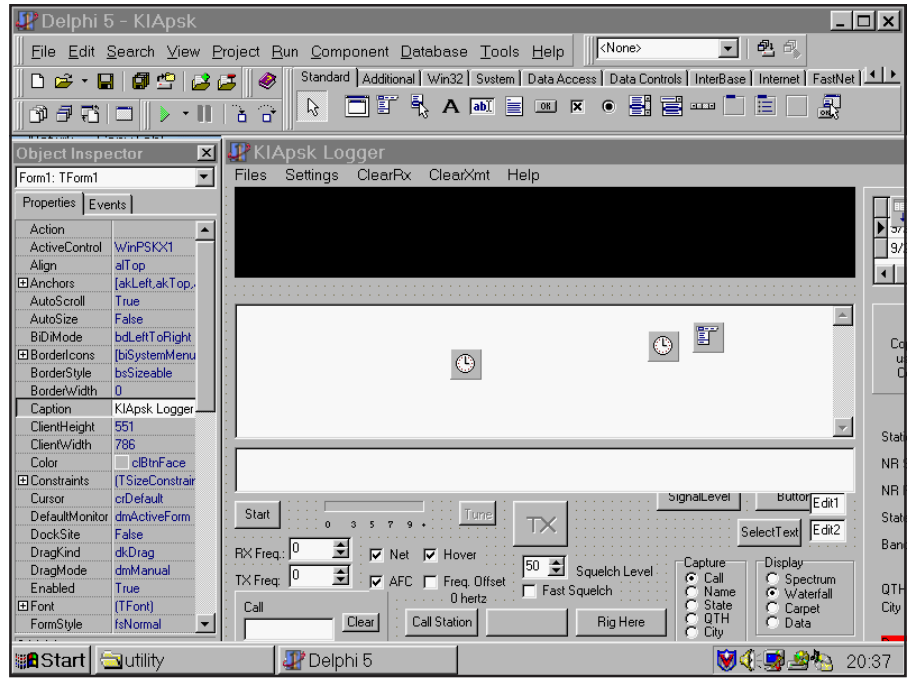



**Fig 7—*Delphi 5* design screen showing *KIApsk Logger* development in progress. *Delphi* uses visual objects and controls to simplify programming tasks.**

**Table 4—Code Sources**

### *Radio Propagation*

R. Rose, K6GKU, "MINIMUF: A Simplified MUF-Prediction Program for Microcomputers," *QST*, Dec 1982, pp 36-38, 43. This is a very simple HF propagation prediction program.

J. Priedigkeit, W6ZGN, "A Simple Computer Model for VHF/UHF Propagation," *QST*, July 1983, pp 32-33.

T. Frenaye, K1KI, "The KI Edge," *QST*, Jun 1984, pp 54-56. Discusses the gray-line and provides mathematical information for calculation.

K. Arneberg, LA9YF, "Beregning av soloppgang og solnedgand," *Amatorradio*, Mars 1985, pp 75-76, *BASIC* code for sunrise/sunset times (Norwegian Radio Relay League Journal).

Source code for *VOACAP*, the propagation program developed by NTIA/ITS for the Voice of America.

### *Satellites*

T. Clark, W3IWI, "*BASIC* Orbits," *Orbit*, Mar/Apr 1981, pp 10-11, 19-20, 29. Article contains a *BASIC* listing of a satellite-tracking program.

I. Jefferson, G4IXT, "Tracking Satellites with a Microcomputer," *Wireless World*, Apr 1983. Describes the mathematics of satellite tracking. A *BASIC* listing was available from the magazine.

### *Finding Directions*

Svein, LA6PV, "Avstandsberegning," *Amatorradio*, Mars 1985, pp 93-94. The article describes distances, bearings and grid-square calculation program in *BASIC* (Norwegian Radio Relay League Journal).

### *Logging Programs*

J. Hess, W9KTP, "The Would-be Contest Killer," *QST*, Oct 1983, pp 20-22. Article contains a *BASIC* program listing.

S. Horzepa, WA1LOU, "*BASIC* Duping," *QST*, Dec 1982, p 74. Article contains a *BASIC* "hash table" listing.

R. Cheek, W3VT, "LOGPROG—A DXer's Log in *BASIC*," *QST*, Sep 1984, pp 24-29. Article contains a *BASIC* logging program listing.

G. Allison, K5IJ, "The Super Duper," *QST*, Pt 1, Sep 1985, pp 27-30; Nov 1985, pp 44-50. Learn *BASIC* language programming techniques while designing a contest duping and logging program.

R. Keller, K3PCS, "Super-Double Bubble," (Technical Correspondence) *QST*, Dec 1985, pp 52-53. Article contains another bubble sort for the Super Duper.

P. Wisiolek, K1TKL, "Super-Double Bubble," (Technical Correspondence) *QST*, Dec 1985, p 53. Still another bubble sort for the Super Duper.

G. Schulz, WB9NDM, "Super Duper POOP," (Technical Correspondence) *QST*, Mar 1986, p 46. Article contains suggested changes to the Super Duper listing.

J. Scott, KA8FSM, "Super Duper Printer," (Technical Correspondence) *QST*, Apr 1986, pp 41-42. Article contains a *BASIC* listing to print log results.

T. Karnauskas, N9BWY, "Better Sort," (Technical Correspondence) *QST*, Aug 1986, p 40. Article contains a better sort routine.

"The Cabrillo File Format," *QST*, Nov 1999, p 102. Article shows the file format ARRL requires for submission of digital contest logs.

### *CW Sending*

D. Whipkey, N3DN, "A Keyboard Keyer and Code-Practice System," *QST*, Jan 1984, pp 13-16. Article contains a *BASIC* program listing and machine-language routine for a Commodore VIC 20.

R. Schetgen, KU7G, "C 64 Keyboard," *QST*, May 1984, p 45. Article adapts "A Keyboard Keyer and Code-Practice System" to work on a Commodore C64 computer.

### *Technical*

C. MacKeand, WA3ZKZ, "The Smith Chart in *BASIC*," *QST*, Nov 1984, pp 28-31. Article contains a listing in *BASIC*.

### *Radio Control and Information Display*

B. Wood, W0DZ, "The Return of the Slide Rule Dial," *QST*, Feb 2002, pp 33-35. Article contains a code snippet from the *Visual Basic* project. Source code and the executable are free by e-mail (**w0dz@arrl.net**).

AA6YQ Web site, *Visual Basic* code fragments for ICOM radio control (**www.ambersoft.com/Amateur_Radio/index.htm**).

The Plickt Brothers Website, Ekkis, DF4OR has *Delphi* code fragments for ICOM and others. (**www.plicht.de/ekki/**).

### *Amateur Radio Education*

The ARRL/VEC exam question pool is available from the ARRL (**www.arrl.org**) as a text or PDF file and is useful although not a program listing.

matically, someone might have an answer on the Internet forum devoted to your particular programming language.

*BASIC* code listings for various Amateur Radio activities are described in the literature and sources (oldest to youngest) described in "Code Sources." These are useful if you are programming in *BASIC* or *Visual Basic* because they can be used in a more modern program with some rewriting. The algorithms can also be adapted for other languages.

Program listings are available in books by the *ARRL, CQ* magazine and so forth or from various sites on the Internet. Sources of material suitable for designing computer programs include *QST*, *The ARRL Handbook* and various books and magazines.

Looking at some of the material discussed above will give you a feel for what a fully coded project will look like. If you have the right language software, you probably will want to load some of the source code you find (after reading instructions about how to do this with your particular coding software). If this has not frightened you away, and it should not, try the program out.

What language should you start to program in? It depends on your budget. The latest professional editions of Windows software start at about $400 but that is because they contain tools to program Internet sites, servers, networks and all kinds of things in addition to desktop applications. What you do need is something that will let you program desktop applications. Both Borland and Microsoft have beginners' versions of their programming software available for less than $100 at present. I would recommend investing in one of these rather than working with the older software. You will get the latest controls and they can be augmented from various freeware and shareware components available on the Internet. Older software is fine, I still use it, depending on what I want to do.

### DLLs and ActiveX Controls

The programming languages developed for the Windows operating system can operate with various mini-programs or program libraries. These tools can work with most of the individual programming languages although they themselves might have been developed with a different language. Such tools may be DLLs or ActiveX controls. A DLL is a function library that works in conjunction with programs that link to it. Several free

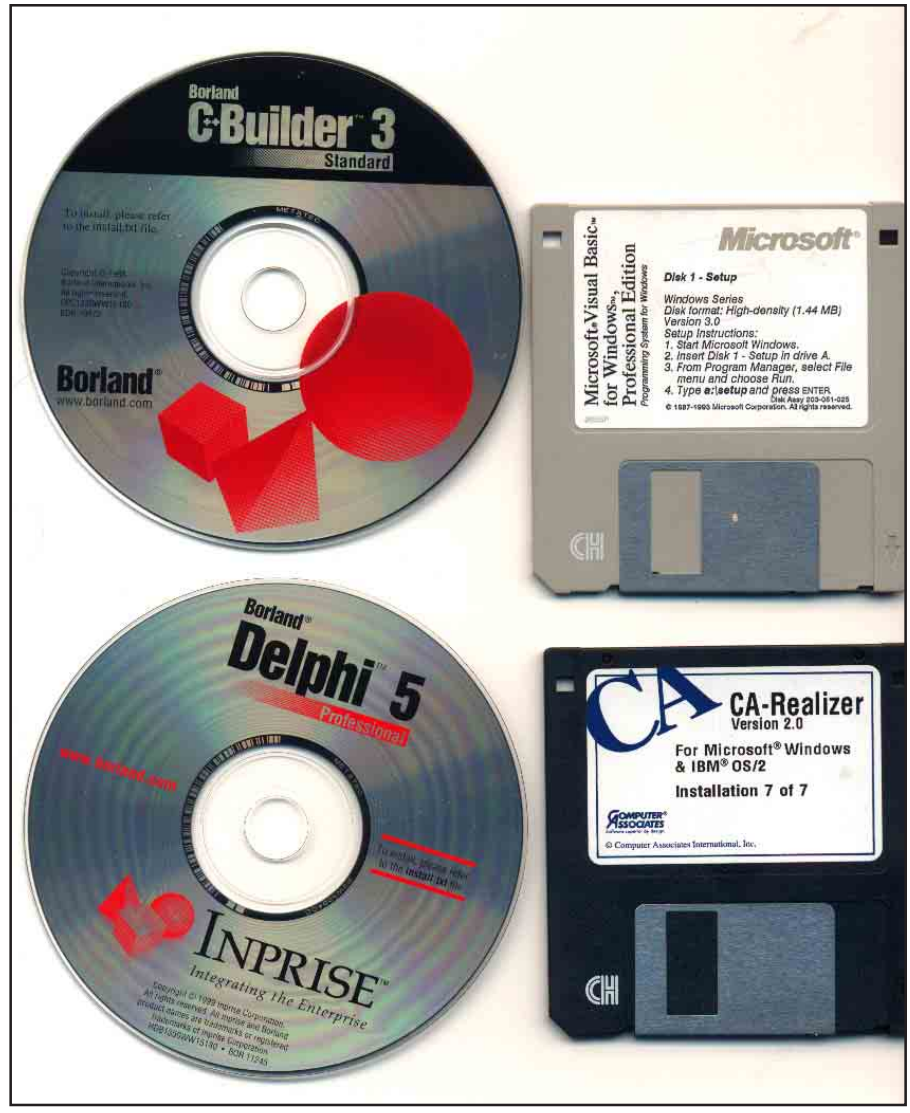



**Fig 8—Programming-language media for *Delphi 5*, *C++ Builder*, *QuickBasic 4.5* and *Realizer*.**

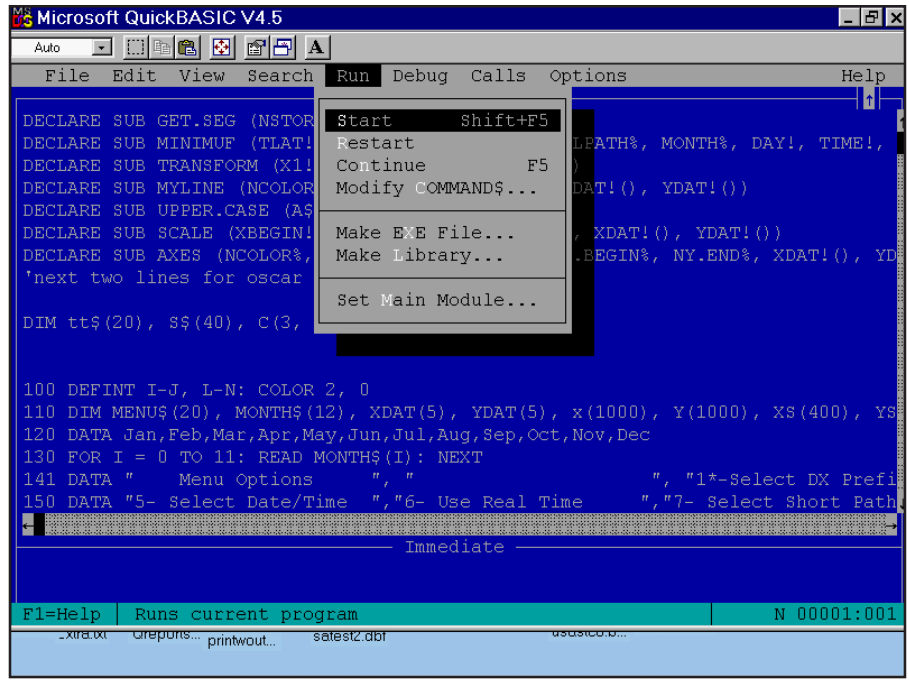



**Fig 9—(right) The *QuickBasic 4.5* development screen showing one of its possible programming menus.**

DLLs related to ham radio topics are available to link to your project including AE4JY's *PSK Core* DLL and JE3HHT's RTTY DLL. The only free ActiveX control I know of for amateur purposes is WA0TTN's *WinPSK*. DLLs can be generated by *Delphi*, and most versions of *C++*. *Visual Basic 3* does not create DLLs but can make use of those generated by other programs. *Visual Basic 4* and above can be used to make DLLs.

There are two types of Windows executable files, programs and DLLs. When you write a Windows application, you typically generate a program file that is an independent program. The executable programs (those with the familiar form *PROGRAM*.EXE) may use calls to functions stored in DLLs. Dynamic link libraries are program modules that contain code, data or resources that can be shared. They allow programs to be modular and simplify updating applications. They are language-independent, so a DLL can be used by *C++*, *Visual Basic*, *Delphi* or any other language that supports DLLs.

ActiveX is a Microsoft technology that is an extension of the older OCX technology. It provides controls that can be used in a Windows visual-programming environment to provide certain functionality. ActiveX controls are add-ons to your programming environment and should be cross platform compatible with most 32-bit systems.

### What Should You Program?

If you have gotten this far in the article, you may have in mind a particular project or perhaps not. It is possible to write a software program describing just about any conceivable process or mathematical relationship. Many very good ham-related programs are available already. You may want to duplicate an expensive piece of commercial software to save costs or just for the fun and satisfaction of doing it. I hope that you have something new in mind or a new way to present familiar material.

### Giving Your Software the "Commercial" Look

On-board help and "about" boxes or splash screens containing information about the programmer/copyright holder can make your program easier for the user to understand and help protect your rights as a developer by providing a place for a copyright notice or licensing terms, even if you choose to give your software away.

Tools necessary to develop Microsoft compatible help files are usually included with the Windows programming language. The Microsoft Windows Help File Compiler was provided with the copies of the development programs I have as part of the development product. The compiler, in its various versions, is a script-like language similar to HTML, the language used to program many Web sites.

"About" boxes are drop-down or pop-up information boxes. Splash screens display a picture or other data on your screen when your program starts (Fig 12). Both of these methods can provide information about your program or its author in a pleasing way.

### Conclusion

Programming for DOS or for Win-

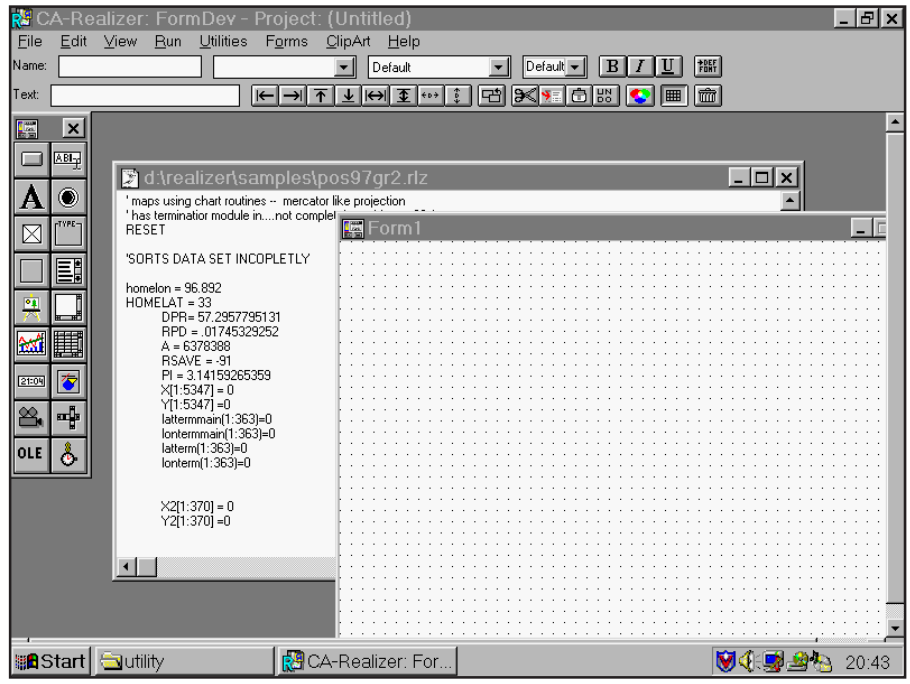



**Fig 10—The *Realizer* for Windows development screen was one of the first to use visual objects.**

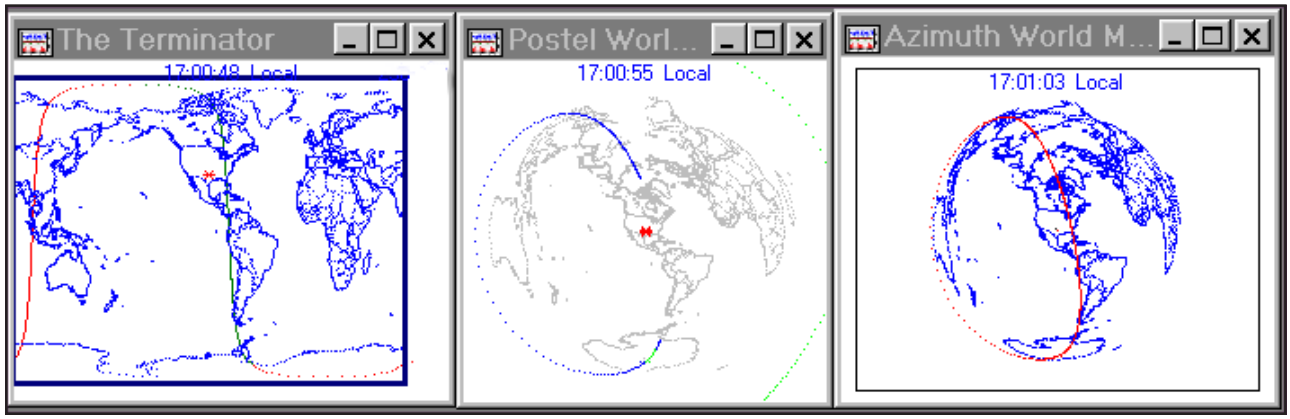



**Fig 11—Pseudo-Mercator, Postel and azimuth-equal area map projections of the world using the CIA World Map public domain database coordinates described in Table 2. A gray-line track of the boundary between areas of sunrise and sunset is also shown.**

## Additional Programming Resources

Readers may want to visit the sites of the Free Software Foundation (**www.fsf.org**) and Sun microsystems (**www.sun.com**). Sun gives away *Java* tools. FSF does the same with many development tools. The only tool I use of these is the *C* compiler from FSF (*Gnu C*); it is a high-quality piece of code. I have downloaded *Java* from Sun, but never got a chance to use it (the book has been gathering dust for three years). It is similar to *Delphi* and *Visual Basic* in its abilities to create Windows programs. There is a lot more than just *C* and *C++* available from FSF. They have an editor (*Emacs*), a free version of just about every *Unix* tool ever written (available for *Windows*, *Linux*, *Unix* etc) and numerous things that are unique to FSF. I just cruised to their Web site and found projects for *C*, *Pascal*, Software Defined Radios, *Java* and a ham-radio section under "Hobbies."

The Free Software Foundation has what they call a "copyleft." They basically detest copyrights, but in order to protect their work, assert an actual copyright that let's you use the software for any purpose (including commercial purposes), but if you improve it, you *must* give your improvements away to the whole world. This is a good model for getting things moving in the ham community. *Linux* is now a viable product because of this model.—*Ray Mack, WD5IFS, QEX Contributing Editor;* **wd5ifs@arrl.org**

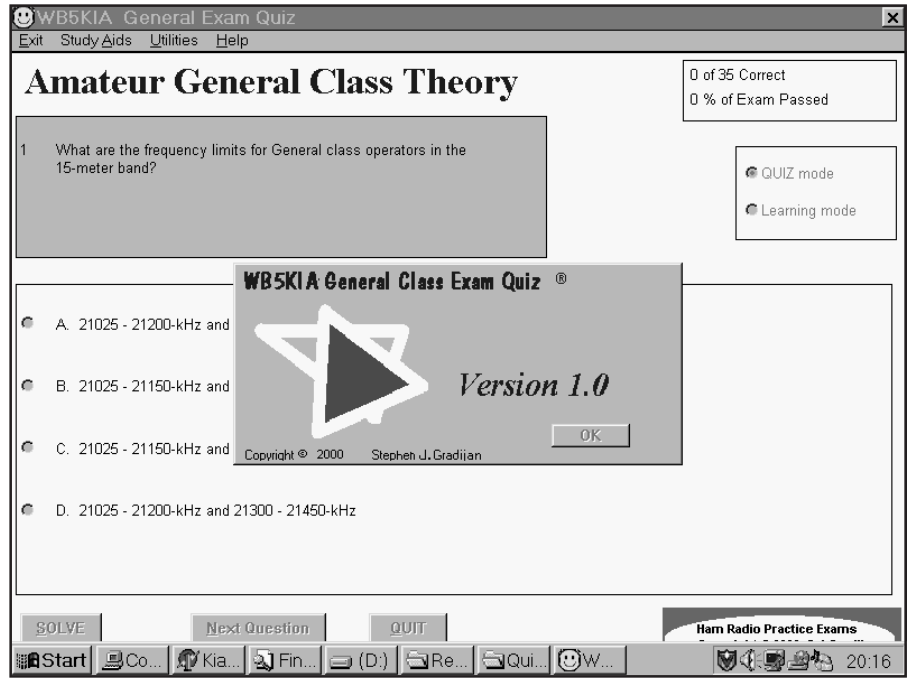



**Fig 12—An about screen from the General license quiz generator and study guide programmed with *Realizer*.**

dows certainly is not for everyone, but with the right tools, it can be an enjoyable and satisfying experience. If all you have available are DOS tools, remember it is possible to run most DOS programs in a window in the various versions of Windows. If you do not program, but have some good ideas for a program or for its improvement, share them. If you do program for fun, consider making your project and code available for others to use in their projects.

*A ham since 1963, Steve Gradijan, WB5KIA, is a geological consultant in the Dallas, Texas area. He holds an Extra class license. Computer programming has been his second hobby since the late 70s. He has previously been licensed as WA8KBK and LA0DY. His wife Chris is WD5EML (ex LA0DZ) and 15-year-old son Francis is KD5HTB.*

# *Understanding Switching Power Supplies, Part 1*

*Many find switching supplies mystifying. Let's begin a tour through that strange land.*

By Ray Mack, WD5IFS

This is the first of a series of articles that will explain how switch-mode power supplies work in detail. They will give enough information to design several different types of switching power supplies.

There are two broad classes of power supplies: linear and switching. Linear supplies use time-continuous control of the output. Switching supplies are time-sampled systems that use rectangular samples to control the output. There are three types of switching power supplies. *Charge pumps* manipulate the charge on a capacitor to change voltage. They can be used to increase voltage or invert the voltage. *Buck* and *boost* switching supplies use an inductor to provide the voltage conversion and regulation to either increase the voltage (a boost regulator), to reduce the voltage (a buck regulator) or invert the voltage (voltage inversion boost regulator). These articles focus on the buck and boost types of supply.

17060 Conway Springs Ct
Austin, TX 78717-2989
wd5ifs@arrl.org

### Inductor Basics

The important features of an inductor are that the current through it cannot change instantaneously and the voltage across the inductor is a result of the change in inductor current. This means that the voltage across the inductor will be positive (with respect to the polarity dot) when you charge the inductor and negative when the inductor is discharging. This allows the voltage to change essentially instantaneously.

If you start current flowing through an inductor by closing a switch (Fig 1A), the current will rise exponentially. Immediately after the switch is opened (Fig 1B), the current through the inductor continues to flow in the same direction at the same value. The current decreases exponentially while the current flows through R2. The property that allows the voltage across an inductor to quickly change from positive to negative is the main property exploited in switching power supplies.

### A Buck Regulator

The buck regulator is a variation on the classic choke-input supply. The output of a choke input supply is always the average value of the input waveform. In the case of a rectangular input waveform, the output voltage is simply the input voltage times the duty cycle of the input (Eq 1). It is

possible for the input duty cycle to be any value between zero and 100%. The defining characteristic of a buck regulator is that the inductor provides current to the load during charging and discharging of the inductor.

$$V_{out} = V_{in} \times Duty\ Cycle \qquad \text{(Eq 1)}$$

Let's analyze the operation of a representative buck regulator. Our example regulator takes a nominal 12.6 V from a battery for a transceiver and creates 5.0 V for the digital logic. We'll assume that the logic draws 200 mA, which is the equivalent of a 25 Ω resistor as a load. The typical switching supply today uses a sample rate of 100 kHz or more. This allows us to use a very small value of inductor and a small value of output capacitor. The sample frequency of 100 kHz means that we will sample every 10 microseconds.

Fig 2 is an idealized representation of our regulator. The regulator switch, S1, is an electronically controlled mechanical switch that opens and closes at the 100-kHz sample frequency. We start our analysis with the output voltage of zero and zero current flowing in the inductor (see Fig 3). During the first sample period, the majority of the inductor current flows into the output capacitor. The output voltage has not risen to 5.0 V yet, so the switch does not open (100% duty cycle). The same is true during the second sample period.

By the fifth sample period, most of the current of the inductor is flowing into the load resistor and the voltage has risen to the desired 5.0 V so the switch opens. When the switch opens, the current through the inductor continues to flow in the same direction into the load resistor. D1 is a commutating diode that allows the inductor current to continue to flow without significantly changing the effective resistance across the inductor. Without the diode, the energy stored in the inductor would be dissipated very quickly across the very high equivalent resistance of the switch. In other words, there would be an arc across the switch contacts.

The regulator has now reached the point where it is in regulation. When the sample period begins, the switch closes and there is 7.6 V (12.6 – 5.0) across the inductor. The inductor current increases until the switch opens. When the switch opens the voltage across the inductor decreases to the 5.0 V across the capacitor plus the 0.7 V drop across the diode. This lower voltage causes the inductor current to decrease at a lower rate than the charging current increase. The average current into the load resistance is easily calculated as the area under the triangular waveform. In our case, this average is 200 mA. Our assumption here is that the output RC time constant of the load resistance and filter capacitor is significantly longer than the 10 µs sample time.

Now we close S2 to model what happens with a large jump in load current. At first, the current in the inductor remains at the level of 200 mA. The additional current required by the second 25-Ω load resistor starts to reduce the voltage on the output capacitor. The regulator responds by keeping the switch closed through several sample periods so that the inductor charges up to 400 mA. You will notice that the ripple current is the same for 400 mA of output current as when delivering 200 mA (remember that this is a regulator with ideal components). We will look more closely at the non-ideal behavior of real components in real regulators in the next installment.

### Step-Up Boost Regulator

Anyone who has ever worked on the high-voltage section of a TV or the ignition coil of an automobile is familiar with the mechanism that is the heart of a boost regulator. In general,

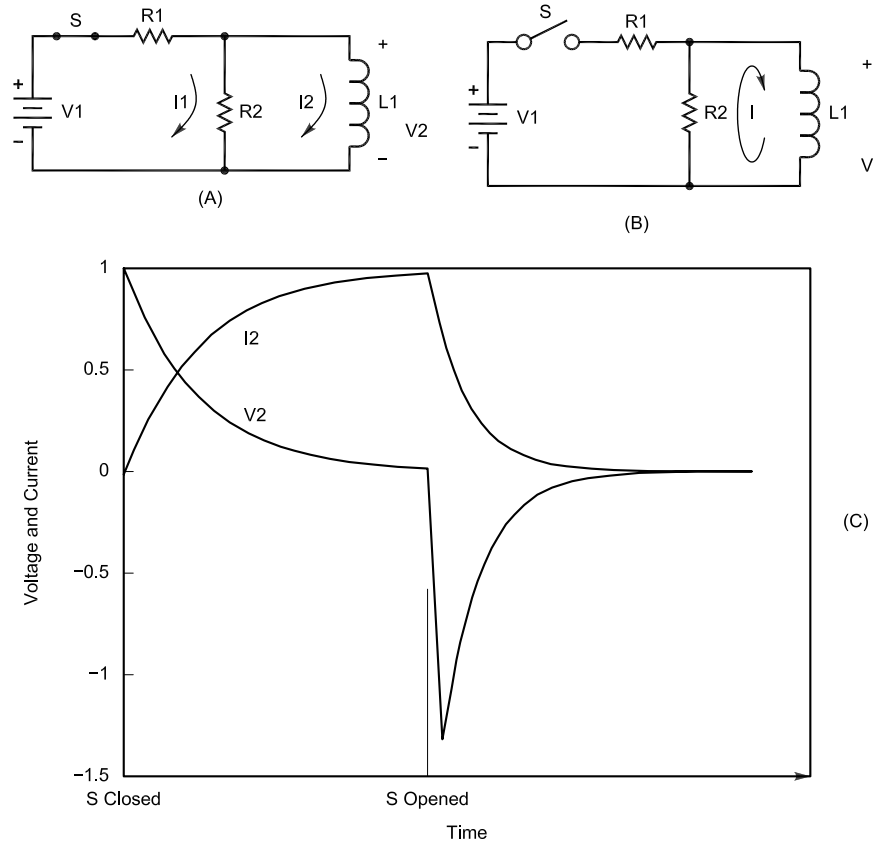



**Fig 1—Behavior of a switched ideal inductor. When S is first closed (A), currents flow through both the load (I1) and inductor (I2). Immediately after S is opened, current from the inductor continues to flow through the load in the same direction at the same value (B). (C) shows how I2 and V2 behave during switching.**

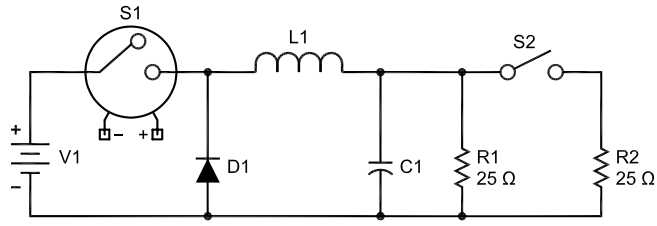



**Fig 2—An idealized representation of our regulator.**

a boost regulator operates by charging an inductor with current from a low voltage, low resistance source for a long period and then discharging the inductor over a much shorter period of time through a high resistance. The defining characteristic of a boost regulator is that the inductor provides current to the load only during discharging of the inductor.

There are two topologies for a boost regulator. One adds the voltage across the inductor to the input voltage to provide a boosted output voltage (Fig 4). If the switch is moved between the supply and the inductor (Fig 5), the polarity of the output voltage is reversed. Again, the commutating diode allows the current in the inductor to continue flowing when the switch opens. In this case, it also prevents current from flowing in the load while the inductor is charging.

Fig 6 illustrates the operation of a voltage-boost regulator. Unlike the buck regulator, a boost regulator must have the duty cycle of the switch restricted to some value less than 100%. This is so because current is only delivered to the load when the switch opens. For our idealized example, we will limit the duty cycle to 95%. Again, we choose a sample frequency of 100 kHz. Before the control circuit can start operating, the commutating diode allows current to flow into the capacitor and the resistor charging it to the supply voltage (12.6 V). During the first sample period, the current in the inductor starts at 52 mA and rises at a rate controlled by the voltage across the inductor. Since the output voltage is 12.6, the control circuitry keeps the switch closed for the full 9.5 µs. The switch, S1, opens and current continues to flow in the same direction in the inductor. Now, however, the voltage across the inductor changes polarity and adds to the voltage of the power supply to charge the capacitor and deliver current to R1.

As the capacitor charges, the inductor current required tends to grow very large until the circuit is in control. For several reasons, it is necessary to limit the current in the inductor (1 A in our example) so that the control circuit will eventually be able to control the output voltage. In our example, the circuit must also keep the switch open allowing the inductor current to return to zero in order to compensate for the overshoot of the output voltage. Once the output voltage has stabilized at 24.0 V, the control circuit implements Eq 2.

$$V_{out} = \frac{V_{in}}{1 - Duty\ Cycle} \qquad \text{(Eq 2)}$$

Again, we can look at what happens when a jump in load current occurs. We close S2, which places an additional 240 Ω of load on the circuit. The capacitor immediately starts supplying the additional current and its voltage starts to decrease. The control circuit responds by increasing the charge time of the inductor to bring the system back into stability. Again, this circuit is underdamped, so the voltage rises out of control and drifts back down to where the control circuit can use Eq 2 to control the output voltage.

### Voltage-Inversion Boost Regulator

The other variation of the boost regulator is the voltage inverter. The operation of the circuit is similar to the step-up boost circuit. The switch and inductor trade places, and the diode is reversed from the boost configuration. We will again start our analysis with initial power on (see Fig 7). When the switch closes, current flows for the full 9.5 µs, charging the inductor. When the switch opens, the voltage across the inductor again reverses. This creates a negative output voltage across the capacitor and load. The control circuit continues to charge the inductor until the inductor current is sufficient to supply all of the load current during each sample period. The analysis of a change in load current is equivalent to that for the step-up boost

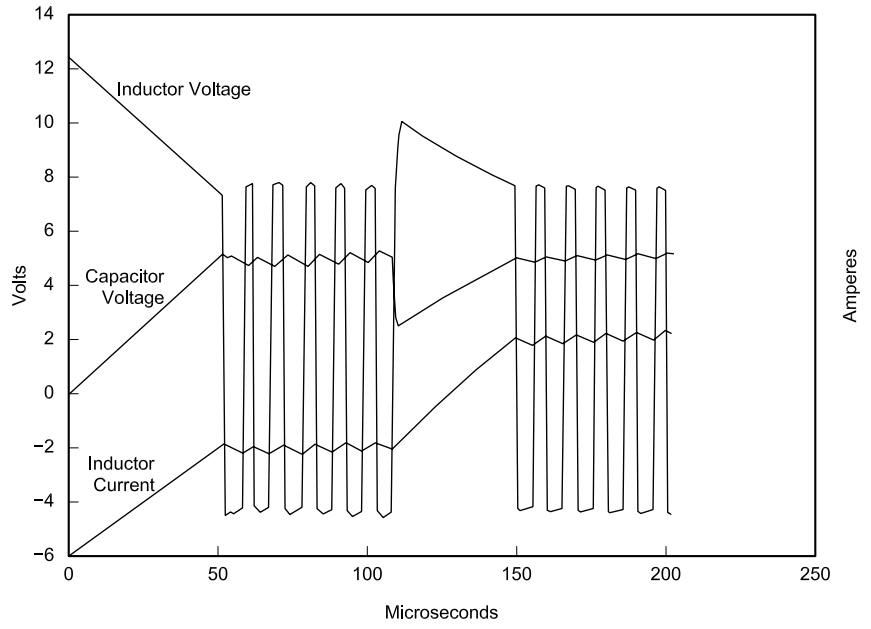



**Fig 3—An operational analysis of the regulator in Fig 2. See the text for details.**

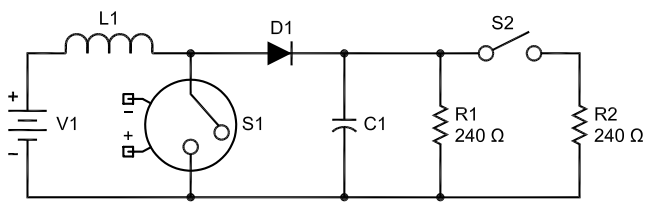



**Fig 4—This boost-regulator topology adds the voltage across the inductor to the input voltage for a boosted output.**

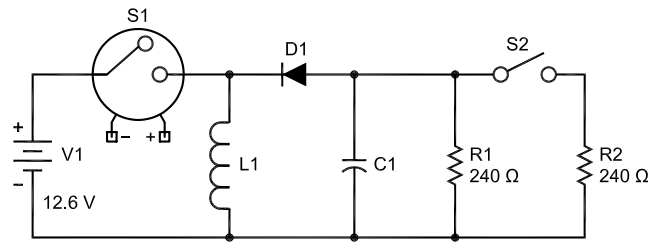



**Fig 5—This boost-regulator topology moves the switch between the supply and the inductor to reverse the polarity of the output.**

regulator. Notice again that the inductor current is limited to allow the circuit to quickly come into control.

### Basic Transformer Topologies

If you read textbooks on switching supplies, you will see transformer-coupled supplies described as either *buck* or *boost* topology.

Buck converters are also called *forward converters* because the transformer operates as a true transformer. This means that while current is flowing in the primary, there is also current flowing in the secondary. Leakage inductance is an undesired side effect of the transformer operation.

A boost converter is also called a *flyback converter*, and the leakage inductance of the transformer is used to store the energy that is eventually delivered to the load. In flyback converters, the secondary current flows while the switch is off. It gets its name from the flyback circuit of a television horizontal-deflection system. In effect, the transformer of a flyback converter is really two inductors that share a common magnetic core rather than a true transformer.

It is very easy to get confused by the topologies when a transformer is involved. A buck regulator can only regulate a voltage to a lower value. A boost regulator can only create a higher voltage or a negative voltage. When a transformer is involved, both topologies can perform one or all of the voltage-conversion functions.

All off-line (hooked to the power mains) power supplies use either forward or flyback designs to isolate the load from the power lines using the inherent isolation of the primary and secondary windings of the transformer.

### The Flyback Converter

Operation of the flyback converter is very similar to the operation of the boost converter. We vary the energy delivered to the load by varying the

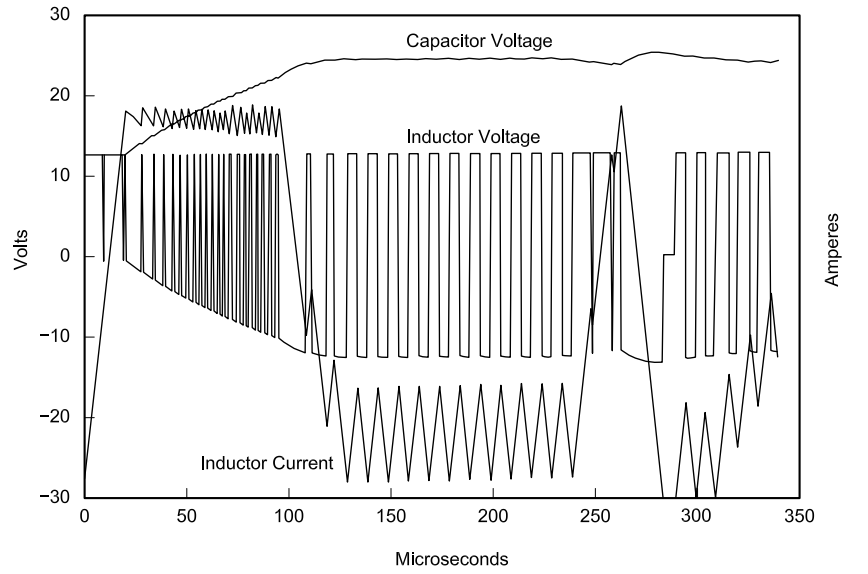



**Fig 6—An operational analysis of a voltage-boost regulator. See the text for details.**

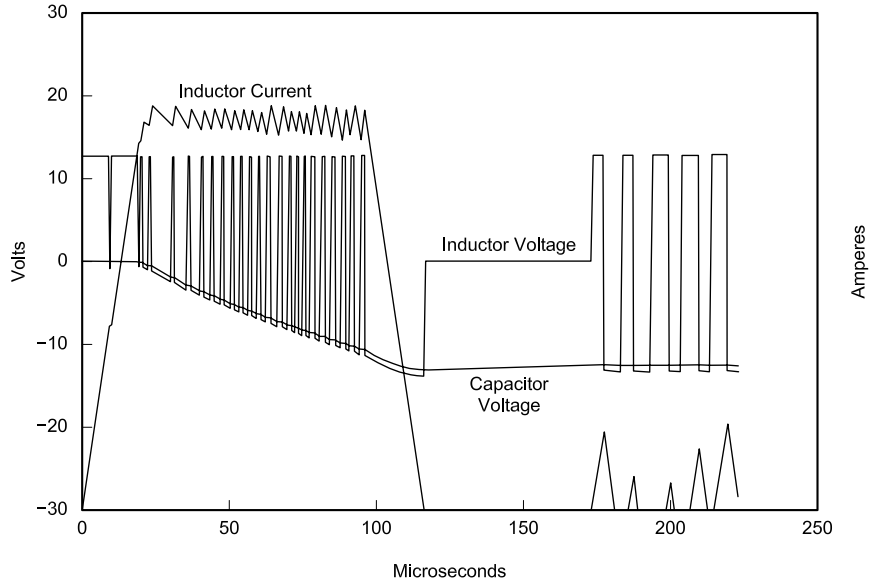



**Fig 7—An operational analysis of a voltage-boost inverter. See the text for details.**

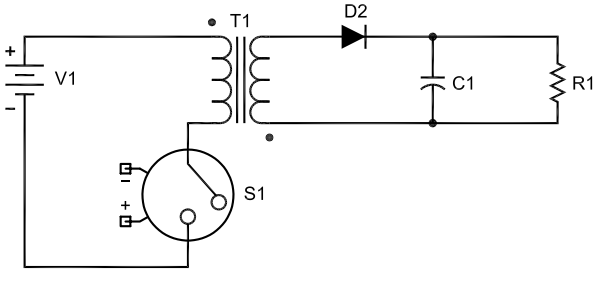



**Fig 8—A model of a flyback converter.**

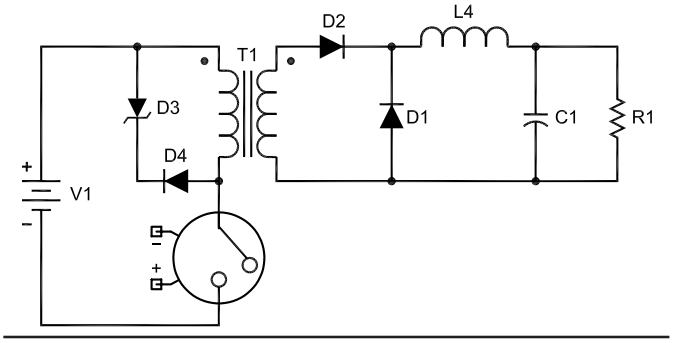



**Fig 9—A model of a single-switch forward converter.**

energy stored in the inductor. Since the energy is stored in the leakage inductance of the transformer, we intentionally control the size of this inductance.

Fig 8 shows a model of the flyback converter. Notice that the phasing is opposite to normal phasing of a transformer. The primary-side inductor is charged with energy while the switch is closed, and the energy in the core is transferred from the secondary-side inductor to the capacitor and load after the switch opens. This operation requires that the duty cycle be limited to a value less than 100 % as in a boost converter. The voltage across the primary inductor reverses (which forward biases the diode in the secondary) when the switch opens, and secondary current flows according to the transformer equation $N1/N2 = I2/I1$. The current flowing in the load circuit controls the voltage across the transformer windings. The voltage across the primary reverses polarity during energy delivery and this adds to the voltage of the input supply. The voltage across the switch is typically twice the input voltage.

### The Single-Switch Forward Converter

Fig 9 shows a single-switch forward converter. Notice that a major difference from the flyback converter is the phasing of the transformer windings. This converter delivers energy to the filter during the time the switch is closed. We use the almost rectangular shape of the voltage across the primary and secondary of the transformer to vary the amount of energy delivered to the filter and load circuit. The output voltage is controlled by the averaging effect of the low-pass filter. We must provide a controlled path for the current in the leakage inductance to flow once the switch opens, since the current cannot immediately return to zero. This path is provided by D3 and D4 in Fig 9. There are various other ways to control the path of the current discharge. We will look closely at those in the design section. The voltage reversal that discharges the current adds with the supply voltage. The consequence is that the peak voltage across the switch is typically twice the input voltage.

### The Push-Pull Forward Converter

Fig 10 shows a push-pull forward converter. This circuit is identical to a push-pull audio or RF circuit except that the signals are rectangular waves. This circuit uses the core to full advantage because the current flowing in opposite windings creates a net zero dc current flow for the core. This allows the full magnetic capability of the transformer core to be utilized. This circuit has the advantage that the drive circuitry for the switches is relatively simple, since both switches are referenced to the common of the input supply. Each switch must withstand twice the input supply voltage. The disadvantage of this circuit is that the transformer primary requires two balanced windings. The 1969 *ARRL Handbook* portable/emergency chapter shows a circuit utilizing self-excited push-pull forward converters. There are two disadvantages to self-excited converters: ensuring proper starting and reproducing the saturation characteristics of the magnetic circuits from supply to supply.

### The Half-Bridge Forward Converter

Fig 11 shows a half-bridge forward converter. This circuit is essentially the same as a totem-pole audio output stage. The difference being that the waveforms are high-frequency square waves instead of audio signals. This configuration also has the advantage that the current flows in opposite directions during opposite cycles, so the transformer core has a net zero dc current flowing in it. The transformer primary has one-half of the input voltage across it. Each switch need withstand only the input supply voltage. This circuit is capable of higher power levels than the single-

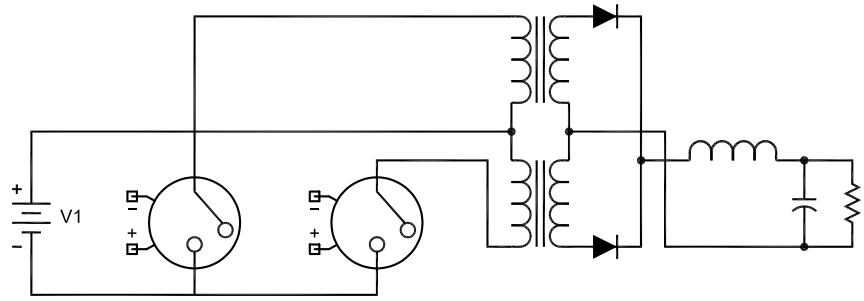



Fig 10—A model of a push-pull forward converter.

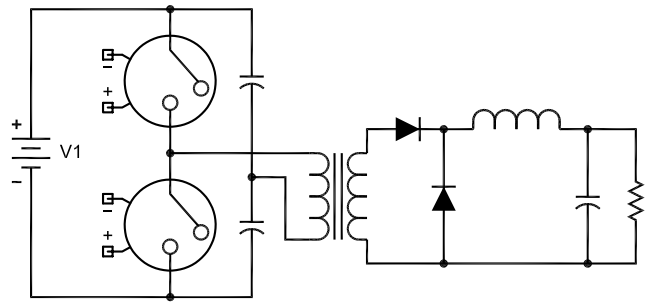



Fig 11—A model of a half-bridge forward converter.

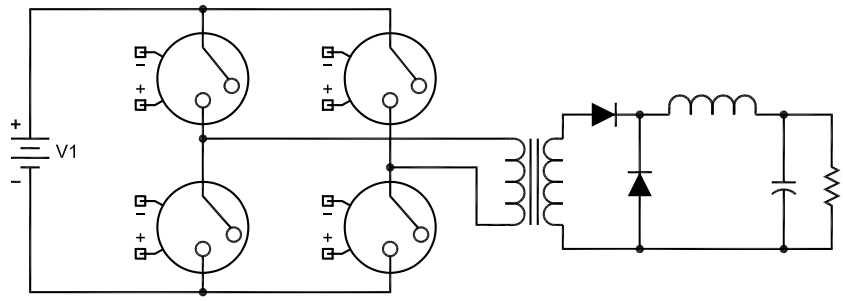



Fig 12—A model of a full-bridge forward converter.

switch converter for equal-size components. The disadvantage is that the drive circuitry for the top switch must be isolated from the input-supply common connection.

### The Full-Bridge Forward Converter

Fig 12 shows a full-bridge forward converter. This circuit replaces the capacitance voltage divider with a second set of switches. Again, there is a net zero dc current in the primary so the full capability of the transformer is used. The full input supply voltage is placed across the transformer primary. Each switch must withstand only the input supply voltage. In this circuit, we need two isolated drive circuits for the top switches. The full-bridge converter gives the highest possible output power for a given size of components at the cost of four switches and two isolated drive circuits.

### *Next Installment*

The next installment will cover the characteristics of real magnetic components and guidelines for how to design a transformer or filter inductor.


### Acknowledgements and References

I thank the folks at Linear Technology, who made a special version of their *SwitcherCAD* program to assist in the drawing of the schematics in this article. *SwitcherCAD* is available free on the Linear Technology Web site.


You can find short application notes on switching power-supply design that complements the information in these articles in the application note sections of the Web sites for Linear Technology (**www.linear-tech.com**) and Maxim Semiconductors (**www.maxim-ic.com**). □□

# The DX Prowess of HF Receivers

*Are you looking for a good receiver for DX hunting? Here are some distilled performance numbers that might point you in the right direction.*

By Tadeusz Raczek, SP7HT

Comparing the performance of one receiver to another is quite a difficult task. Receiver-performance tests are described in detail in *The ARRL Handbook*, Chapter 26. Shortwave DX hunters and contest participants have requested that testing of receiver front ends be made at conditions representing real on-the-air situations. That is, we should test receivers when extremely weak DX signals from the other end of the world are present at the same time as several strong local signals that are close to that tiny DX signal. In contrast to standards set by Amateur Radio community, equipment manufacturers prefer that their products be evaluated at 50 kHz or even 100 kHz signal spacings, where much more optimistic results can be achieved. Table 1 illustrates IC-765 receiver front-end dynamic-range measurements performed by the ARRL laboratory at various signal spacings.

We can see a big difference between 5- and 50-kHz test results; that is, blocking dynamic range (BDR) and intermodulation-dynamic-range (IMD DR) measurements for widely spaced signals produce much better results than for 5-kHz spacing. That explains why manufacturers are opting for wide-spaced measurements.

Closely spaced tests can inform us much more realistically about a receiver's usefulness for DXing and contesting on our crowded HF Bands. The ARRL laboratory, G3SJX and W8JI have published measurement results for some HF receivers with closely spaced signals. The ARRL laboratory and G3SJX used 20 kHz for wide-spaced signals and 5 kHz for narrow-spaced signals. W8JI has used 10 kHz for wide-spaced signals and 2 kHz for narrow-spaced signals.

### BDR and Two-Tone Third-Order Dynamic-Range Tests

BDR is the difference, in decibels, between the minimum discernable signal (MDS) and an off-channel signal that causes 1dB of gain compression in the receiver. Two-tone third-order dynamic range (IMD DR) is the difference between MDS and the levels of two interfering signals causing IMD products just equal to the MDS.

IMD DR depicts the strong-signal capabilities of a receiver; that is, how it behaves under real-world conditions,

Skrytka Pocztowa 738
25-324 Kielce 25
POLAND
sp7ht@wp.pl

when strong signals are delivered from the antenna to the receiver input. Receiver IMD immunity is determined by the limits of its linear signal-handling capabilities. Those, in turn, are determined by the limiting effects of receiver active circuitry such as the preamplifier, mixer and first IF amplifier. Passive components may also exhibit such limiting effects. For instance, fast RF silicon diodes used for receiver input-filter selection and for receive-transmit switching or preamplifier/attenuator activation often cause additional IMD in some present-day HF transceiver models. Moreover, overload of varactor diodes in automatically tuned preselectors, as well as subminiature, inexpensive inductors and monolithic two-pole first-IF filters placed immediately after the first up-conversion mixer, can have a role in IMD generation and receiver performance degradation.

Table 2 demonstrates 20 and 5-kHz-spacing test results of BDR and IMD DR for some HF receivers tested in the ARRL laboratory. Two columns are added for convenience in analysis. In the fourth column, the decrease in BDR is calculated between the 20 and 5-kHz tests. In the sixth column, the decrease in IMD DR is calculated between 20 and 5-kHz tests. Italic numerals distinguish 5-kHz spacing test results.

Table 3 demonstrates 10 and 2-kHz-spacing test results of BDR and IMD DR for some HF transceivers tested by W8JI. Table 3 includes the same additional columns as in Table 2. In column four, the decrease in BDR is calculated between the 10 and 2-kHz tests. Consequently, in column 6, the decrease in IMD DR is calculated between 10 and 2-kHz tests. Italic numerals distinguish 2-kHz test results.

The 2 and 5-kHz closely spaced receiver tests represent real-world, on-the-air DX hunting (split operation), when many strong signals are very close to a very weak DX station signal barely copied in the noise. Less degradation of BDR and IMD DR values means better receiver performance for strong closely spaced signals. You can see that some receivers perform better and some are not as good as we want them to be.

Considering the decrease in BDR and the lowering of IMD DR between widely and closely spaced tests, I consider the best receivers for split-frequency operation with DX stations to be those of the following HF transceivers:

- Elecraft Model K2
- Ten-Tec Model Omni-VI+
- Heavily modified Drake R-4C

The three best results in Table 2 and one result in Table 3 are distinguished by boldface lettering.

### K2 and OMNI-VI+: Design Concepts and Features

Elecraft and Ten-Tec manufacture the K2 and OMNI-VI+, respectively *[The Omni-VI+ has been discontinued as of 2001 in favor of a superior design–Ed]*. Drake discontinued the manufacture of the R-4C about 20 years ago.

**Table 1—IC-765 Receiver Front-End Dynamic-Range Measurements**

| *Signal Spacing* | ***Blocking DR (dB)*** | | ***IMD DR (dB)*** | |
|---|---|---|---|---|
| *(kHz)* | *IF Shift Off* | *IF Shift On* | *IF Shift Off* | *IF Shift On* |
| 5 | 120 | 91 | 85 | 73 |
| 10 | 130.5 | 105 | 90 | 88 |
| 20 | 151.5 | 139.5 | 97 | 95 |
| 50 | 152 | 152 | 99 | 99 |

**Table 2—20 and 5-kHz-Spacing BDR and IMD DR for some HF receivers tested by the ARRL**

| *Manufacturer* | *Model* | *BDR (dB)* | *BDR Decrease* | *IMD DR (dB)* | *IMD DR Decrease* |
|---|---|---|---|---|---|
| **Elecraft** | **K2** | **133** and ***126*** | **only 7 dB** | **97** and ***88*** | **only 9 dB** |
| ICOM | IC-706MkIIG | 120nl and *86* | 34 dB! | 86 and *74* | 12 dB |
| ICOM | IC-746 | 113 and *88* | 25 dB! | 92 and *78* | 14 dB |
| ICOM | IC-756PRO | 120 and *104* | 16 dB | 88 and ***80*** | **only 8 dB** |
| ICOM | IC-775DSP | 132 and *104* | 28 dB! | 103 and *77* | 26 dB! |
| Kenwood | TS-570S(G) | 119 and *87* | 32 dB! | 97nl and *72* | 25 dB! |
| Kenwood | TS-570D | " | " | " | " |
| Kenwood | TS-2000 | 121nl and *99* | 22 dB! | 92 and *67* | 25 dB! |
| **Ten-Tec** | **OMNI-VI** | 128nl and ***119*** | **only 9 dB** | 100 and ***86*** | 14 dB |
| **Ten-Tec** | **OMNI-VI+** | " | " | " | " |
| Yaesu | FT-847 | 109nl and *82* | 27 dB! | 89 and *73* | 16 dB |
| Yaesu | Mark-V FT-1000MP | 126 and ***106*** | 20 dB! | 98 and *78* | 20 dB! |

**Table 3—10 and 2-kHz-Spacing BDR and IMD DR for some HF transceivers tested by W8JI**

| *Manufacturer* | *Model* | *BDR (dB)* | *BDR Decrease* | *IMD DR (dB)* | *IMD DR Decrease* |
|---|---|---|---|---|---|
| ICOM | IC-751A | 98 and 83.5 | 14.5 dB | 91 and 79 | 12 dB |
| Drake | R-4C (stock 1)* | 109 and 57 | 52 dB! | 82 and 48 | 34 dB! |
| Drake | R-4C (stock 2)† | 116 and 80 | 36 dB! | 86 and 68 | 18 dB |
| **Drake** | **R-4C (heavy mod)††** | **131 and 127** | **only 4 dB** | **119 and 118** | **only 1 dB** |

*Stock 1 has MOSFET second mixer.
†Stock 2 has vacuum-tube second mixer.
††Heavy mod is rebuilt with solid-state doubly balanced high-level mixers and Sherwood 600-Hz roofing filter.

For CW-oriented DX hunters, the R-4C is not an impressive receiver when compared to rec2ent models. But after radical modifications, an upgraded R-4C is a good receiver for weak DX signal CW reception on crowded amateur HF bands, thanks to the low phase noise of the R-4C PTO (permeability tuned oscillator). As shown in the ON4UN questionnaire results in the second edition of *The Antennas and Techniques for Low-Band DXing,* a significant number of responders have reported using the R-4C for DXing on 80 and 160 meters.

The K2 and OMNI-VI+ BDR for 5-kHz spacing between strong signals is (126 – 106) 20 dB and (119 – 106) 13 dB, respectively, greater than that for the third-ranked FT-1000MP Mark-V (106 dB). Accordingly, the two-tone third-order dynamic range (IMD DR) of the K2 and OMNI-VI+ for 5-kHz spacing from two strong signals is, respectively, (88 – 80) 8 dB and (86 – 80) 6 dB better than for the third-ranked IC-756PRO (80 dB). This advantage is especially useful for DX-oriented operators.

Such good receiver front-end parameters prove the design concepts implemented by Elecraft and Ten-Tec in the K2 and OMNI-VI+ models. Both makers have abandoned ideas commonly exploited during last 20 years by most other makers of HF transceivers and returned to proven designs used previously but with modern implementations.

The K2 and OMNI-VI+ use the following crucial design ideas in the receiver front end:

- HF ham-band coverage only, no general coverage capability
- Only single (K2) or double (OMNI-VI+) conversion is used instead of a chain of several mixers commonly used by other makers
- Both models have excluded the first up-conversion IF into the 50 to 90-MHz range with the associated wide bandwidth first-IF roofing filter (with its passband set wide enough for narrow FM transmission and adequate for noise-blanker operation)
- Both models use a relatively low first IF that allows installation of narrow SSB/CW crystal filters with good shape factors to greatly attenuate out-of-band IF signals just at the front of the IF amplifier
- The main IF selectivity of the crystal filters is very close to the receiver front end, which helps substantially to obtain high BDR and good IMD DR even for closely spaced strong signals
- Both models implement ham-band-only preselector filters that substantially suppress strong signals outside of the ham bands and prevent receiver front-end overload and IMD

In designing its K2, the main goal of Elecraft was to construct an HF transceiver devoted only to the ham bands, useful for DX hunting—mainly CW—with SSB as an option. As Table 2 indicates, this has been done successfully.

The K2 HF transceiver implements a single-conversion superhet receiver:

- A doubly balanced diode mixer offers excellent dynamic range. Narrow and ham-band-only double-tuned preselector filters are switched by relays, so the receiver front end offers much better IMD response than when diode switching is used
- A switchable HF preamplifier and switchable attenuator increase the range of receiver sensitivity adjustments, which allow the operator to adjust the receiver to particular propagation conditions and the receiving antenna actually in use
- AGC is derived from the IF signal. AGC offers fast attack time and smooth operation (without any popping effect on strong signals) for fast and slow settings. It is even possibile to switch the AGC off, which is sometimes the last chance to copy extremely weak DX surrounded by strong signals—experienced DXers know it.
- A sharp IF crystal filter is close to the mixer and because of the relatively low IF (4.915-MHz), the crystal filter greatly attenuates out-of-IF signals. That helps to prevent receiver overloading by strong signals from outside the IF-filter pass-band. The IF crystal filter offers an adjustable passband for CW from wide (2000 Hz) to narrow (200 Hz).
- A low-phase-noise PLL local oscillator

Implemented microprocessor control offers:

- Split operation with two VFOs
- Dual-range RIT and XIT
- Memory operation for mode (CW or SSB), dual VFO A/B split operation, receive IF crystal-filter passband selection, receive CW sideband selection (allows canceling of one-side interference from strong nearby station by switching to opposite received sideband—a rudimentary IF-shift function),
- Direct keypad entry of frequencies and memory channels
- Three tuning rates: 1, 10 and 100 kHz per main-knob revolution
- 10-Hz tuning resolution
- Adjustable receive CW offset with a tracking sidetone
- Auxiliary I/O RS-232 interface for computer logging and remote-control purposes

The K2 itself is devoted to CW QRP enthusiasts, but could be tailored for other preferences by adding following options:

- The SSB option offers an adjustable speech compressor and optimized seven-pole, 2.2-kHz-wide IF crystal filter,
- 100-W PA Module (offered since the Dayton 2002 convention)
- 160-meter band with second receive antenna
- An automatic antenna tuner
- A noise blanker
- An auxiliary I/O RS-232 interface
- An audio filter, eliminating residual noises outside the desired passband

The K2 is sold in kit form with assembly instructions that are well written. Anyone can complete the kit and buy what one really prefers. The K2 Product Review, written by Larry Wolfgang, WR1B, appears in *QST* (March 2000, pp 69-74). "Impressions of the Elecraft K2 Transceiver" by Rich Arland, K7SZ, appears in *QST* (April 2001, p 99).

In designing the OMNI VI+, Ten-Tec has also departed from the prevailing general-coverage receiver concept and returned to ideas used 20 years ago. Ten-Tec have abandoned:

- Wide semi-octave, noisy first local oscillators generated by synthesizers
- First-IF up-conversion into the 50 to 90-MHz region
- Wide first-IF roofing filters

The OMNI VI+ HF transceiver is designed for ham bands only**,** from 160 to 10 meters. There are only two mixers in receiver chain: first IF = 9 MHz, second IF = 6.3 MHz. All ham bands are covered in 12 segments of 500 kHz, each having 30-kHz margins at lower and upper band edges. This model is a successful comeback of already proven concepts but with an implementation using present-day components:

- The first local-oscillator signal is produced with band-dependent crystal oscillators mixed with a low-noise 4.97 to 5.53 MHz PLL. Therefore, all synthesis noise problems causing reciprocal mixing have been avoided.
- The first IF is low enough to implement a narrow IF crystal filter with a good shape factor (having a passband adequate for SSB and CW) offering great attenuation of out-of-passband signals

The first IF is at 9 MHz and can be fitted with the following passband IF crystal filters:

- SSB: 1.8 kHz or 2.4 kHz

- CW: 250 Hz or 500 Hz
- A special 500-Hz, 6-pole IF crystal filter centered for digital modes

The second IF at 6.3 MHz can be equipped with the following bandpass crystal filters:

- SSB: 1.8 kHz
- CW: 250 Hz or 500 Hz

Such a mixing concept allows installation of narrow crystal filters in both IF chains right at the beginning of first and second-IF receiver amplifiers. Therefore the receiver main selectivity filters are close to the mixers, where they should be according to DXers—and where they are not in most ham radio HF transceivers made in the last 20 years.

Depending on chosen crystal-filter combinations, the following good shape factors should be achieved:

- 1.3 for 2.4-kHz first and second-IF crystal filters for SSB reception
- 1.4 for 1.8-Hz first and second-IF crystal filters for SSB reception
- 2.6 for 500-Hz first and second-IF crystal filters for CW reception
- 2.9 for 250-Hz first and second-IF crystal filters for CW reception

Other combinations of first and second IF crystal filters are possible. All installed IF crystal filters can be selected independently of the mode. Superior receiver selectivity significantly decreases interference even from very close signals.

DSP noise reduction (5 to 15 dB), DSP auto-notch elimination of interfering carriers and DSP low-pass (five choices) help to customize receiver selectivity in addition to the selectivity already offered by IF crystal filters.

**Influence of Phase Noise**

The main limiting factor of modern receiver performance is local-oscillator phase noise. Phase noise contributes to poor receiver BDR in the form of desensitization by nearby strong signals resulting from reciprocal mixing.

In the OMNI VI+, phase noise is –122 dBc for 1-kHz spacing, –123 dBc for 10-kHz spacing and –138 dBc for 20-kHz spacing. In the K2, phase noise is –120 dBc for 4-kHz spacing and –126 dBc for 10-kHz spacing.

Therefore, both OMNI VI+ and K2 have superb ham-band performance with an extremely high close-in dynamic selectivity. That enables reception of very weak signals from DX stations when strong signals are only a few kilohertz away. Several on-the-air A/B reception comparisons (using the same switchable receive antenna) of HF transceivers made by other makers against OMNI VI+ and K2 have been made recently. Generally, these comparisons favored the OMNI VI+ and K2, especially in the case of CW reception on 160-meter band.

Fig 1 explains the superior performance of the OMNI VI+ and K2. The figure demonstrates a typical situation where a barely heard DX station—only a few decibels above the receiver noise floor (dotted line)—is operating SSB on 14.195 MHz. That DX station is operating split and listening upward a few kilohertz. A pile-up of strong stations is calling where he is listening. For simplicity, only four signals are shown on the graph. Also for illustration, let us say that a QSO is in progress just 3 kHz higher on the neighboring frequency of 14.198 MHz.

The ability to copy such a weak DX station in presence of many nearby strong signals will depend on several receiver qualities: selectivity, BDR, IMD DR and the amount of phase noise on the LO signal.

We can presume that almost any modern HF receiver has enough sensitivity and selectivity to copy the weak DX station with no other signals present. Nevertheless, for real, on-the-air situations when plenty of strong signals are present near DX-station frequencies, some receivers will do better than the others. That will depend on how great is their BDR, how great is their IMD DR and how much phase noise accompanies the LO for any particular HF transceiver.

If the receiver has only average BDR, even a single adjacent signal —for instance, on 14.198 MHz, if it is strong enough—will desensitize that receiver and the weak DX station will not be heard in the presence of strong interference.

When many strong stations are calling, spread out 3-20 kHz up from weak DX signal, the IMD DR plays a big role in performance of the receiver. We can find in pile-up situations that many combinations of $2F_1 - F_2$ and $2F_2 - F_1$ are present. Those will produce intermodulation products on the weak DX station's frequency and these IMD products will interfere with or distort the weak DX signal. They can even completely bury the DX signal in noise and hiss. As the tables show, some receivers are more and some are less prone to IMD.

Most present-day HF transceivers implement synthesizers to produce LO signals for mixing. Analyzing BDR and IMD DR results, you can judge for yourself which makers do better and which ones are not as good—look for noise-limited remarks in test results. Some synthesizer designs produce more phase noise than one can obtain using methods implemented by Elecraft in the K2 and by Ten-Tec in the OMNI VI+ models. Therefore, K2 and OMNI VI+ models are better predisposed to deal with pile-ups on crowded ham bands.

The dotted line on Fig 1 indicates the receiver noise floor. The noise-floor levels of the OMNI VI+ and K2 do not change in the presence of strong nearby signals, because the OMNI VI+ and K2 have much less phase noise than most HF receivers using frequency synthesis. The dot-dash line illustrates the general situation for synthesized LOs. The presence of many strong signals near a weak DX-station frequency leads to the appearance of reciprocal mixing signals on the DX frequency that will

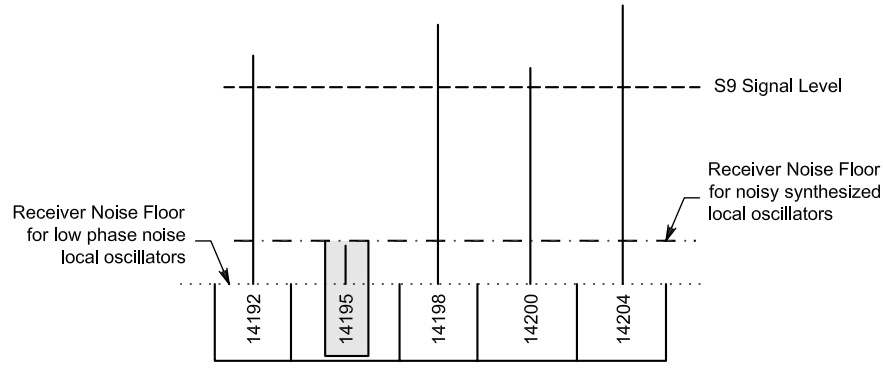



**Fig 1—A representation of a typical DX pileup situation in the frequency domain. Vertical lines represent the strengths of incoming signals. There is a weak DX station (shaded) at 14.195 MHz. The dotted line is the receiver noise floor for low-phase-noise receivers. It does not change in the presence of nearby strong signals and allows the tiny DX station (shaded) to be heard. The dot-dash line indicates the noise floor for a noisy synthesized local oscillator, which has increased in the presence of nearby strong signals. The increased noise floor hides the DX station at 14.195 MHz. The dashed line is the S9 signal level.**

interfere with that signal. When LO phase noise and calling stations' signals are high enough, then reciprocal-mixing products can bury a DX station signal completely in noise. That case is illustrated by the shaded bar around 14.195 MHz.

**Summary**

The K2 by Elecraft and the OMNI VI+ by Ten-Tec are relatively new. American makers have manufactured both. As far as I know (as of October 2001), there is no response to the call for superior dynamic range from other makers of HF transceivers yet. DX hunters can optimistically expect that good times have come at last for them and other makers will offer their new models designed appropriately for DX hunting and contesting. Nevertheless, this is still a market economy and the next steps of other makers will depend on how much popularity and admiration the K2 and OMNI VI+ achieve among the DX community.

I've analyzed equipment-review articles published in *QST* and some articles devoted to receiver front ends published in *QEX* for some time now. At the same time, I was gathering components to build my own homemade dream receiver to perform better in extreme DX-hunting situations than equipment offered commercially on the market—European QRM on low HF bands is much, much stronger than in other parts of the world. I've planned to begin construction upon retirement. Recently, I've noticed that there are models on the market performing almost as well as I need. Additionally, Elecraft offers the K2 as a kit. Its many options can be purchased and tailored according preferences, without the unnecessary bells and whistles found in general coverage multipurpose machines.

According to W8JI, there is also a challenge for ambitious constructors to upgrade old R-4Cs having the narrow 600-Hz Sherwood roofing crystal filter in the first IF. You can replace the poor second mixer with a high-level-input doubly balanced low-noise mixer and add more gain after the narrow IF filters following the second mixer (using a solid-state IF amplifier instead of a tube version). An R-4C upgraded that way, with gain properly distributed in the receive chain, could offer better performance for extreme DX situations than most modern HF transceivers.

Perhaps I am an old-fashioned man. But my motto is: If equipment is designed properly to achieve best performance in some specific and narrow area—in this case solely for reception of weak CW and SSB DX signals only on crowded HF ham-bands—you can expect better performance from it than from general-coverage multiband machines.

Therefore, if an HF transceiver is used mainly for CW and SSB DXing only inside the ham bands, a general-coverage receiver with its associated up-conversion and its first-IF wide roofing filter is not the best way to reach the main goal. Adversely to the concept used in general-coverage receivers, the main bandwidth selection should take place at a point as close to the front end as possible. That will enable us to achieve the greatest immunity against strong adjacent signals.

Unfortunately, that crucial demand is not acted upon in most of HF transceivers offered in the ham-radio market at the present time. Being myself a devoted DX hunter, I recognize the concepts implemented by Elecraft in the K2 and Ten-Tec in the OMNI VI+ as a step in the right direction. To meet demands of DX hunters, first of all, we need very good receiver performance and immunity to strong adjacent signals. In my opinion Elecraft in the K2 and Ten-Tec in the OMNI VI+ have properly designed receiver front ends for DX-oriented hams. This article was written late in the autumn of 2001. Since then, Ten-Tec has announced their ORION Model new HF Transceiver. I believe this is a big step in the right direction.

**References**


I've drawn material from many sources to produce this article. These include:

Many Product Review articles in *QST*. My articles published in SP HF Magazines

J. Devoldere, ON4UN, *Low Band DXing*, second edition (Newington, Connecticut: ARRL). The third edition is available from ARRL as Order No. 7040, ISBN: 0-87259-704-0; $28.

Web sites: W8JI's (**www.w8ji.com**, receiver measurements as of 8 Aug 2001) and several others: **sherweng.com/table.html**; **drakelist@baltimoremd.com**; **www.tentec.com**; **www.elecraft.com**; Elecraft mailing list **Elecraft@mailman.qth.net** (throughout the summer and autumn of 2001).


*SP7HT has been involved in DX hunting for last 45 years. During the first 25 years of his activities all of his HF equipment was homemade (including SSB crystal-filter production). Until recent years, a homemade rig has been the only way to be on the air from this part of the world. The last 20 years he has used several ICOM and Kenwood HF transceivers, but he's had no experience with other makers.*

*Tadeus was the very first DXer from Poland to reach the DXCC Honor Roll (1981) and DXCC Honor Roll #1 (1986). For last 28 years his occupation has been associated with microwave satellite telecommunication. Actually he is a specialist at the Polish Telecom Satellite Services Center "TP SAT" in Psary, Poland. He will retire in January 2003.*

# *Software-Defined Hardware for Software-Defined Radios*

*Using programmable logic in Amateur Radio applications.*

By John B. Stephensen, KD6OZH

Recently, I decided to upgrade my homebrew HF transceiver.[1] The goals of the new design were to replace most of the analog filtering and analog-control circuitry with software using digital signal processing (DSP) and to provide separate transmit and receive signal processing for full-duplex operation with amateur satellites. I also suspected that DSP could be used to improve AGC action and noise blanking.

My first impulse was to use a DSP chip—either on an evaluation board or by making a board with high-speed ADCs, DACs, digital up-converters and

[1]Notes appear on page 50.

153 S Gretna Green Way
Los Angeles, CA 90049-4015
kd6ozh@arrl.net

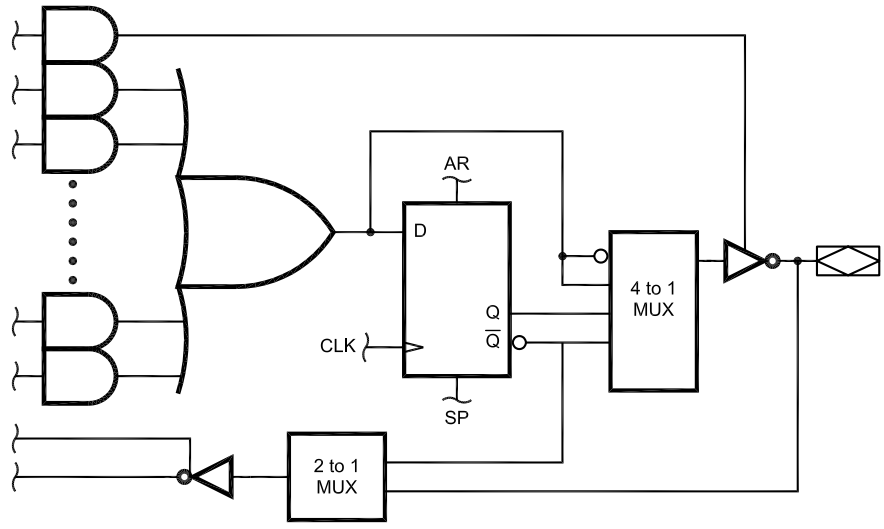



**Fig 1—A SPLD macrocell (source: Lattice Semiconductor Corporation).**

down-converters. The only inexpensive board uses the Analog Devices ADSP-2181 and that is going out of production. I looked at newer DSP chips, but evaluation board prices are in the $300 to $1000 range. The chips themselves are inexpensive in quantity but most are packaged only in ball-grid arrays. I also evaluated ASICs (application-specific integrated circuits) designed for the wireless infrastructure market such as digital up- and down-converters. Some are available in QFP packages with 0.65 or 0.8-mm-pitch leads, but they are expensive, their dynamic range is limited, and they are not optimal for the narrow-band modes amateurs tend to use.

After doing several paper designs and estimating costs, I took a different approach by using programmable logic devices (PLDs). PLDs are better suited to Amateur Radio applications than ASICs because they are more customizable. The facilities that are needed for narrow-band modes like AM, SSB, MFSK16, PSK31 and CW can be programmed into these parts. In addition, implementing only the functions that are needed for amateur applications minimizes the cost of a software-defined radio.

Four types of PLDs are available: simple programmable logic devices (SPLDs), complex programmable logic devices (CPLDs), field-programmable gate arrays (FPGAs) and combinations of MCUs and FPGAs called a system on a chip (SoC). Since many outside of the computer industry are unfamiliar with these parts I'll describe them here.

### SPLDs

SPLDs have been available for 15 years. They consist of 8-10 D-type registers with programmable combinatorial logic ahead of the registers as shown in Fig 1. The registers may be bypassed to create pure combinatorial logic. The logic for each register consists of a number of **AND** gates that are connected to an **OR** gate that drives the register input. The multiplexers are configuration devices and are controlled by nonvolatile memory. The **AND**-gate inputs are also programmable and may be connected to any input pin, any register output bit or left unused (see Fig 2).

The original SPLD products used fusible links for programming, and they could be configured only once. Present devices contain EEPROM memories to hold the configuration data; they can be reconfigured 100-1000 times. Typical parts include the 16V8 (which has 8 registers and 16 inputs) and the 22V10 (which has 22 inputs and 10 registers).

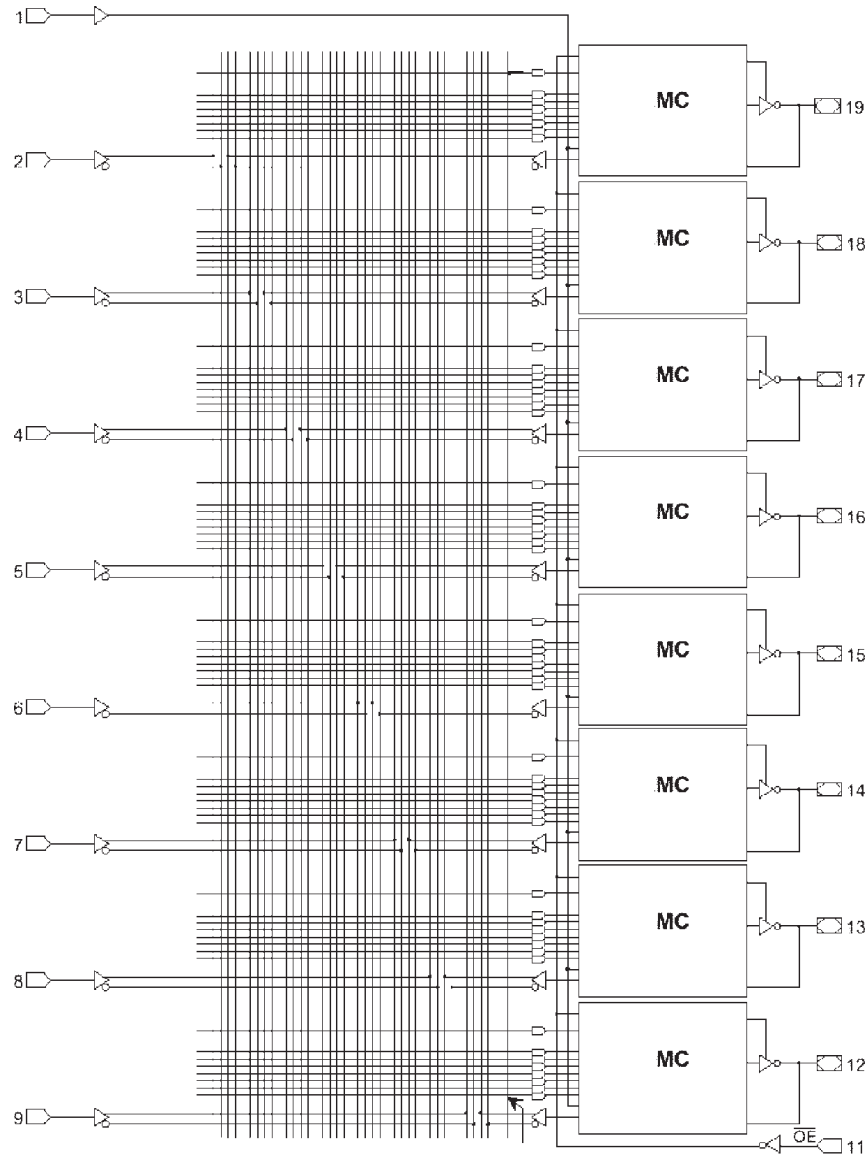



**Fig 2—A 16V8 SPLD programmable AND-array (source: Lattice Semiconductor Corporation).**

SPLDs are mature devices and are being replaced by CPLDs in new designs. Many CPLDs now cost less than SPLDs and offer more functionality.

### CPLDs

CPLDs build on the SPLD by putting arrays of SPLDs on a single chip. They provide more programmability in the form of programmable-interconnection arrays and specialized I/O control blocks. Fig 3 shows the general architecture of a CPLD. Notice that there are several clock, enable and clear pins, which have programmable polarity and may be routed to any *macrocell* via the programmable interconnect array (PIA). The PIA also accepts inputs from the macrocells. CPLDs may contain 32 to 512 macrocells in 2 to 32 logic-array blocks (LABs).

The macrocells in the logic-array blocks are the equivalent of SPLDs plus additional logic as shown in Fig 4. SPLDs typically have one dedicated clock input, while CPLDs provide multiple clock inputs and allow clocks to be derived from the programmable **AND**-array. Many manufacturers also provide programmable expansion of the **AND**-array where unused logic in one macrocell may be rerouted to another macrocell.

The I/O facilities contained in CPLDs are more flexible that SPLDs (see Fig 5). In a SPLD, each macrocell had a dedicated output pin. CPLDs have dedicated drivers next to the I/O

pins that may be connected to macrocells via a programmable switch matrix. Driver slew rates and logic levels are also programmable in many devices. Input logic levels also may be programmable and optionally registered.

### *CPLD Programming*

CPLD manufacturers provide software for programming CPLDs. This consists of a compiler that takes a description of the desired logic and creates the configuration data and a downloader that loads the configuration into the CPLD (see Fig 6). The logic description may be entered by drawing schematic diagrams or by entering Boolean equations in a language such as *ABEL*. The download software usually works with a cable that attaches a few pins on the CPLD to the PC parallel port. Most manufacturers provide a free version of the compiler and download software that is suitable for amateur purposes.

Schematic design entry is easy to use and works well for small designs. A typical application for a CPLD is a phase-locked loop (PLL). For example, the phase-locked crystal oscillator design that I published in *QEX*[2] can be simplified by using a CPLD to replace both the microcontroller (MCU) and PLL chips. The reference and VCO counter modulus can be programmed directly into the CPLD. Fig 7 shows the design of the phase detector. Fig 8 shows the design of the VCO and reference frequency dividers. The design software includes TTL MSI equivalents to minimize the design effort. Finally, Fig 9 shows the top-level schematic with the preset counter modulus. The modulus can be set to any value and programmed into the CPLD EEPROM configuration memory.

The PLL design fits into a 32-macrocell CPLD that costs $1 in small quantities from suppliers such as Lattice Semiconductor (Mach 4 series) or Altera (MAX 3032A series). The original PLL plus MCU cost was almost $10.

### FPGAs

The field-programmable gate array (FPGA) provides even greater logic densities. The original products had a "fine-grained" architecture. They consisted of a sea of gates that could be interconnected via programmable switches and busses. Anything could be constructed from the gates, but gate utilization of 50% or less was common because of routing limitations.

Recently, FPGAs have begun to look more like huge CPLDs as the basic cells have become more complex. The cells of modern FPGAs typically contain one or more look-up tables (LUTs) to define arbitrary logic functions and one, two or four output registers. These can be interconnected to form adders, multipliers or other functions that are commonly used in digital signal processing (DSP). The Atmel AT40K series is a good example as it is particularly suitable for amateur use.

The FPGA consists of a square array of 256 to 2304 core logic cells arranged as groups of 16 cells as shown in Fig 10. Between the groups of core cells are RAM cells that may be interconnected with the logic cells. I/O cells are located near the bonding pads around the periphery of the die. They may be connected to logic and RAM cells via direct connection or busses that are dispersed throughout the FPGA.

The core cell consists of two three-input LUTs and a D register as shown in Fig 11. There are four inputs to the cell: W, X, Y and Z. These inputs may

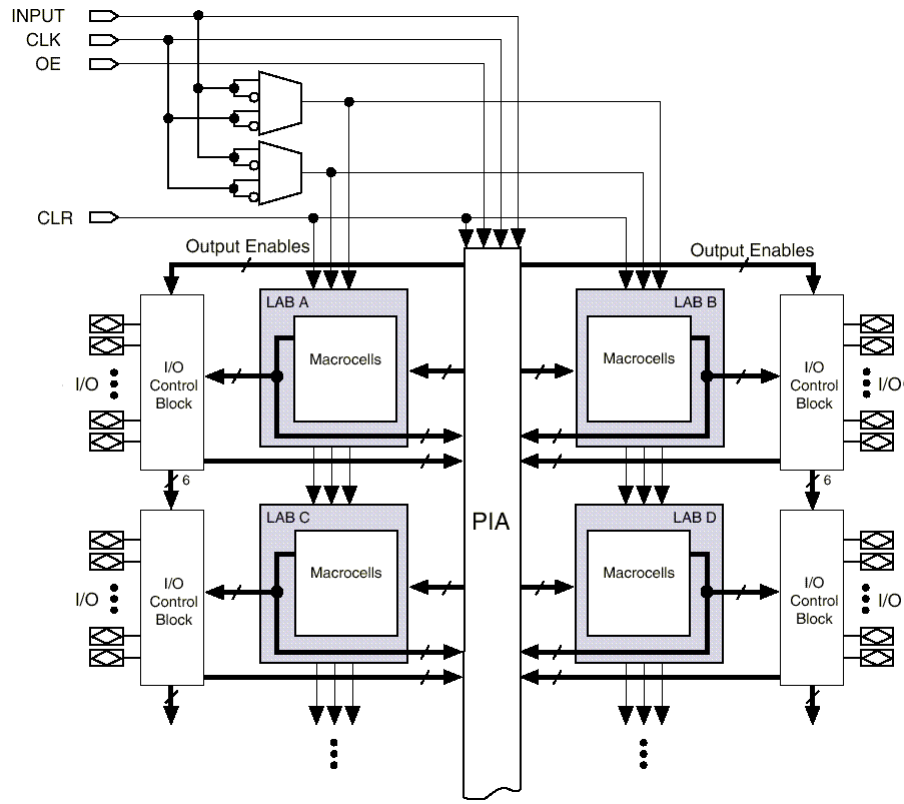



**Fig 3—A typical CPLD block diagram (source: Altera Corporation). See Note 4.**

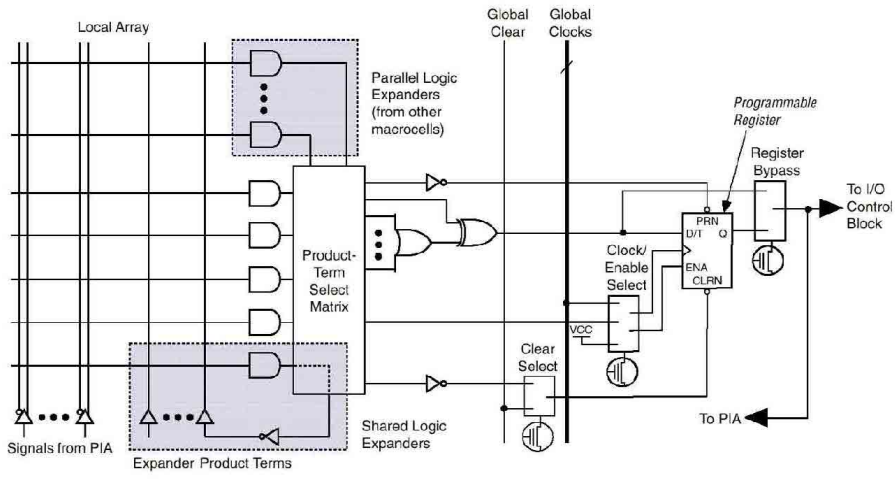



**Fig 4—A typical CPLD macrocell logic block (source: Altera Corporation). See Note 4.**

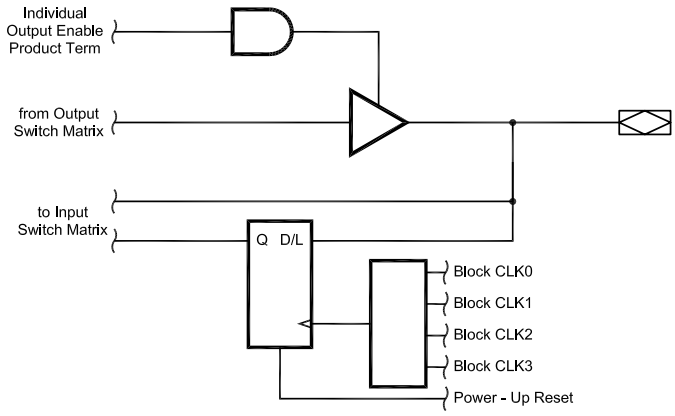



**Fig 5—A typical CPLD I/O control block (source: Lattice Semiconductor Corporation).**

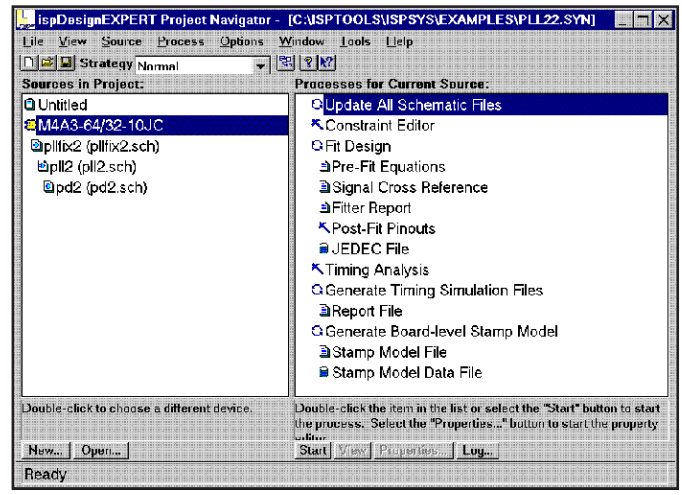



**Fig 6—CPLD design software for a PC. See Note 4.**

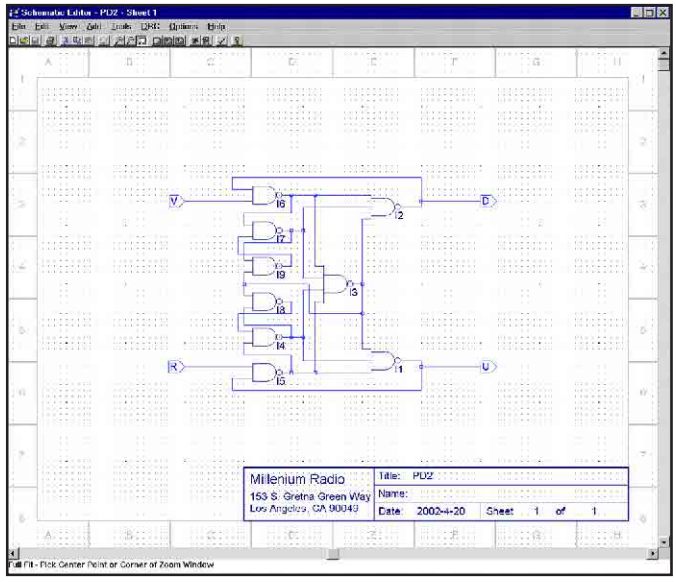



**Fig 7—Phase-detector schematic input. See Note 4.**

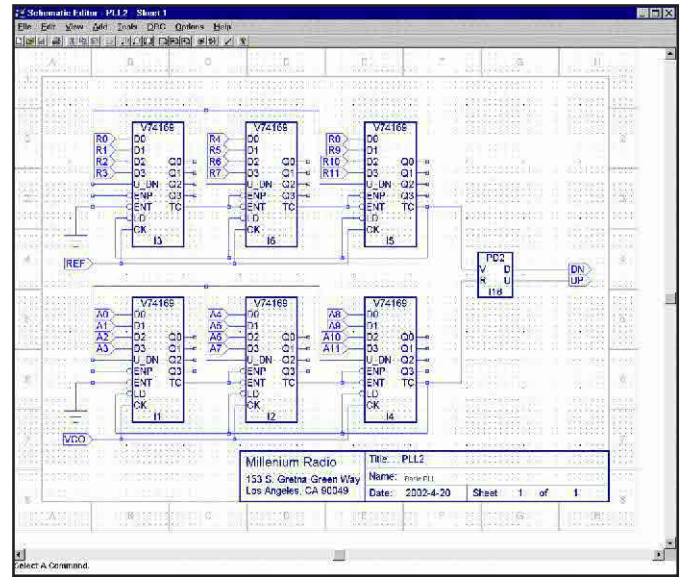



**Fig 8—Reference and VCO divider schematic input. See Note 4.**

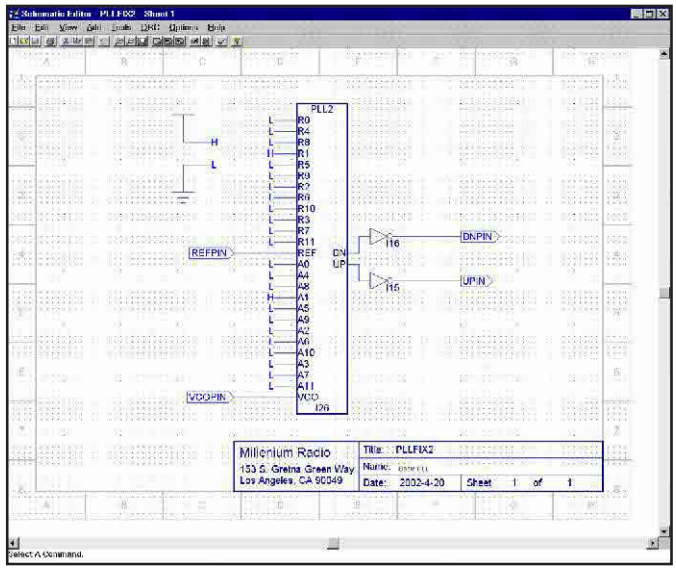



**Fig 9—A top-level schematic for a custom PLL. See Note 4.**

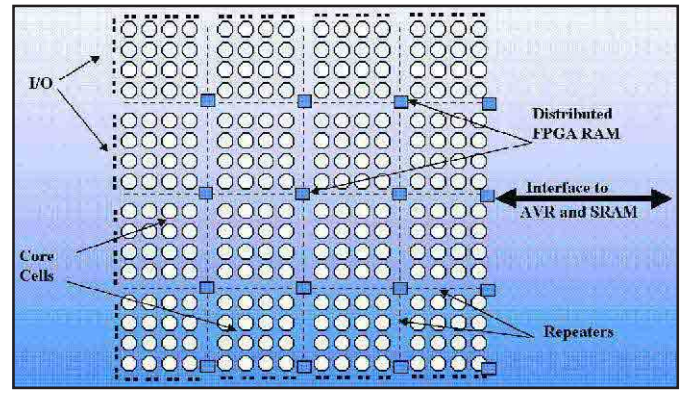



**Fig 10—AT40K FPGA architecture (source: Atmel Corporation). See Note 4.**

come from direct connections to adjacent cells as shown in Fig 12 or from busses that run throughout the FPGA as shown in Fig 13. The busses have programmable lengths. Bus segments may be isolated to a group of 16 core cells or interconnected via programmable repeaters to run through the entire array.

The core cell has three outputs that may come from the LUTs, the register or the tristate bus driver. The X and Y outputs may use the orthogonal or diagonal direct connections to adjacent cells shown in Fig 12. This is useful for fast-carry propagation and the construction of efficient parallel adders and multipliers. The tristate L output connects to one of ten bus lines adjacent to each cell as shown in Fig 13. The busses are useful for multiplexing data from multiple cells and connecting to input ports on multiple cells.

The core cells may be configured for various applications. A full adder is shown in Fig 14A. The adder may be combined with the **AND** gate in the cell to create parallel multipliers as shown in Fig 14B. The counter cell in Fig 14C shows how internal feedback may be used without requiring any external busses or connections. The cell may be used as plain combinatorial with three inputs and two outputs or four inputs and one output as shown in Fig 14D.

The FPGA also contains 2,048 to 18,432 bits of distributed RAM in 16 to 144 RAM cells as shown in Fig 15. Each RAM cell contains 32 4-bit-wide entries, and it may be configured as a single or dual-port RAM with synchronous or asynchronous I/O. In any configuration, the RAM has a 12-ns cycle time. Address and data ports are available via the busses shown in Fig 16.

RAM is desirable for many purposes including storage of data constants or microcode for FPGA processing elements. It can also be used to replace registers where access to individual bits is not required. Both applications allow the placement of more logic into each FPGA.

The I/O connection on the periphery of the FPGA die may be reached directly from adjacent cells or via the bus network as shown in Fig 17. Input and output pins may have dedicated registers or tie directly to cells or busses. Inputs and outputs can be programmed to operate at TTL or CMOS levels. The inputs may optionally have Schmidtt-triggers for noise-rejection and the outputs may optionally be tri-state to drive external busses. The drive current is programmable so slew rates may be limited in order to reduce EMI.

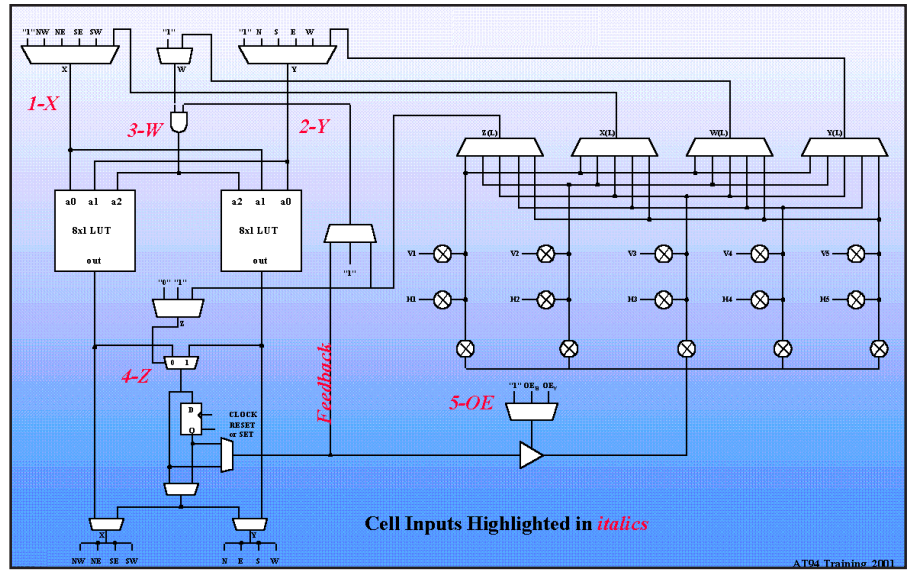



**Fig 11—AT40K core cell (source: Atmel Corporation). (Beware! Antel uses a circle with cross to indicate *switches*. They are *not* mixers.) See Note 4.**

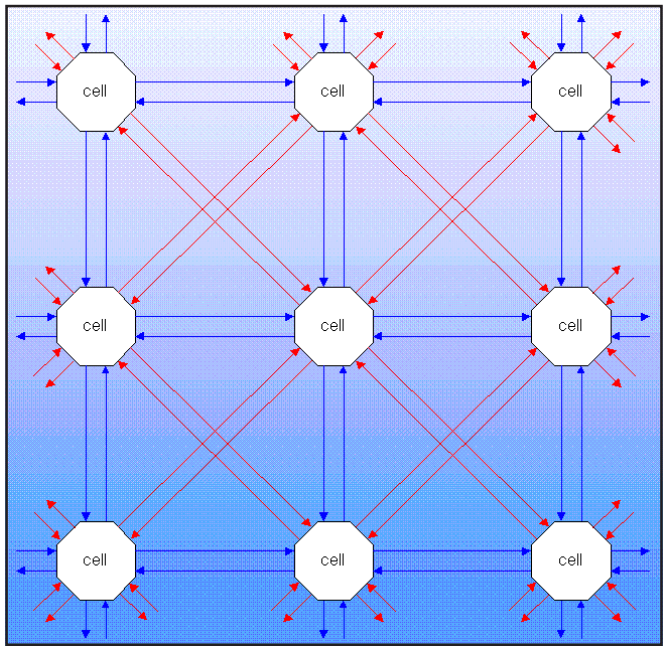



**Fig 12—Direct cell-to-cell connections (source: Atmel Corporation). See Note 4.**

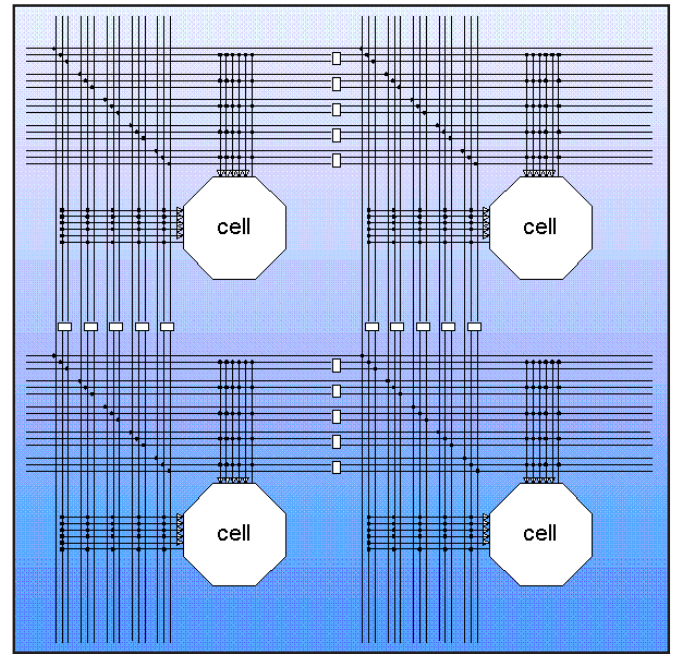



**Fig 13—AT40K busses (source: Atmel Corporation). See Note 4.**

### *FPGA Programming*

The FPGA manufacturer provides software packages for entering design information and compiling it into configuration bits for downloading to the FPGA. Design entry can be done via schematics or *ABEL* as with CPLDs or by using higher-level languages such as *Verilog* or *VHDL*.[3] These languages provide a way to describe the behavior of the logic and to create "test benches" to simulate the input to the logic and verify the correct output. A simple *Verilog* language description of combinatorial logic is shown in Fig 18.

The design software synthesizes a gate-level design and then places and routes the design for the FPGA being used. The result is a configuration file that can be loaded into the FPGA via a PC parallel port. One major difference between FPGAs and CPLDs is that the configuration is loaded into RAM on the FPGA. This has the advantage that the configuration may be changed as many times as desired. Configurations can even be changed in real time so only the logic needed for the current operating mode need be resident in the FPGA.

Atmel also provides a design language called *Macro Generation Language* (*MGL*) that allows the specification of logic designs with very tight control over the placement and routing in the FPGA. These macros can be optimized for minimal cell usage and/or maximum speed. *MGL* allows the creation of fast reusable modules that can be referenced from *Verilog* or *VHDL* files.

*MGL* is similar to many structured programming languages. A macro definition consists of an interface block and

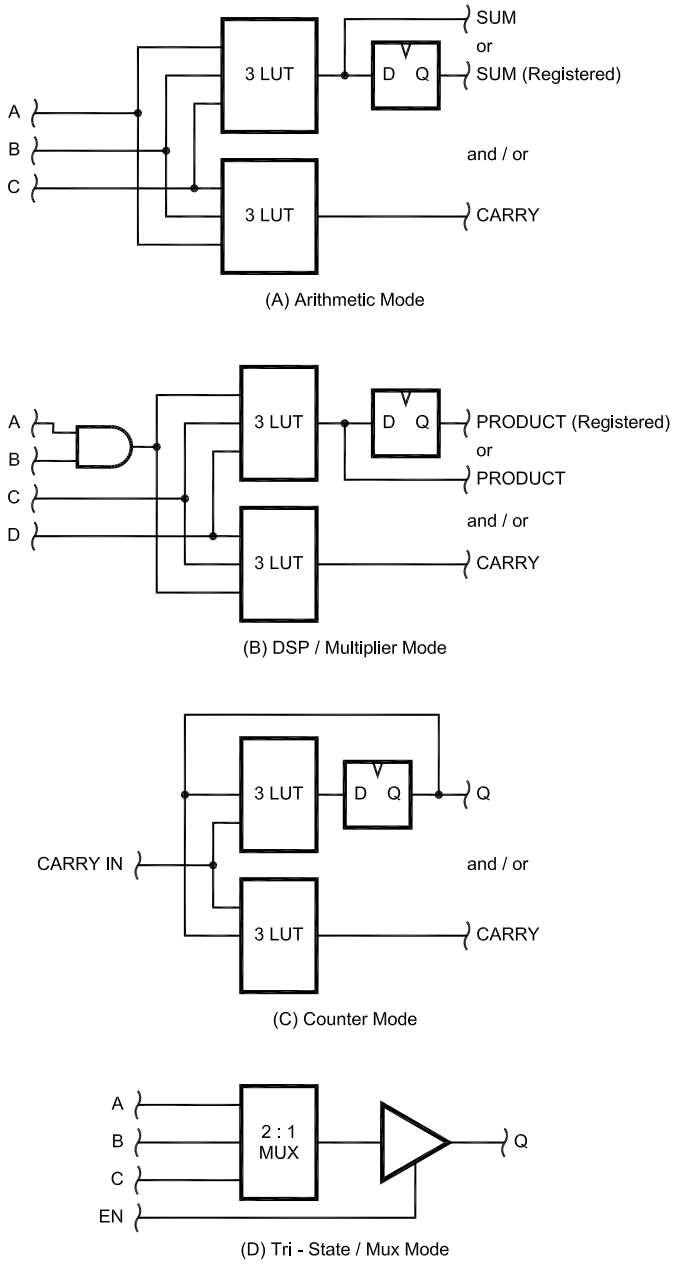



Fig 14—Core-cell configurations (source: Atmel Corporation).

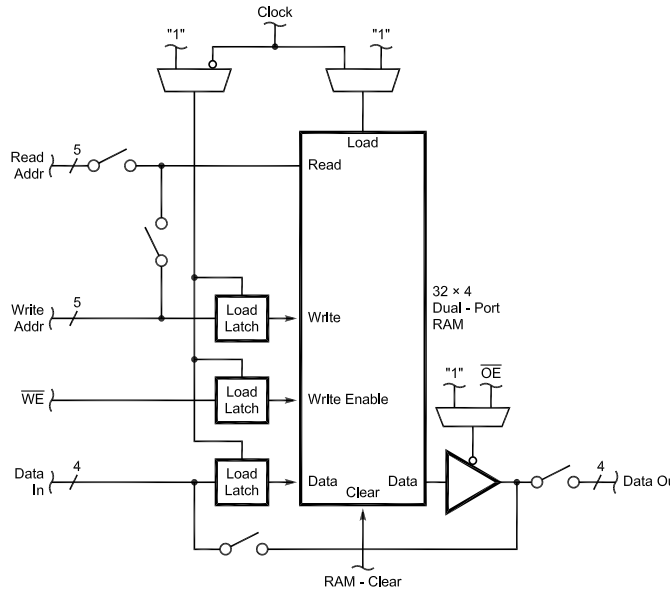



Fig 15—AT40K RAM cell (source: Atmel Corporation).

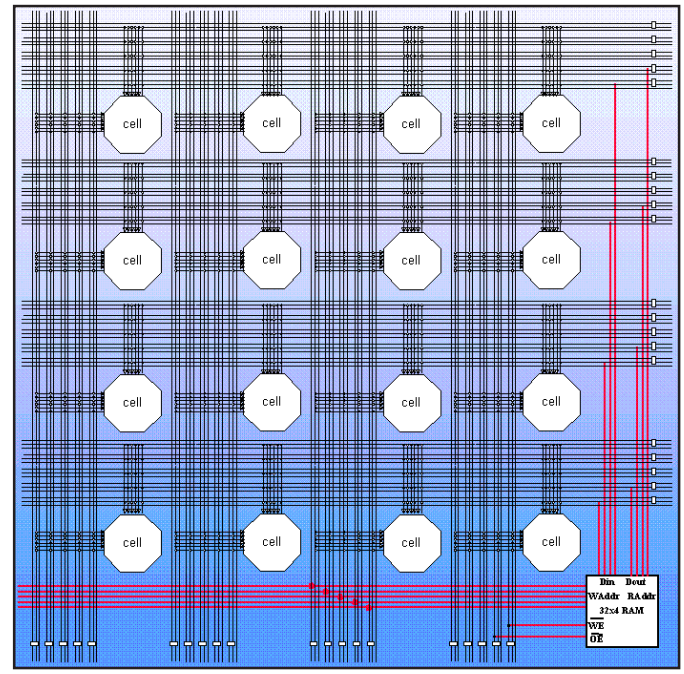



Fig 16—AT40K RAM cell bus usage (source: Atmel Corporation). See Note 4.

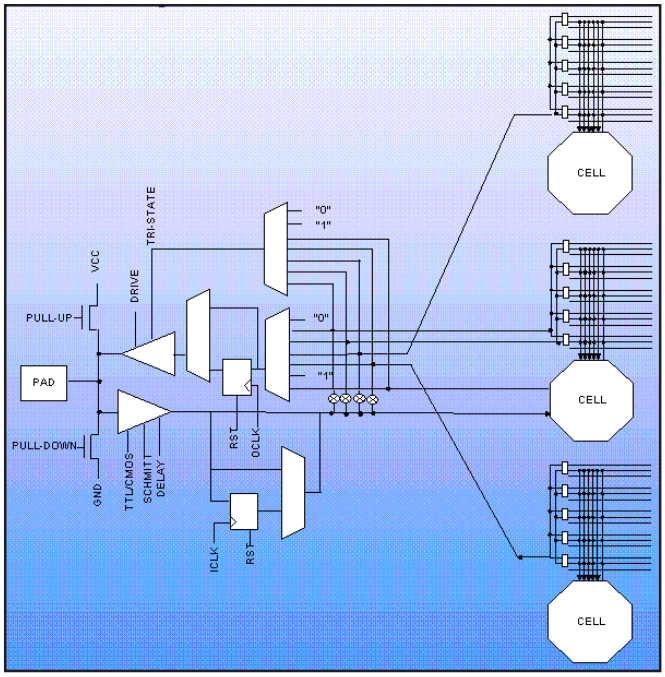
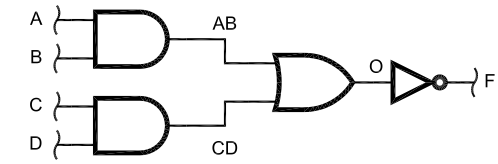


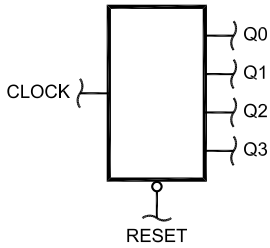

Fig 17—AT40K FPGA I/O (source: Atmel Corporation). See Note 4.




```
module AOI (A, B, C, D, F);
   input A, B, C, D;
   output F;
   wire F;
   wire AB, CD, O,

   assign AB = A & B;
   assign CD = C & D;
   assign O = AB | CD
   assign F = ~O;
endmodule
```

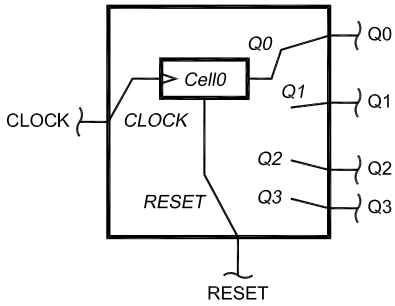
Fig 18—Schematic diagram and equivalent *Verilog* description (source: Doulos CBT).




```
counter : macro;
...

interface "Count" {width} of counter is
   inputports ( "CLOCK", "RESET" );
   for i in 0 to (width - 1) loop
      outputports ( "Q" {i} );
   end loop;
end interface;
```

Fig 19—Interface description in Atmel *MGL* (source: Atmel Corporation).

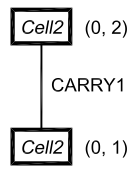



```
// Get macro from vendor library
aMacro   : macro := getmacro ( "FGEN1RF" );


instance "Cell0" of aMacro is
   location ( 0, 0 );
   functiong ( "!FB" );
   connections ( "CLK - > "CLOCK",
                 "RS" - > "RESET",
                 "G" - > "Q0" );
end instance;
```

Fig 20—Component "instantiation" in Atmel *MGL* (source: Atmel Corporation).

a contents block. The interface block defines the input and output ports for the macro. These are the signals that will be connected to external logic when the macro is used. Fig 19 shows one example, the interface to a four-bit counter. The snippet of *MGL* code defines **RESET** and **CLOCK** as the two inputs to the counter and Q0 through Q3




```
route of "CARRY1" is
   nodes ( (0, 1, "yOut" )
           (0, 2, yIn) );
end route;
```

Fig 21—Routing in Atmel *MGL* (source: Atmel Corporation).

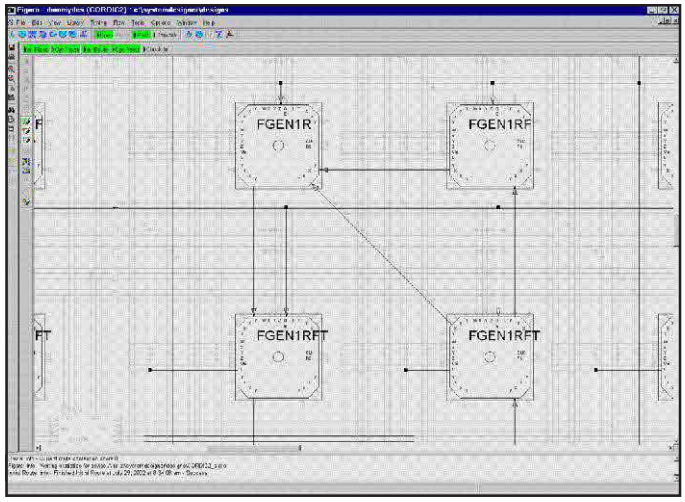



Fig 22—FPGA design software showing four cells and interconnection. See Note 4.

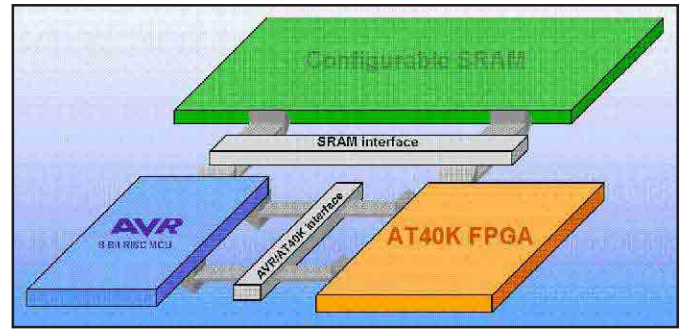



Fig 23—An Atmel FPSLIC (source: Atmel Corporation). See Note 4.

as the outputs from the counter.

The *MGL* contents block describes the underlying implementation of the macro. It *instantiates* components, connects the components together via nets and specifies the physical routing of these nets. Fig 20 shows the instantiation of a flip-flop and its connection to nets. First, the variable *aMacro* is assigned a value of FGEN1RF, which is part of the Atmel vendor library of dynamic macros. It defines a configuration of the AT40K core cell that has a LUT producing one output, which is stored in a register, and the stored value is fed back to the LUT.

The *location* statement creates an instance of the core cell and places it at the bottom left corner of the macro. This position is relative to the eventual placement of the macro. The *functiong* statement defines the contents of the LUT such that the output is the complement of the value fed back from the register. The *connections* statement then connects the ports on Cell0 to the ports of the macro.

Fig 21 shows how a direct connection between core cells is specified in *MGL*. The *route* statement contains a list of *nodes* that are to be interconnected. In this case, the Y output of Cell1 is connected to the Y input of Cell2. A more complex route, using a bus, would have a longer list of nodes and a specification of the type and location of the bus to use. The route can be specified to any degree of completeness as the routing can be completed using automated tools.

After the macro has been defined, debugged and executed, the generated macro can be imported into *Figaro*—the Atmel-provided tool for placement and routing on the FPGA. The process is similar to routing traces on PC boards. If the macro has been defined correctly, the cell placement will already be optimal. The automatic-routing software can be run to route any connections not fully defined in *MGL*. Fig 22 shows a close-up of four core cells after routing.

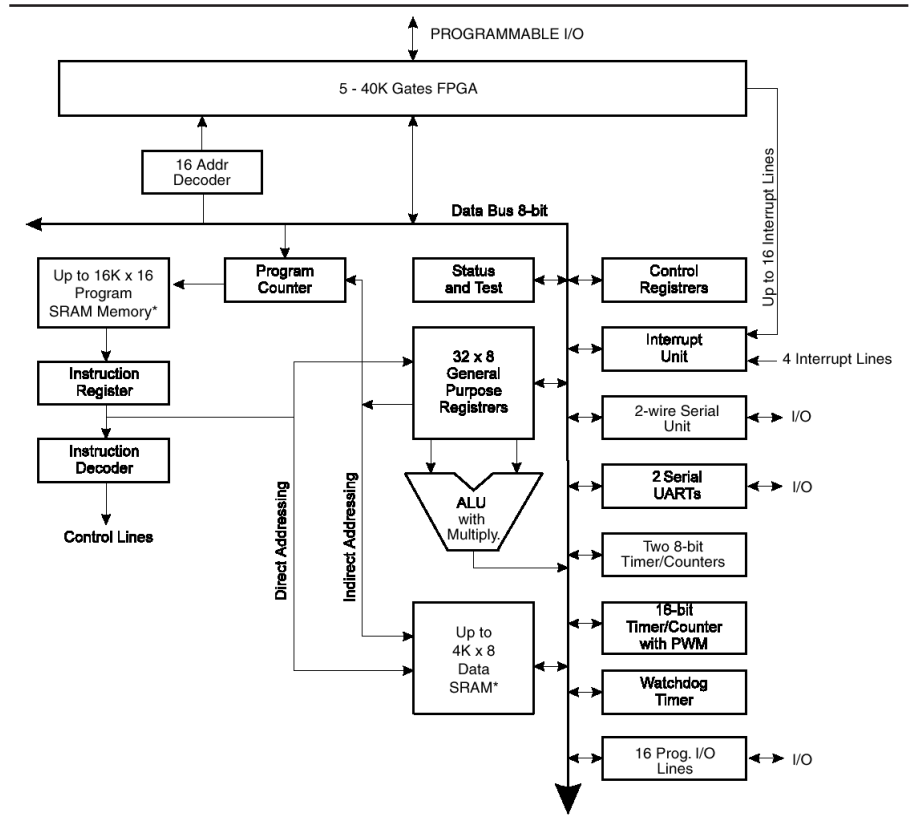



Fig 24—The AVR RISC MCU and peripherals (source: Atmel Corporation). See Note 4.

## SoC—System on a Chip

A recent trend in the semiconductor industry has been the introduction of various "systems on a chip." These products provide a CPU, memory and programmable logic on one die to minimize the size of portable programmable devices. The CPU may be either "soft" (a gate-level design that can be downloaded onto the gate array) or "hard" (a custom-designed CPU sharing the die with an FPGA). The custom CPU uses less die area, but there are few companies with both gate array and microprocessor products.

A software-defined radio requires several processing functions that have traditionally resided in multiple chips. A microcontroller provided general housekeeping functions. A specialized

DSP chip provided filtering, modulation and demodulation. Multiple PLL and DDS chips provided frequency control. Multiple chips required long interconnections and tended to increase the level of spurious emissions.

A SoC with a CPU and FPGA can provide all major housekeeping, signal-processing and frequency-control functions. This simplifies the design and reduces cost without sacrificing any performance. The SoC that I have selected is the Atmel AT94K10AL. Atmel calls this a field-programmable system-level integrated circuit or "FPSLIC." It contains a 20- or 32-MIPS 8-bit RISC CPU, two serial ports, counter-timers, 36 kB of fast dual-port memory and a 576-cell FPGA that can be programmed from the MCU. Fig 23 shows the major components in the FPSLIC and Fig 24 is the MCU block diagram.

## IC Packaging

One initial concern when selecting the SoC was the package size. The desire to produce small portable devices has driven package sizes down towards the size of the die. Technology has progressed from DIPs in the 1970s, to plastic J-leaded chip carriers (PLCC) in the 1980s, small outline packages (SOP) and quad flat packs (QFP) in the 1990s and now the ball-grid array (BGA) packages. A BGA package has all connections on the bottom of the package using a rectangular grid of solder bumps spaced as close as 0.8 mm. BGAs are mounted on a PC board that has solder paste silk-screened onto the mounting pads. The assembly is then exposed to a hot inert gas that melts the solder bumps and the solder paste to attach the component to the board. BGA packages are not suitable for home projects.

Luckily, manufacturers of components for industrial control and professional video/audio equipment do not have to produce tiny components for tiny products and continue to use SOP, QFP and LCC packages. Small CPLDs with 32-64 macrocells are available in a PLCC-44 package. Several FPGAs in sizes up to the 1200-cell range and the Atmel FPSLIC are available in PLCC-84 packages. The PLCC was originally designed to ease transition from through-hole to surface-mount PC-board technology. There are contacts on all four sides of the package spaced 50 mils (0.05 inches) apart. This package can either be soldered directly to the surface of a PC board or plugged into a socket. The sockets have leads on a 100-mil grid for compatibility with through-hole designs. This is ideal for construction of a prototype (or a one-of-a-kind unit) with point-to-point wiring as the sockets fit in pre-punched copper-clad boards.

## Design Software

This type of design moves much of the traditional hardware prototyping work onto the PC with software-based simulation. I use the Atmel-provided FPSLIC-design software that integrates the FPGA and RISC CPU programming and verification tools (Fig 25) into one package.

The MCU programming is done in assembly language. The AVR CPU is

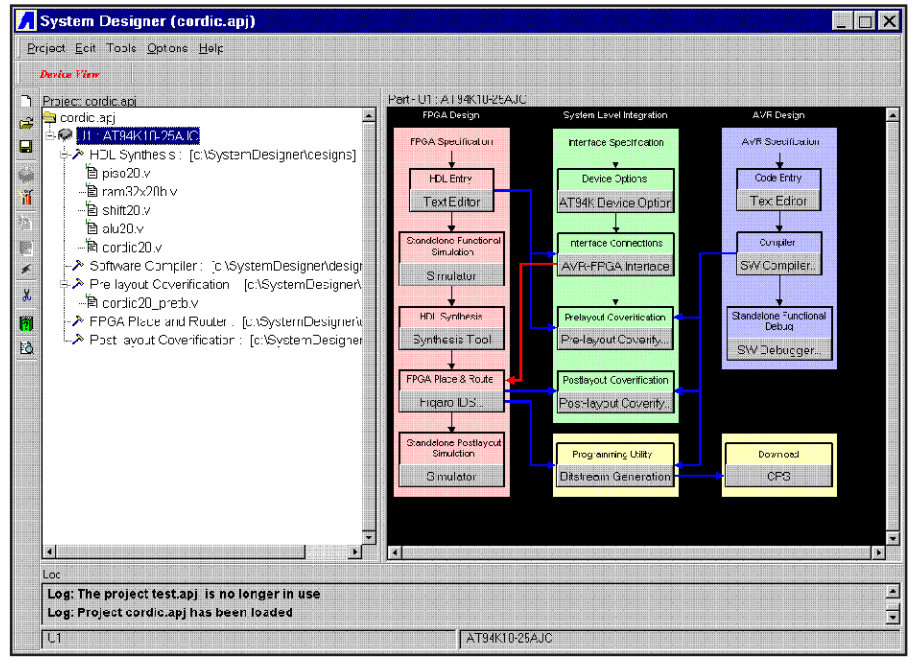



**Fig 25—FPSLIC design software main screen. See Note 4.**

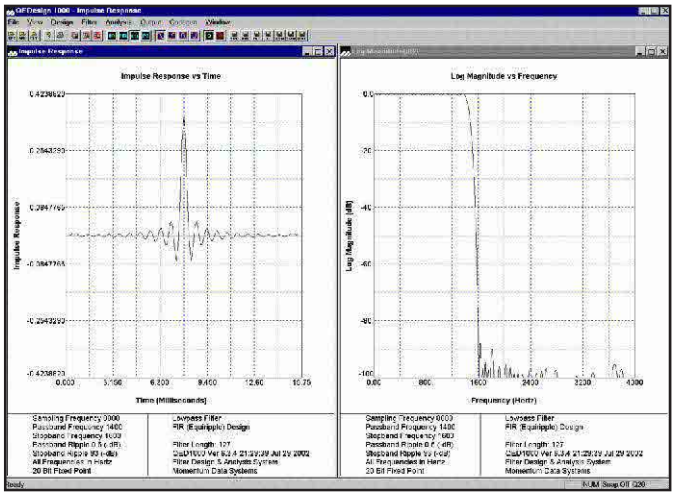



**Fig 26—DSP filter design software. See Note 4.**

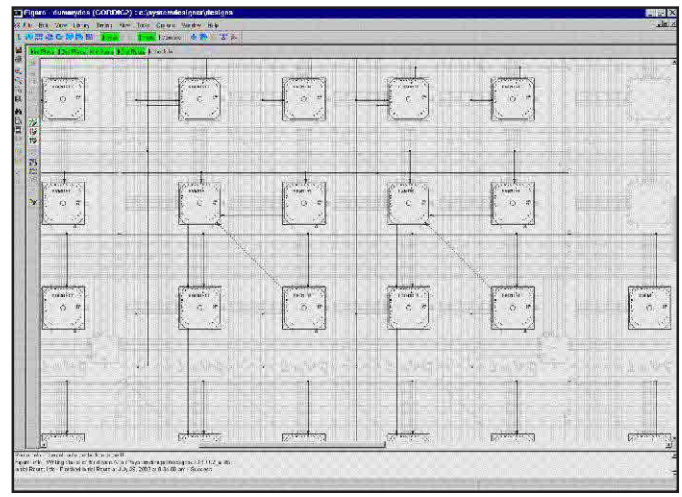


**Fig 27—The upper right corner of CORDIC quad serial arithmetic unit. See Note 4.**

a two-address general register machine that makes assembly-language programming easy compared to the old single-accumulator, single-address architecture used in 8051 MCUs.

The DSP filter-design software is from Momentum Data Systems (Fig 26). This software generates coefficients for either FIR or IIR filter designs optimized for a minimal number of taps for a given frequency response. There are several public-domain filter-design packages available on the Web that could be used in its place, and filter design could also be done with products like *MathCAD*.

### FPGA Performance Results

The DSP version of my transceiver has the last IF at 13-19 kHz. This frequency is low enough to allow use of low-cost 24-bit audio converters with high dynamic ranges, and it is high enough to allow use of low-cost monolithic crystal filters as roofing filters. The frequency-conversion scheme is similar to that used in the Drake R8 but with ferrite filters replaced by DSP. The FPSLIC generates and processes digitized baseband in-phase and quadrature signals at a combined 16 ksps rate.

The approach used has been to design macros that implement high-speed DSP functions efficiently in the gate array hardware and do the rest of the processing in software. The following functions have been implemented as macros for the gate array:

- Dual 40-bit DDS phase accumulator
- Dual 19-bit CORDIC (coordinate rotation incremental computer) phase-angle-to-amplitude conversion unit
- 20-bit MAC filter coprocessor unit
- 24-bit serial ADC and DAC interface

These functions are tied together and interfaced to the MCU in *Verilog*. This allows hand-optimization where needed for speed and quicker programming for control circuitry that has less stringent requirements. Low-speed functions, such as AGC, are implemented in AVR assembly language. The two-cycle multiply instruction in the AVR CPU is ideal for implementing DSP functions.

Hardware implemented in a gate array has a different set of constraints than hardware in ASICs or TTL logic ICs on a PC board. The designer must always be aware of routing delays. In the AT40/94K series FPGAs, a direct orthogonal or diagonal connection from cell to cell has a delay of only 0.1 ns. A connection via a bus can incur delays of up to 11 ns depending on bus length. Very often, serial implementations of arithmetic functions will outperform parallel implementations. The MAC and CORDIC macros were the most difficult to implement, and they use a combination of serial and parallel logic to minimize size while retaining maximum speed.

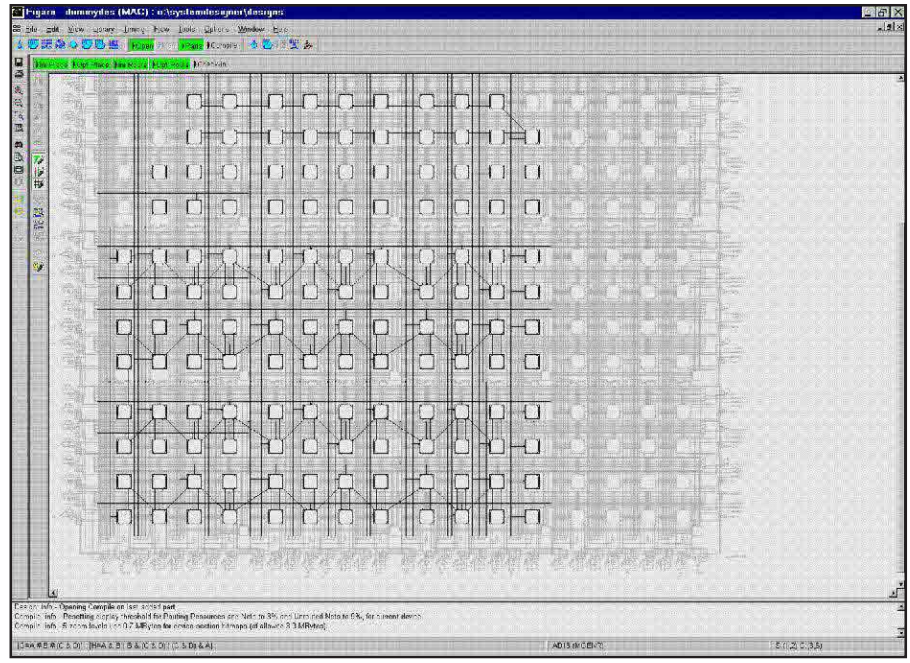


**Fig 28—A serial-parallel multiplier-accumulator unit. See Note 4.**

The CORDIC serial arithmetic unit requires only 98 FPGA core cells and is capable of 800,000 sine and cosine calculations per second. CORDIC is an algorithm that calculates sines and cosines using only shift and add operations. It is used to generate the frequency-reference signals for the transceiver and has spur levels below any current DDS ASIC.

Fig 27 shows the upper end of two serial arithmetic units that implement simultaneous bit-serial dual cross-connected shift and add operations to implement the core of the algorithm. The propagation delay in the most critical path has been reduced to 9.54 ns.

The MAC arithmetic unit requires only 145 FPGA core cells and is capable of 4.8 million 20-by-20-bit multiply-accumulate operations per second. Addition is done with 20 bits in parallel and multiplication is done by serial add and shift operations. The accumulator is 44 bits wide to accommodate all 40 bits of the product and prevent rounding errors. Four additional bits are provided to the left of the decimal point to prevent overflow when the transient value of the sum of products exceeds ±1.

Fig 28 shows the entire serial-parallel multiplier-accumulator unit. Orange cells (light squares) are used in the macro and gray cells are unused. The high packing density is achieved with serpentine routing that minimizes the length of several vertical and horizontal delay paths simultaneously. The maximum delay in any bit-serial data path is 9.17 ns. The 10.16-ns delay shown in the figure is for one multiplicand data bit, which changes state only once every 20 clock cycles.

### Conclusions

The FPSLIC has proven viable for use in SDRs, and it provides a better solution for narrow-bandwidth modes than do ASICs designed for wireless applications. A follow-on article will describe the hardware that surrounds the FPSLIC to convert between the digital and analog domains and translate signals to and from the 16-kHz IF.

#### Notes

[1]J. Stephensen, KD6OZH, "The ATR-2000: A Homemade, High-Performance HF Transceiver," *QEX*, Pt 1, Mar 2000, pp 3-15; Pt 2, May 2000, pp 39-51; Pt 3, Mar 2001 pp 3-8; Letters to the Editor, May 2001, p 62.

[2]J. Stephensen, KD6OZH, "A Stable, Low-Noise Crystal Oscillator for Microwave and Millimeter-Wave Transverters," *QEX*, Nov 1999, pp 11-17.

[3]J. Wiseman, KE3QG, "Modern Digital Design for the Radio Amateur," *QEX*, Dec 1997, pp 3-12.

[4]Several of the figures in this article are captured from complex computer-screen images that do not reproduce well in print or in black and white. Interested readers can view these images full size and in color on their computers by downloading a package from the ARRL Web **www.arrl.org/qexfiles/**. Look for 9X02STEP.ZIP. □□

# Tech Notes

*[The SurCapAdapt test jig came to fruition while Dan was constructing a DSP-10 radio to be an IF in his microwave station. Here are the details for building the SurCapAdapt, a simple jig that holds SMT components undergoing measurements.*—Peter Bertini, K1ZJH, *QEX* Contributing Editor; **k1zjh@arrl.org**]

## SurCapAdapt

*By Dan Hinz, W6LSN, 1738 Manitou Ct, San Jose, CA 95120;* **w6lsn@arrl.net**

During my recent foray into SMT construction, I was concerned about determining the values of some components. While resistor values were easily verified on my DVM, I noticed that many SMT capacitors have no markings. My Autek RF-1 analyzer can measure values from 1 pF to about 0.01 µF, but I lacked a means to hold the SMT components for testing. The RF-1 was supplied with several alligator clips that I could use, but they were cumbersome to use on SMT parts, and too many parts ended up lost on the workshop floor. I needed a better way to hold capacitors while measuring them. So was born the *SurCapAdapt* in Fig 1.

### *Building the SurCapAdapt*

RadioShack's #278-186B solder-less style PL-259 connector, with it's stiff center pin on the "back side" (internal to the body) of the connector, is a key element in the construction of the *SurCapAdapt*. Other vendors may sell similar connectors, but I am not certain of this and I can't verify the suitability of possible substitutes.

1. Clamp the back of the connector (cable end) firmly in a vice—don't worry if it slightly deforms, the rear of the connector will be cut off and discarded in a later step. Carefully remove the center pin by grasping the pin with vise-grip pliers and pulling it out. Set it aside, as it will be needed later (Fig 2).

2. Cut off the rear (cable end, Fig 3) of the PL-259 housing using a hacksaw or Dremel tool equipped with a cutoff wheel. Screwing the connector onto a barrel connector provides some strength and prevents deforming the housing as it is clamped into the vice. Once the rear is removed, the cut surface of the PL-259 connector needs to be smoothed. This took me about 20 seconds using an orbital sander with 100-grit abrasive paper. The brassy color of the base metal should be visible, and the surface should be smooth and even (Fig 4). This area will be one of the two contact points made to the SMT device under test. Cut a radial groove on the newly sanded connector end with a hacksaw or a Dremel tool using a cutoff wheel. The depth of the groove can be shallo (Fig 5). This notch serves to grip and stabilize SMT components being held in the SurCapAdapt during measurements. If the notch catches the edge of a SMT body, it's deep and wide enough to do what is required.

**Fig 1—The author's *SurCapAdapt* mounted on the Autek RF-1 Analyzer.**

3. The $^{7}/_{16}$-inch diameter dowel serves as an insertion jig for reinstalling the center-pin assembly back into the PL-259 body. Drill a hole into the end of the dowel that is large enough and deep enough to clear the center pin of the PL-259. Set the connector face up on the edge of an open vice. Carefully press the center pin back in using the dowel. Once you're sure the pin is straight, set the PL-259 aside.

4. This step is tricky. We need to drill a hole along the axis and centered on the threaded end of a #6-32×$^{3}/_{4}$-inch

**Fig 2—(above)The center-pin assembly is removed from the connector by securing the connector body in a vice, grasping the thick pin with pliers and pulling the assembly free.**

**Fig 3—(right) Remove the knurled rear portion of the connector. Here the job is easily done with a band saw in the ARRL Lab.**

brass machine screw using a #55 drill bit. This hole must be drilled through the entire length of the screw body.

This step can be facilitated by making a wood jig to hold the screw to a drill-press table. Drill a #36 hole through a block of wood that is slightly less than 3/4-inch thick. Carefully thread the machine screw into the wood (Fig 6). Place the block on the drill press with the screw head downward and centered on the drill clearance hole in the drill press table. By the way, even after several attempts I was never to able to successfully drill a steel #6-32 screw, but I was successful with the first brass screw I tried.

Slowly drill through the full length of the screw. You may need to clamp the screw from underneath to prevent it from turning; the drilling tends to unscrew it from the block. Don't use a lubricant or cutting fluid; either one might impair the electrical performance or the following fabrication steps. Drill slowly and gently—back the bit out often to clear material from the hole. Align the hole as close to the center of the screw as possible. Don't despair if the finished product isn't perfect. As long as the screw clears and doesn't contact the connector body once it's installed on the connector center pin, it will do fine (Fig 7).

5. Remove the screw from the wood block. You may need to use vice-grips because the drilling has probably damaged the screw head beyond use. This is not a problem because the head will be removed and discarded.

6. Check that the threads at the end of the screw are not damaged. Screw on the #6-32 steel nut or #6-32 die. Cut off the screw head with a hacksaw or a Dremel tool equipped with a cutting disc. Removing the steel nut or die will dress the screw threads as it passes

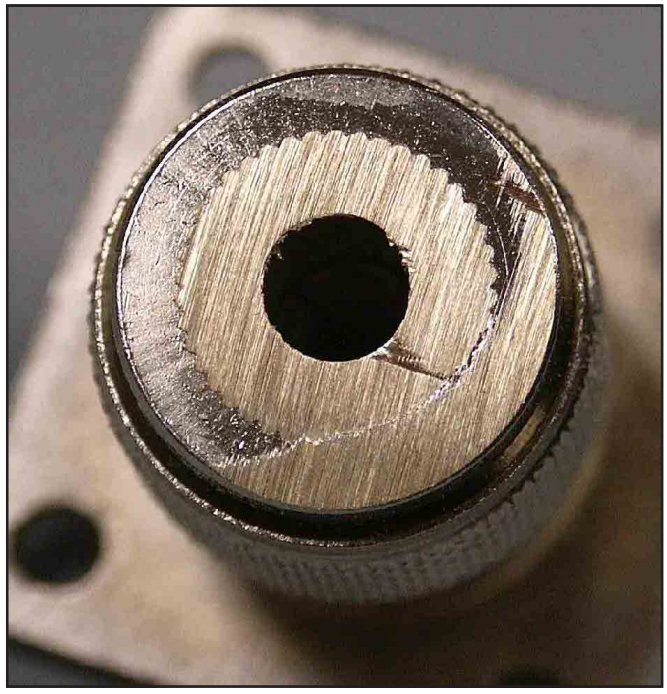


**Fig 4—After abrading away the saw marks, the rear of the connector is smooth, with some of the base metal showing.**

**Fig 5—A shallow notch in the rear face of the connector serves to contact and hold the device under test.**

**Fig 6—(above) To facilitate drilling a lengthwise hole in the #6-32×3/4 screw, drive it snugly into a 3/4-inch-thick board. Here we see the drilled end of the screw slightly protruding from the board.**

**Fig 7—(right) The prepared #6-32 screw is in place on the rear of the connector. It is important that the screw does not contact the rear of the connector anywhere.**

over them. Don't worry if the threads are not perfect on one end; just be sure that the end with the damaged threads goes on the PL-259 center pin when that step of assembly is reached.

7. Cut a scrap of PC board to approximately 0.25×0.6 inches (the length should be about equal to the width of the PL-259). Mark the center of the PC board and drill a hole large enough to clear a #6-32 screw. Tin the board with solder.

8. Lightly clamp the connector in a vice with the sharp end of center pin (internal pin) pointing upward. Strip about four inches of insulation from a length of stranded wire. Thread the wire through the entire length of the hole previously drilled through the #6-32 screw shaft. Now comes the tricky part.

Since the center pin is not easy to solder we will make an interference fit. At the end of the screw that will be attached to the PL-259 center pin, carefully untwist one end of the wire strands for about 3/8-inch. Cut off all but three or four strands of wire (Fig 8A). Pull the stranded wire back into the screw until about 1/8-inch of each uncut strand is visible. Form the wires back up over the end of the screw (Fig 8B). Force the pin into the hole where the three or four wire strands are folded back. This leaves a short length of reduced-diameter stranded wire inside the screw body—enough for center pin clearance and a snug fit, while the wire folded back keeps the wire from pushing back out. If there are too many wire strands, you will not be able to force the screw onto the pin. If there are too few strands, the screw will not fit the pin snugly and it will easily pull off the pin. The goal is to achieve both a good electrical connection and an adequate mechanical connection. It may take several attempts to get things just right. Check for a low resistance connection from the pin to the wire.

9. Put the PC board on with the copper facing the back of the connector (the surface that was sanded and notched earlier). The PC-board foil will form the second contact point to the SMT device under test. Screw the brass nut down the screw as far as possible. (You want to minimize the mass that you must heat while soldering.)

You should be able to feed some solder into the top of the hole in the screw and down to the top of the pin. I used 0.025-inch solder. Tin the wire and heat the top of the screw so that the solder melts and flows, securing the wire to the inside of the screw. This little bit of solder helps relieve mechanical stress on the three or four wire strands at the other end of the screw. Minimize the solder on the screw threads. (Actually, some is good. It keeps the nut from backing off.) Solder the other end of the wire to the PC board. Tin the wire to provide some additional strength, but not so much as to prevent you from bending it. The wire should be able to hold the PC board away from the connector against the force of gravity. Trim away any excess wire. Lastly, align the PC board with the groove on the back of the PL-259. Now, I hope you have something that looks like Fig 1.

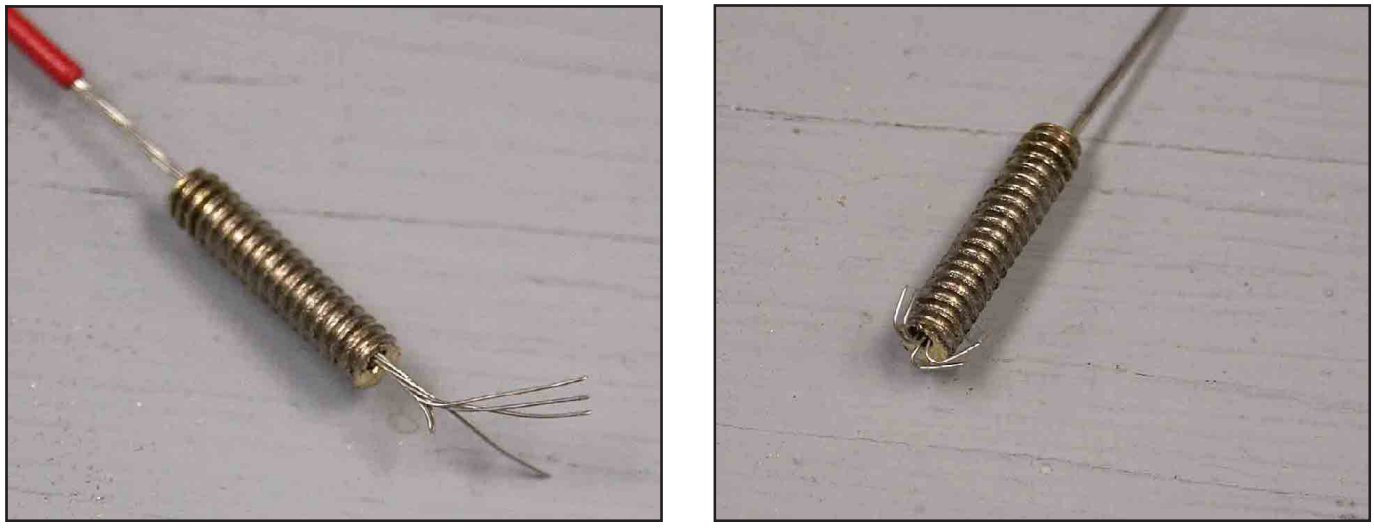
10. Be sure there is a good dc connection from the center pin of the connector to the PC board and that it is not shorted to the connector housing.

### *Check Out and Correction Factor*

Attach the adapter to your analyzer. Avoid twisting the wire or pulling on the screw. To determine the correction factor, screw the plates together onto a scrap of PC board, plastic bag or other insulator. Measure the *SurCapAdapt's* self capacitance. Mine shows 1 or 2 pF, depending on the measurement frequency. This represents the capacitance that is to be subtracted from a component's reading. You are now ready to measure a component. I hope that your analyzer's manual provides a graph similar to Fig 10,

**Bill of Materials**

| | |
|---|---|
| 1 | Solder-less PL-259 connector (RadioShack #278-186B) |
| 1 | #6-32×3/4 inch brass machine screw |
| 1 | Small section PC board material |
| 1 | #6-32 steel nut (#6-32 die may be used instead) |
| 1 | #6-32 brass nut |
| 1 | 7/16-inch-diameter wooden dowel a few inches long |
| 4 inches of #24 to #26 AWG stranded wire | |



(A) (B)

**Fig 8—Installing the stranded wire in the prepared screw. At A, about 3/8-inch protrudes from the screw and a few strands are removed. At B, the stranded wire is retracted back inside the screw leaving enough of the remaining strands exposed to fold back over the end.**

## A SurCapAdapt PL-259 Style

For Figures 2, 3, 4, 5, 6, 7 and 8, I constructed a SurCapAdapt in the ARRL lab. Since I was buying the RadioShack connectors, I also stopped at a local connector shop and bought some generic solderless PL-259s. The RadioShack connectors make the shield connection by means of a screw in the knurled rear section that pierces the cable jacket. The generic versions have a tubular rear protrusion meant to slide *inside* the cable jacket, where it is secured by a crimped ferrule (Fig A). While it may be possible to construct a SurCapAdapt fixture from such connectors, I did not bother. Instead, I constructed a SurCapAdapt from a *normal* PL-259 connector. Here's how:

Begin by removing the rear portion of the connector, leaving enough to protrude about 1/16 inch beyond the coupling ring when it is installed in a socket (Fig B). Smooth and notch the rear face as on the RadioShack connector (Fig C).

I then found that a #6-32 tap can successfully thread the dielectric and the inside of the center pin when started from the rear of the connector. I used a #6-32×1 1/4 screw that was on hand, which allowed it to penetrate about four turns into the center pin. A longer screw would be better to allow more turns into the center pin. Since the threading of the center pin stops, the screw makes good contact when driven into the bottom of the threads.

The screw I used is stainless steel, so I couldn't solder to it very well. Instead, I used a solder lug and an extra nut to secure it against the screw head. A wire from the lug to the PC board piece completes the PL-259 version of the SurCapAdapt (Fig D).—*Bob Schetgen, KU7G, QEX Managing Editor;* **ku7g@arrl.org**


**Fig A—The rear of a generic solderless PL-259 purchased locally is unsuitable for construction of a SurCapAdapt.**

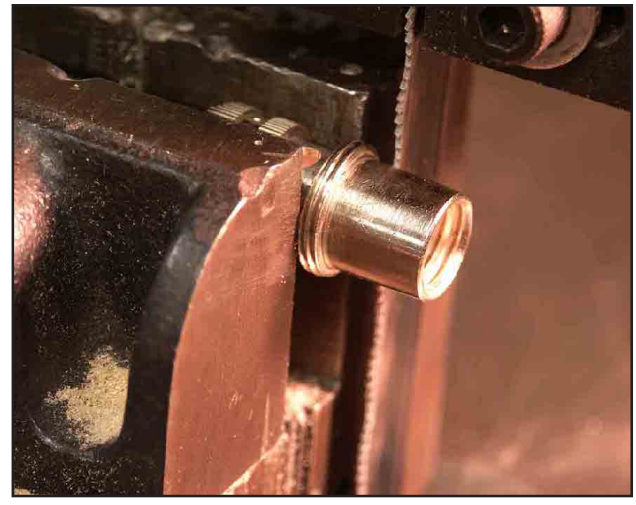


**Fig B—Trim excess material from the rear of the PL-259.**

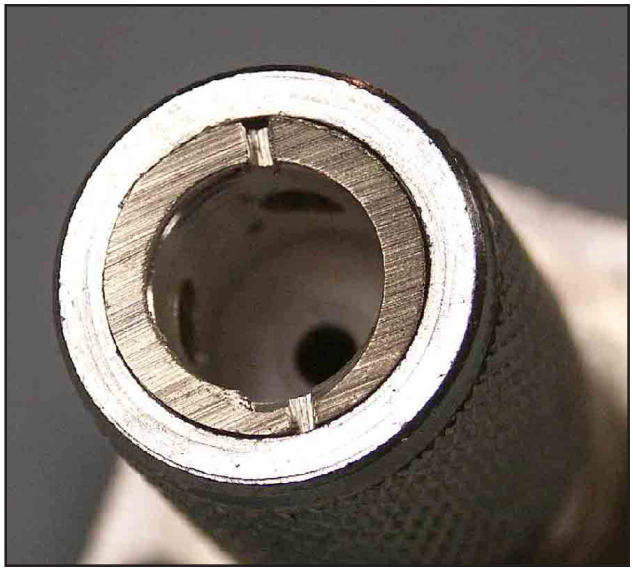


**Fig C—(above) Here is the trimmed, sanded and notched PL-259. I trimmed too much; it would be better if the notched shoulder stood at least 1/16-inch proud of the coupling ring.**

**Fig D—(right) A finished SurCapAdapt PL-259 style. I used a stainless-steel screw that was on hand. With a brass screw, the wire could be soldered to the screw head, eliminating the second nut and solder lug.**

which is taken from my RF-1 instruction manual.[1] I normally operate my analyzer at approximately 12 MHz, this permits me to measure values from 1 to 1000 pF with reasonable accuracy. With a frequency of 1.66-MHz, I can measure up to 0.01 µF.

Grasp the chip capacitor to be measured in a pair of tweezers and place it's body between the notch cut in the back of the connector and the PC board above it. Try to set the capacitor under the outline of the nut to reduce any angling of the PC board. You don't want the part to "squirt" out. Carefully tighten the nut until PC board is snug and holding the capacitor against the base. Over tightening will cause the capacitor body to tip and squirt out—gone forever! Switch on the unit and read the capacitance. The first capacitor I measured indicated 17 pF. Allowing for the internal 1-pF stray capacitance of the *SurCapAdapt* jig, yields 16 pF, which is darn close to the actual 15 pF marked on the capacitor.

The RF-1 manual gives additional insights for achieving accurate readings. The *SurCapAdapt* will allow you to measure SMT inductor values that are within the range of the analyzer. For the RF-1, that should be about 0.05 µH to 0.3 mH. I hope you find the adapter useful.

[1]Fig 9 is taken from the RF-1 instruction manual. (Instructions RF Analyst Model RF-1Autek Research 4/94) This figure is used by permission of Autek Research.

*Dan Hinz, W6LSN, was first licensed in 1973 as WN3VHS. He assumed his grandfather's call sign after passing his extra exam in 1999. While at the US Naval Academy he served as the president of the radio club and earned a BSEE. He served more than 11 years on nuclear submarines before leaving active duty for civilian life. When not shuttling his family to or from some activity, he is active on 40 through 10 meter CW or putting together a microwave station to do weak-signal and satellite work.*

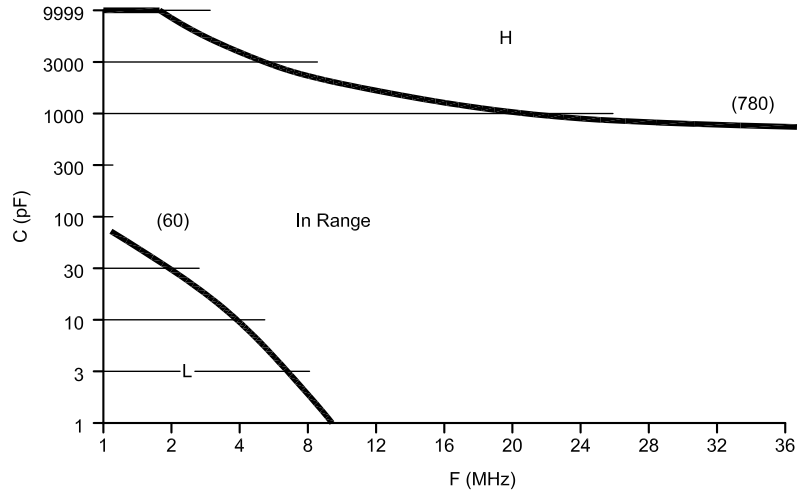



**Fig 9—A graph of the capacitance range for the Autek RF-1 analyzer.**

# RF

By Zack Lau, W1VT

### A Small 2-Meter Yagi

I've been working on a compact 2-meter Yagi for several years now—this is really a work in progress rather than a polished design ready for production (see Fig 1). It works well—except for the matching network—I'd be interested in hearing about designs that sacrifice a little bandwidth for lower SWR in the amateur band. Many hams would like the SWR to be less than 1.5:1, so the automatic SWR circuitry in sensitive radios does not fold back the power. The first prototype, with a carefully adjusted matched network, has a SWR of less than 1.5:1 across the band. The second one, built according to plan (Fig 2), has a similar SWR curve and an SWR of less than 1.8:1 across the band.

The Yagi was designed to fit in the trunk of my Saturn SL2 without any disassembly. The boom is made out of 32 inches of 1-inch-square aluminum tubing—you can make three booms out of one 8-foot length. A square boom makes it easier to drill the element-

225 Main St
Newington, CT 06111-1494
zlau@arrl.org

**Table 1—Frequency versus SWR**

| *(MHz)* | *Ant 1 (259B)* | *Ant 2 (259B)* | *Ant 2 (Bird)* |
|---|---|---|---|
| 141 | 2.1 | 1.8 | |
| 142 | 1.5 | 1.6 | |
| 143 | 1.3 | 1.5 | |
| 144 | 1.4 | 1.6 | 1.6 |
| 145 | 1.5 | 1.7 | 1.7 |
| 146 | 1.5 | 1.7 | 1.8 |
| 147 | 1.5 | 1.6 | 1.7 |
| 148 | 1.3 | 1.4 | 1.5 |
| 149 | 1.0 | 1.0 | |
| 150 | 1.4 | 1.6 | |

**Table 2—*Yagi Analyzer* Model (Element Center Spacing and Half-Element Lengths)**

1/4-inch elements
*144.000 145.986 148.000 MHz*

4 elements, inches

| | |
|---|---|
| | 0.250 |
| 0.000 | 20.230 |
| 6.950 | 20.000 |
| | |
| 29.600 | 17.190 |

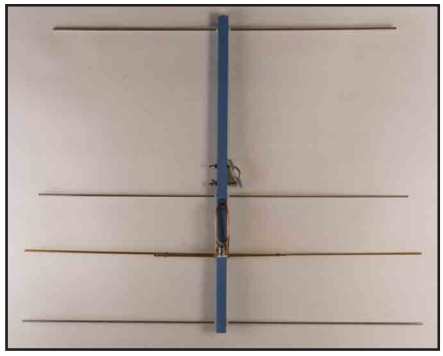
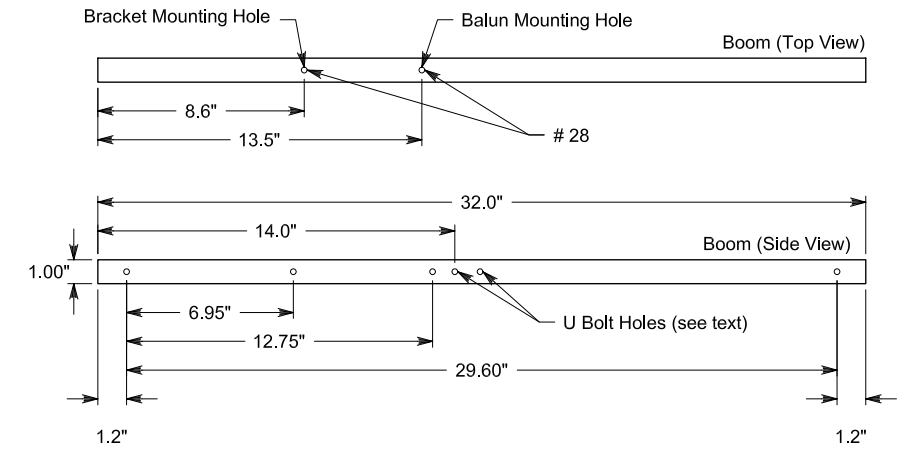


**Fig 1—A prototype of the four-element Yagi. The matching network has telescoping T-bars that are soldered in place after adjustment.**



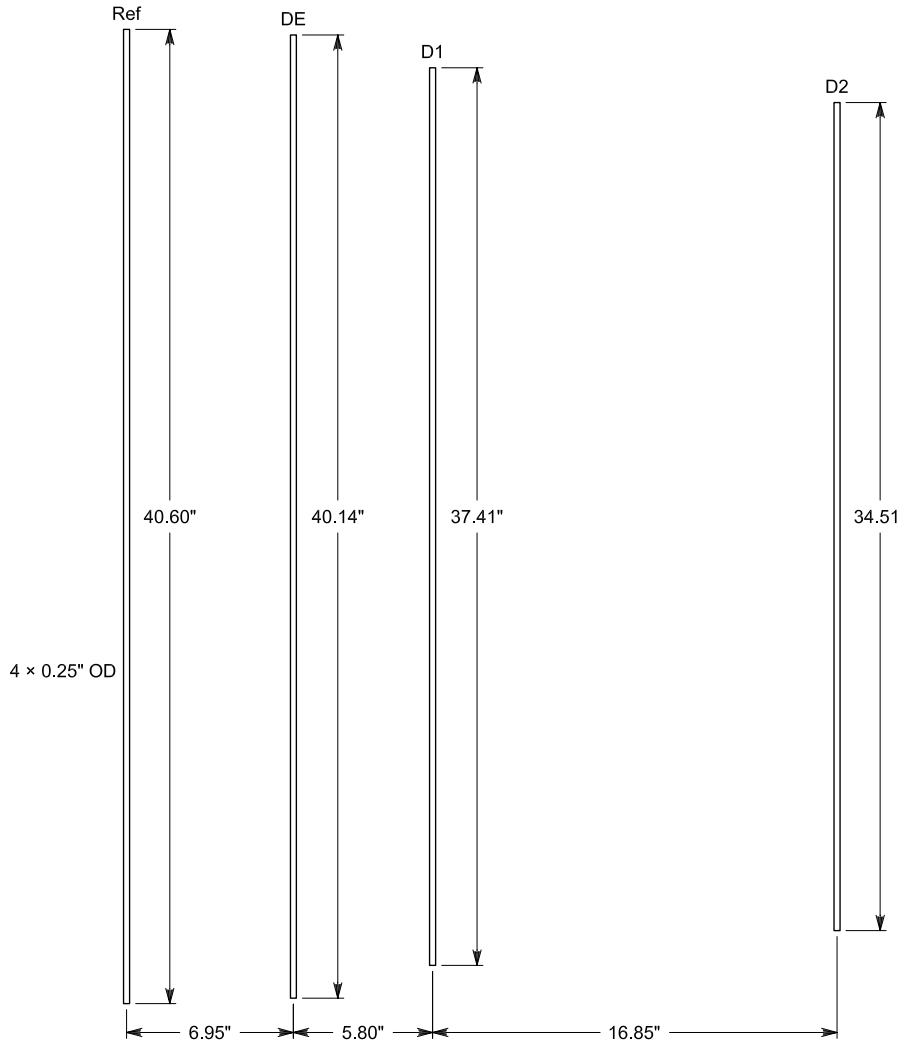




**Fig 2—The boom and elements.**

mounting holes accurately. A round 1-inch boom could be used with no change in element lengths or spacing. This antenna was designed for good performance across the band. The computer-calculated gain is 8.3 dBi across the band. The front-to-back ratio is at least 20 dB across the band. I used *Yagi Analyzer*, YA, the program that comes with *The ARRL Antenna Book*, for these computer calculations. Table 2 shows the computer file used by YA.

The most challenging part of designing a 2-meter Yagi is the driven element (see Figs 3 and 4). It needs to be simple and easy to construct yet weather resistant and capable of matching impedances efficiently, without distorting the antenna pattern. From an electrical standpoint, perhaps the simplest solution is to just split the element, and match it to 50 Ω. However, a split element is highly undesirable mechanically, unless someone devises an insulator with the excellent mechanical properties of aluminum or brass. I decided to use a T-match with a machined Teflon center insulator that helps keep everything in alignment. The resulting driven element is relatively rugged, with minimal wind loading. This reduces the stress on the mast in adverse weather conditions.

I made the balun out of λ/2 of semi-rigid UT-141A 50-Ω semi-rigid coax (see Figs 5 and 6). It is easily coiled to save space. I published an analysis of this design in the Jan 2000 *QEX*, in an article titled "Feeding Open-Wire Line at VHF and UHF." I use a copper strap (Fig 7) to attach it to the boom. It is a good idea to make the exposed center conductors long, so they may be wrapped across the T-bars. This makes it more difficult for the antenna to come apart. I machined a Teflon insulator (Fig 8) to fit across the T-bars—this also helps to hold the driven element together.

The T-bars are a little unusual in that they are actually thicker than the driven element. Experimental trials and computer modeling show better performance than with thinner bars. The bars are made out of brass tubing. The driven element is also made out of brass, so the driven element can be easily soldered together for excellent electrical performance. It is difficult to attach aluminum and copper antenna parts together in a fashion

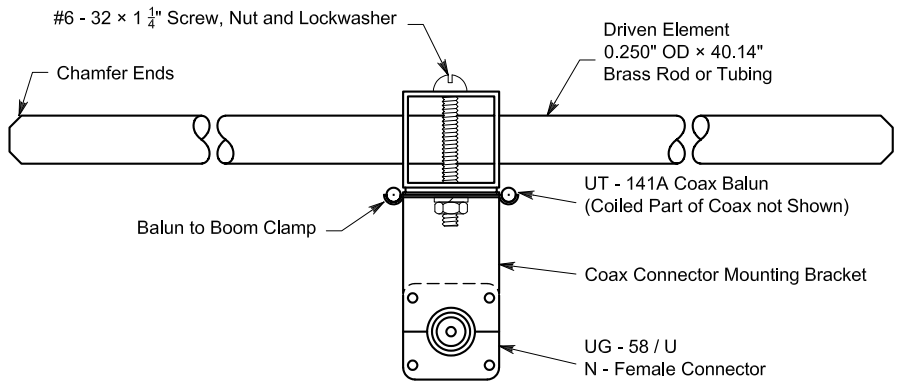



**Fig 3—The Yagi feed system before the N connector screws and T-bars are attached. This view is from the reflector side.**

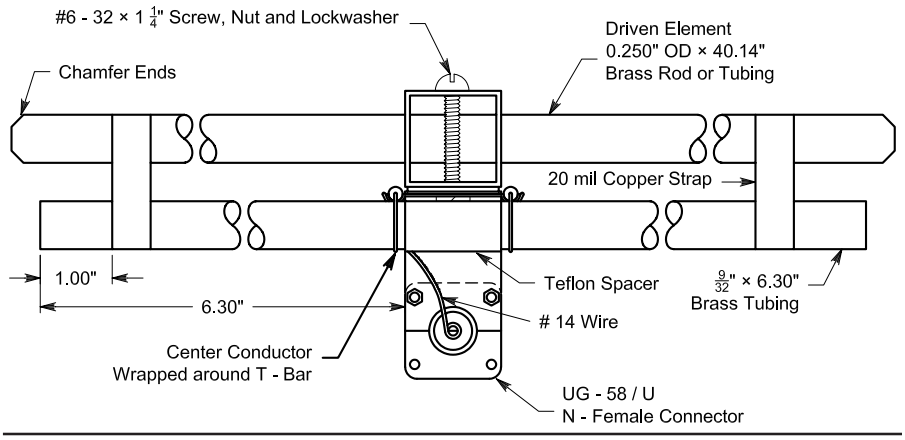



**Fig 4—The fully assembled Yagi feed system. AWG #14 wire connects the center pin of the UG-58 connector to one of the T-bars.**

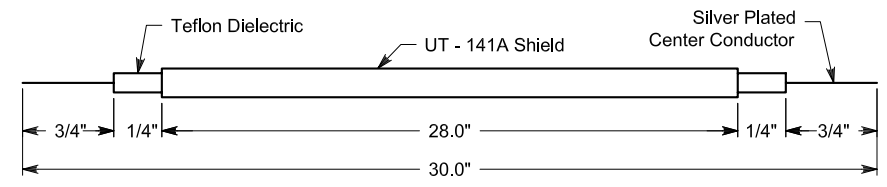



**Fig 5—Use λ/2 of semi-rigid coax for the balun.**

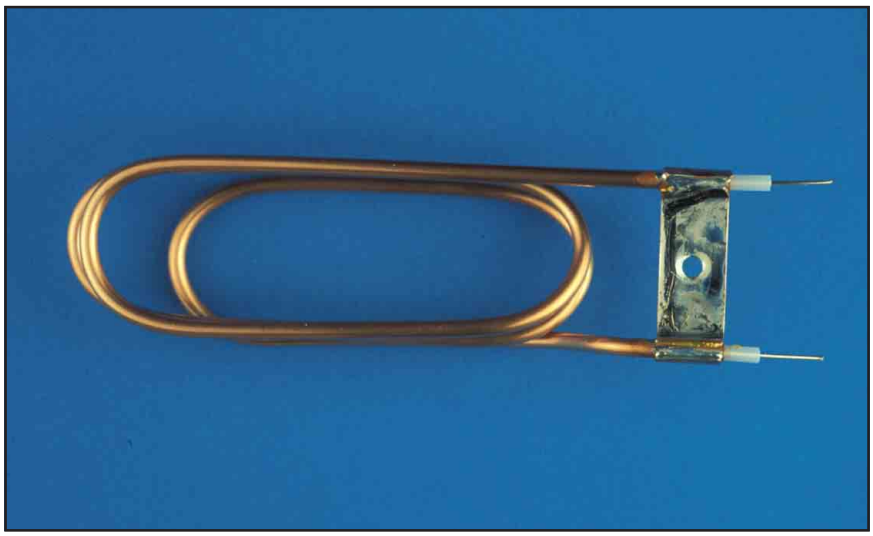


**Fig 6—(left) A prototype of the 2-meter balun assembly. The exposed center conductors are a little shorter than in the final version.**

that will withstand the normal stresses experienced by antennas.

The parasitic elements are made out of 1/4-inch aluminum rod. I've made 2-meter antennas using 3/16-inch rod, but the elements are rather flimsy. The 1/4-inch elements are a bit more material-efficient with this design—both directors can be made out of a single 72-inch rod. With 3/16-inch rod, the lengths become too long, requiring two lengths of 72-inch rod. Computer analysis programs work with the free-space element lengths, which is a good approximation for a non-conductive boom. I use an insulated metal boom, which requires slightly longer element lengths to account for the detuning effect of the boom. *The ARRL UHF/Microwave Experimenter's Manual* has a chart on page 9-5 for figuring out the boom correction. A 1.0-inch boom is 0.012 λ on 146 MHz—the element correction is 0.0033 λ, or 0.267 inches at 146 MHz. However, this is for elements that are in electrical contact with the boom—insulating the elements reduces the correction by 50%. Thus, a mere 0.133 inches needs to be added for either square or round booms. This is rather negligible on 2 meters, just 0.3% of λ/2. However, this correction becomes more significant on higher frequencies, like the 70 and 23-cm bands.

## Construction

Start by studying my July 2001 RF column, which describes a small 70-cm Yagi. It has many similarities. The biggest difference is the use of 1/4-inch rod, instead of 3/16-inch rod. This does require the use of bigger insulators. Get the insulators first. There are suppliers of suitable insulators, but I've not purchased any myself, preferring to machine my own (see Fig 9) on a small lathe.[1] You may need to modify the dimensions slightly for the insulators you buy. Similarly, the **U**-bolt holes in the boom should match the **U**-bolt you buy. I like to machine my own saddles to precisely match the masts I use. The extra gripping surface of homebrew saddles allows finger-tight wing nuts to hold the Yagi firmly in place.

The **U**-bolt hole locations are designed for use as a single horizontally polarized Yagi or a vertically polarized Yagi mounted on a horizontal cross boom. Another alternative for vertical polarization is to rear

[1]Notes appear on page 60.

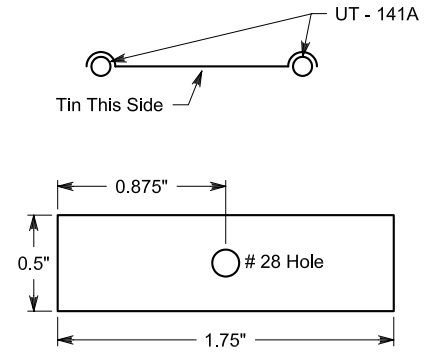



**Fig 7—Balun-to-boom mount: use 32 or 20-mil copper sheet.**

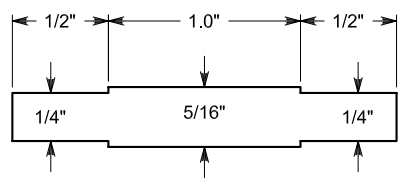



**Fig 8—The machined Teflon insulator between the T-bars.**

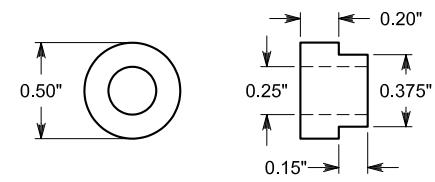



**Fig 9—Teflon insulators between the boom and elements. Two insulators are required for each element: one on each side of the boom.**

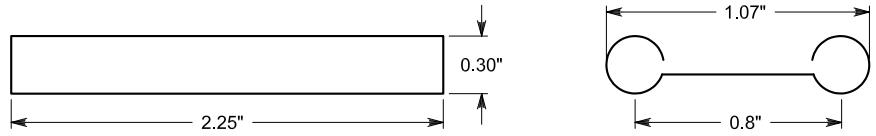



**Fig 10—Straps for the 1/4 to 9/32-inch T-match bars. Use 20-mil copper sheet.**

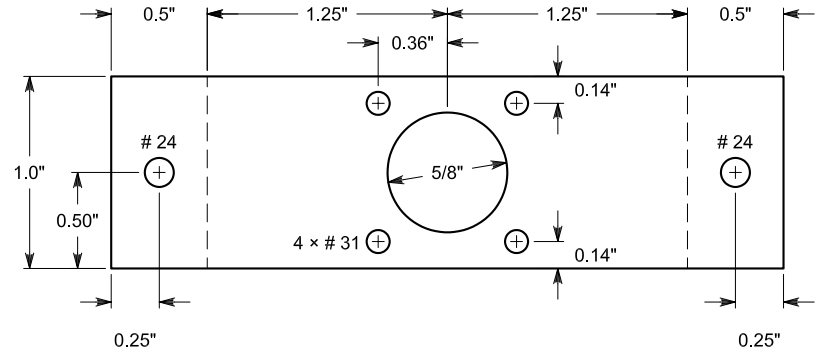



**Fig 11—An N-connector mounting bracket. I make two at a time, shearing the large hole in half as the final step.**

mount the Yagi—extend the boom and install the mast mounting hardware behind the reflector.

The first prototype used inexpensive aluminum rod purchased from Enco. Unfortunately, it was intended to be machined—the actual diameter was approximately 0.256 inches. The extra metal is useful in machine work—machining off the excess removes dings and blemishes for a nice clean surface. While the extra diameter has a negligible effect on electrical characteristics, the rod is too thick for standard external retaining rings. Instead of using retaining rings, I glued the elements in place using RadioShack silicone rubber sealant. I used inexpensive zinc-plated retaining rings sold by Small Parts to hold all the 0.250-inch diameter elements.[2] Texas Towers is a good source of 0.250-inch diameter 6061-T6 aluminum rod.[3] The 6061-T6 alloy has an excellent temper for aluminum elements.

The thicker elements should make it easy to drill the element-mounting holes. As the drill gets thicker, it tends to wander less as it passes through the boom. While it is possible to eliminate this wandering by drilling the holes on one side of the boom and then drilling them on the other side, this requires a great deal of accuracy in positioning the holes. It is easy to have two sets of holes that don't quite line up if the end of the tubing isn't cut perpendicular. Beginning metal-workers prefer to drill the holes in pairs—just push the drill through the boom to make both holes.

I find that the Vargus B30 deburring blade works great on cleaning up the boom holes. It removes the internal and external burrs simultaneously.[4] It is the best tool I've found for removing internal burrs in long metal tubes.

The first prototype used 20-mil copper straps (Fig 10), while the second used 32-mil copper straps to hold the **T**-bars to the driven element. 20-mil copper is much easier to work with, but less rugged. I don't recommend thicker straps—you can damage or even break a pair of pliers attempting to bend thick copper. I used a pair of pliers with thick round tips.

It is generally easier to make a 5/32-inch hole in the balun-to-boom clamp (Fig 7) with a hand punch than to drill a hole—copper is very stringy and difficult to drill cleanly. If you must drill copper, hold it in a heavy vise or clamp it down—copper straps will often grab a drill bit, becoming a dangerous propeller. I'd make the clamp after bending the connector bracket—you may need to move the exact location of the hole if you have trouble bending aluminum precisely.

The surface of the balun-to-boom clamp that touches the aluminum bracket should be tinned with solder to minimize corrosion. Solder is much more anodic than copper, and thus a better match to aluminum, which is very anodic. A Digi-Key RP323 black cable clamp and #6-32 hardware is used to clamp a coil of the balun to the boom.

When bending the semi-rigid coax for the balun, it may help to first concentrate on getting the ends of the coax aligned with the copper bracket. After the coax is soldered to the bracket, the remaining coax can be formed neatly into place. It may be a good idea to wear safety goggles—the coax can form a spring that splashes solder.

AWG #14 wire is used to connect the center pin of the UG-58 coax connector to one of the **T**-bars. Which **T**-bar you choose becomes important if you intend to stack Yagis. Choosing the other **T**-bar changes the phase by 180°. This allows you to flip one of the Yagis to minimize phasing-line lengths. Stacked Yagis perform rather poorly if phased incorrectly.

I make my connector mounting brackets two at a time, shearing the big 5/8-inch hole in half as the last step (see Fig 11). I find little advantage to

attaching the N connectors to the bracket with all four holes.

The prototype T match used 5.97-inch long 9/32-inch T-bars and sliding sections of 1/4-inch tubing. The extra sections added 0.33 inch to each T-bar. Brass tubing with 14-mil-thick walls is designed to telescope together. The sliding sections of tubing make it easier to adjust the assembly to the proper length. A difficulty with sliding sections is maintaining good electrical contact—they may work better if the tubing isn't quite round, so that the tubing binds. It is easy to solder tubing together once the optimum dimension is found. If the tuning changes after soldering, it indicates that the sliding section wasn't making good electrical contact.

I machined the insulators out of Teflon rod with a miniature lathe (Fig 9). UV-resistant black Delrin could also be used, as the element insulators are at low impedance points. The Teflon center insulator is across the 200-Ω balun connection. I cut the shoulders first and then drill the hole—the extra material makes it easier to cut the Teflon cleanly.

I made SWR measurements with an MFJ 259B SWR analyzer. I tried using an Autek RF-5 analyzer, but the SWR readings were abnormally low—1.0 across the entire 2-meter band on both Yagis! I suspect the detector isn't sensitive enough to accurately measure low SWR. Results similar to those of the 259B were obtained using a Bird wattmeter and a 2-meter CW transmitter. The antenna was approximately 6 feet above ground.

**Notes**

[1]Byers chassis sells black Delrin insulators and stainless keepers. **www.flash.net/~k3iwk/info.htm**. C3i sells molded Celcon insulators and stainless keepers; **www.c3iusa.com/**.

[2]SMALL PARTS Inc, 13980 NW 58th Ct, PO Box 4650, Miami Lakes, FL 33014-0650; tel 800-220-4242, fax 1-800-423-9009; **www.smallparts.com**.

[3]Texas Towers, A Division of Texas RF Distributors Inc, 1108 Summit Ave Ste #4, Plano, TX 75074; tel 800-272-3467, 972-422-7306; **www.texastowers.com**.

[4]Enco has three types, stock numbers 380-1524, 380-0234 and 380-1324. Enco Manufacturing, 400 Nevada Pacific Highway, Fernley, NV 89408; tel 800-873-3626; **www.use-enco.com**. □□

# Letters to the Editor

### HF Receiver Dynamic Range: How Much do We Need? (May/Jun 2002)

***Dear Sir,***

Peter E. Chadwick, G3RZP, reported on his findings about the dynamic range a receiver should have for adequate performance in the 7-MHz band. In 1995, I did some measurements to answer the same question and published the results later.[1] The approach used was different from Chadwick's methods. It consisted in measuring the power the antenna delivered after passage through a 6 to 8-MHz filter. Such filters were common at that time for good Amateur Radio receiver front-ends.

This is a worst-case approach because it assumes that there are always two frequencies of sufficient strength in the spectrum to cause interference to the received frequency of interest. Measurements were taken from an inverted V-dipole in a suburban environment continually for 30 days using an A/D converter and a computer. Data were averaged and converted to a diagram showing the variation in power level during a whole day. Maximum power peaked at –15 dBm for the average at 18:00 UTC. Taking into account the one-hour time difference between the UK and Germany, this is in accordance with Chadwick's data.

Of course, there were also days with signal levels up to –10 dBm. I wasn't able to measure the noise floor continually, but I took several measurements that were between –105 and –115 dBm using a CW (400 Hz) bandwidth. Maybe these were a few decibels too low. Nevertheless, these data are pointing to a required dynamic range of 90-100 dB. Because of rapidly varying band conditions in the evening, it's good to have some decibels of headroom available. I would therefore recommend a dynamic range of 100 dB for receivers with broad input filters.

In addition to the broadband measurements, I also made measurements with a relatively narrow bandwidth. In the early '90s, I was active in building QRP equipment for portable operation with low power consumption. Low power consumption and a receiver front-end having 100 dB of dynamic range are in sharp contrast. So, I asked myself if it were possible to reduce the needed dynamic range by a not-too-sophisticated filter. In the '70s and '80s, high-Q, two-resonator filters for 7 MHz were advocated here in Germany to reduce receiver IMD.[2] Using this information, I constructed a bottom-coupled Cohn filter with a 70-kHz bandwidth and 12-dB insertion loss centered at 7.020 MHz (European CW subband). I then connected it to a wattmeter and A/D converter. This narrow filter and the broad 6 to 8-MHz filter mentioned above were driven in parallel by a power splitter and the whole setup calibrated with a signal generator. As a 30-day measurement and a short control with a spectrum analyzer showed, the narrow filter isn't able to eliminate the power coming from the nearby BC-band. However, it eases the load to the receiver front end by more than 20 dB. Using such a filter, a dynamic range of 85 dB is sufficient for operation in the European CW subband.

As a whole, I came to the same conclusions as Chadwick, and it would be interesting to know if other hams in other locations have similar experiences.—*Werner Hallenbach, DK8PD, Lichtenbergerstrasse 68, 40789 Monheim, Germany;* **whallenbach@compuserve.com**

[1]W. Hallenbach, "Belastung von 40-m-Band-Empfängern durch BC-Stationen," *cq-DL* 1997, p 870. *["The burden on 40-m-band receivers from BC stations"—Ed.]*

[2]A. Heinrich, "Bessere 40m-Vorselektion am Beispiel von TS-930S und TS-140S," *cq-DL* 1993, p 540 and literature cited therein. *["Better 40-m pre-selection by example of the TS-930S and TS-140S"—Ed.]*

### A Homebrew Regenerative Superheterodyne Receiver (May/Jun 2002)

I would like to make available to you and the readers of *QEX* some additional information concerning the "floating-rotor capacitor" which was part of my article in the May/Jun 2002 issue of *QEX*. I have thought of a way to estimate the total change in capacitance of the floating-rotor tuning capacitor from out to in. I turned on the receiver normally and tuned in the 10-MHz WWV to zero beat with the receiver's IF oscillating. I then tuned off zero-beat to an approximately 1-kHz beat note. I measured (approximately) through what angle the floating rotor was turned to reach 1 kHz from zero beat at about 2 MHz IF. The angle was about 7°. I previously calculated the tuning inductance of that stage, so I was then able to calculate total capacitance at both zero-beat and at 1 kHz from zero beat. I then scaled the ΔC or change in capacitance for 7° rotation to a value of capacitance for 180° rotation, or full scale.

This assumes, of course that the relationship is linear. The estimated ΔC for 180° rotation is 1.5 pF. That's considerably less than can be explained by replacement of dielectric by metallic conductor as I suggested in my article. The floating-rotor capacitor continues to work well as a fine-tuning or band-spread capacitor in parallel with the main tuning capacitor, but I don't want readers to think that it produces a greater change in capacitance than it does. It may be possible to construct a capacitor of this type with more than one rotor and smaller clearances resulting in a greater change in capacitance, but I must say that I don't fully understand how the device works. It does work well in my application as described in *QEX*, however.—*Bill Young, WD5HOH, 343 Forrest Lake Dr, Seabrook, TX 77586;* **blyoung@hal-pc.org**

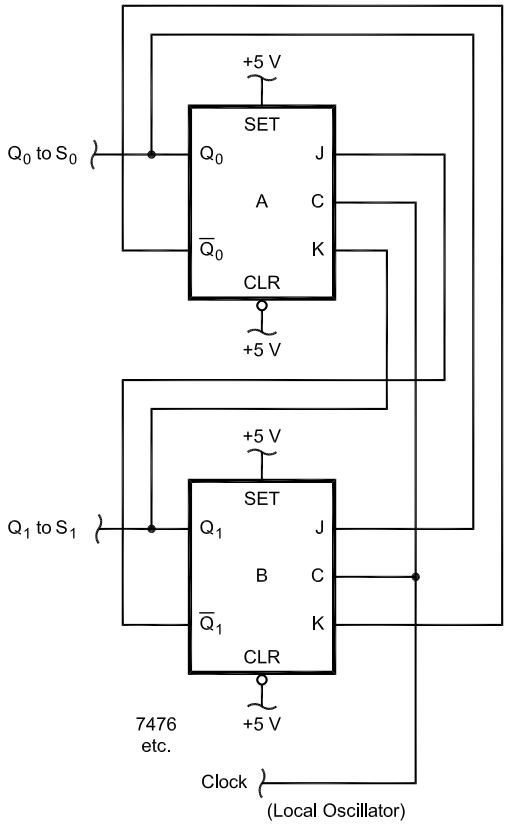
### On Quarter-Wave Transformers

I happened to see the letter from Lincoln Kraeuter concerning the quarter-wave transformer in the Jul/Aug 2002 *QEX*. If you're interested in further exploring the wave mechanics of this device, John Kraus has a very good explanation in his book *Electromagnetics* (pp 515-517). His explanation is based on partial reflections and partial transmissions of the voltage and current waves at the discontinuities of the transformer. Details are left out, unfortunately. I saw this several years ago and, to convince myself of Kraus's method, laboriously worked out the values of the various current and wave components. Thinking that this would be of interest to other hams, I wrote an article that another ham posted at his Web site. You can view the PDF at **home.attbi.com/~kb7qhc/**. The button at the lower left-hand corner of that page opens the PDF.—*Bill Klocko, N3WK, 701 Hillcrest Dr, Annapolis, MD 21401-4642;* **n3wk@arrl.net**

### The Dirodyne: A New Radio Architecture? (Jul/Aug 2002)

I enjoyed reading the article on the Dirodyne and the Tayloe detector. One thing bothered me from the start of the article. The two-bit counter is not a single-bit transition counter. Then as I read, I found out that noise was a problem with birdies. If the counter did not transition one bit at a time, I would expect that the spikes that occurred during the transitions would cause noise.

What do I mean by single-bit transitions? With a normal counter, the




| State | Clk | $Q_1$ | $Q_0$ | |
|---|---|---|---|---|
| 1 | 2 | 0 | 0 | 0° |
| 2 | 3 | 0 | 1 | 90° |
| 3 | 4 | 1 | 1 | 180° |
| 4 | 1 | 1 | 0 | 270° |

$$\overline{Q}_1\overline{Q}_0 \rightarrow R_{Q_1}S_{Q_0}$$
$$\overline{Q}_1 Q_0 \rightarrow S_{Q_1}S_{Q_0}$$
$$Q_1 Q_0 \rightarrow S_{Q_1}R_{Q_0}$$
$$Q_1\overline{Q}_0 \rightarrow R_{Q_1}R_{Q_0}$$

$R = K$
$S = J$
$R \Rightarrow$ RESET
S $\Rightarrow$ SET

$$R_{Q_1} = \overline{Q}_1\overline{Q}_0 + Q_1\overline{Q}_0 = \overline{Q}_0$$
$$S_{Q_1} = \overline{Q}_1 Q_0 + Q_1 Q_0 = Q_0$$
$$R_{Q_0} = Q_1 Q_0 + Q_1\overline{Q}_0 = Q_1$$
$$S_{Q_0} = \overline{Q}_1\overline{Q}_0 + \overline{Q}_1 Q_0 = \overline{Q}_1$$

**Fig 1—A schematic and support logic for a two-bit counter with single-bit transitions.**

binary output of the counter would be 00, 01, 10, 11, 00 ... (0, 1, 2, 3). Transitions between 00 and 01 or 10 and 11 would be a single-bit change. But transitions between 01 and 10 or 11 and 00 would be two-bit changes. Thus, the transitions could be 01, 11 10 or 11, 10, 00 or 11, 01, 00. As can be seen, the bit transitions could cause invalid states to be enabled (11 or 10 or 01) thus turning on the CMOS switches that you do not want at certain bit transitions. A single bit-transition counter would count 00, 01, 11, 10, 00; or 0, 1, 3, 2, 0; and only one bit changes at a time. A counter can be made from asynchronous logic that will be very fast (only two gate delays) and that will do single-bit transitions. The input would be the clock (local oscillator) and the output would be both the two-bit counter output ($Q_0$ and $Q_1$ for $S_0$ and $S_1$) and the four decoded outputs: CTL1, CTL2, CTL3, CTL4. See my Fig 1.—*Alvin P. Schmitt, KE4GVG, PO Box 10336, Blacksburg, VA, 24062-0336;* **schmitta@blacksburg.net**

***Dear Doug,***

I have no problem with Alvin's comments. In fact, they have me wondering!

I showed his idea regarding the two transitions to my work colleague (chief engineer) and he feels as do I that since *all* the transitions are synchronous from the 74AC161, that there are no spurious, asynchronous signals generated. Nevertheless, we could be wrong and I am happy to try a Gray-code counter to feed the Tayloe detector and simply reroute the audio paths. Of course, all of this takes time and I am currently working on a two-path system that is much quieter. It may be noteworthy that there are no dual transitions with this, which uses a standard quadrature generator using a Johnson counter straight out of *The ARRL Handbook*. As I mentioned in the article, there's plenty of work for whoever wants to take it on.—*Rod Green, VK6KRG, 106 Rosebery St, Bedford, Western Australia 6052;* **rodagreen@bigpond.com**

### Improved Dynamic-Range Testing (Jul/Aug 2002)

In addition to all the good SDR material, your dynamic-range article caught my attention. I recently [had a] discussion of similar ideas with Wes Hayward, W7ZOI. You don't reference Wes' correspondence ("More Thoughts on Receiver Performance Specification," *QST*, Nov 1979, pp 48-49) but it shares the concerns about MDS and associated bandwidth, solving the problem with "Receiver Factor." This is the receiver input IP3 (in dBm) minus the noise figure.

Unfortunately, there are errors in the noise discussion, making Eqs 5 and 6 wrong. The concept of "noise over relatively short time frames" just doesn't give the right answers. Bandwidth-limited noise has an auto-correlation function. Once enough time goes by for this function to essentially go to zero (a few inverse bandwidths), you are drawing a new random sample. The probability of its taking on any particular value is described by the probability density function (PDF). For the receiver voltages under consideration, there are no bounds on this. Likewise, the statement, "It is also equally likely to be small as large," is only valid if the PDF is uniform, which is quite far from the usual

situation for receiver voltages.

"In a receiver circuit, a noise voltage..." could have various meanings. Assuming it is the audio (or IF) output from a SSB receiver, the noise would have a Gaussian PDF. Since you are looking at this with an ac voltmeter consisting of a diode detector and a filter capacitor, this output signal is the one of interest. Your reference to absolute value agrees with this idea. The signal is approximated by its "envelope" and the PDF is Rayleigh (W. B. Davenport and W. L. Root, *An Introduction to the Theory of Random Signals and Noise*, McGraw-Hill, 1958, Section 8-5). The envelope PDF is very poorly approximated by a uniform distribution. Thus Eqs 5 and 6 don't work out.

The answers for Eqs 3 and 4 are correct for the case of a signal consisting of the absolute value of another [kind of] signal having zero-mean uniform PDF. I think this was the intention! However, the formulation of the problems is wrong and the intermediate math has errors.

The formulation in the first line of Eq 3 is for the case of a uniform distribution, but not the absolute value. The third line of Eq 3 has an error in sign, so the average should come out zero, not A/2 as shown. The answer is correct for the absolute value with a signal having a zero-mean uniform PDF. Eq 4 is formulated for the zero-mean uniform PDF and the answer is correct, as well as for the absolute value (they are the same). There is an error in the second line. [It] should have only one term.

Your point of using a true RMS voltmeter for measuring signal plus noise is excellent. However, at 10-dB of SNR if one does not make a correction for the noise, the estimate of signal is still too high by 10 log(1.1) ≈ 0.4 dB, which may or may not be acceptable. More importantly, if one measures the noise power with no signal present, the noise power can be subtracted from the measured S + N. Incidentally, if one wants to try to use the envelope-like voltmeter, the PDF of Gaussian noise plus a sine wave is the Ricean distribution (S. O. Rice, "Statistical Properties of a Sine-wave Plus Random Noise," *Bell Systems Technical Journal*, Jan 1948, also in Davenport and Root, above).

Doug, I hope this is useful and helps lead to good measurements for folks.—*Bob Larkin, W7PUA, 2982 NW Acacia Pl, Corvallis, OR 97330;* **boblark@proaxis.com**

***Dear Bob,***

You are right that the amplitude distribution of broadband thermal noise is not uniform but Gaussian. That means the simplifying assumption I made about noise amplitudes' being just as likely small as large is not strictly correct but only an approximation to measurements made on so-called peak-reading meters. Let me have a crack at it explaining it better.

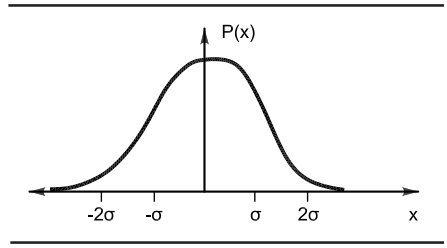



**Fig 2—The bell curve, or Gaussian probability-density function.**

The normalized amplitude distribution of thermal noise is:

$$P(x) = \frac{1}{\sqrt{2\pi}} e^{\left(\frac{-x^2}{2}\right)} \qquad \text{(Eq 1)}$$

This is the familiar probability density function (PDF) known as the "bell" curve, shown in Fig 2. I write normalized here because $x$ is normalized with respect to a standard deviation of $\sigma = 1$ and because the mean, $\mu$, is assumed to be zero. The probability that noise magnitudes lie in the range 0-A is found by integrating $P(x)$ over that range and doubling the result.

Most peak-reading audio voltmeters are more properly called quasi-peak-reading, because the charge and discharge times of the detector are chosen to match the ear's response to impulsive peaks. Standards for VU meters developed by NAB, DIN and IEC all use quasi-peak-reading techniques. Those meters don't produce readings that match the equation.

What happens when reading noise voltages from such a meter is that a certain averaging process takes place; but because of the detector and meter characteristics, it is a nonlinear process. The net effect is to make the noise look almost uniform. It can be shown that the part of the time broadband noise magnitude exceeds its own RMS ($\sigma$) value is only about 31.8% and that it exceeds $\sqrt{3}\sigma$ less than 10% of the time.

McClaning and Vito (*Radio Receiver Design*, Noble, 2000, p 460) indicate that the RMS-to-average ratio of broadband noise is 1.96 dB. My figure of 1.25 dB represents the difference produced by the meter.

The ARRL lab uses an RMS voltmeter for its measurements and I should make clear that they are not victims of the voltmeter problem that I tried to explain. I used an old Ballantine voltmeter and a Helper Instruments Sinadder for my measurements. When meters are bopping around, it is sometimes hard to average measurements; however, the Ballantine consistently indicated an 8-dB SNR when the Sinadder showed 10 dB. The 2-dB difference is evidently caused by the characteristics of the Ballantine. Other meters may produce different results. Nonetheless, a spectrum analyzer can pick discrete IMD products out of noise when nothing else can, providing a method for avoiding the vagaries of noise.

Finally, I acknowledge the intermediate errors in Eqs 3 and 4. They should read:

$$\begin{aligned} E_{AVG} &= \frac{1}{A}\int_0^A e\, de \\ &= \frac{1}{A}\left[\frac{e^2}{2}\right]_0^A \\ &= \frac{1}{A}\left[\frac{A^2}{2} - 0\right] \\ &= \frac{A}{2} \end{aligned} \qquad \text{(Eq 3)}$$

and

$$\begin{aligned} E_{MS} &= \frac{1}{A}\int_0^A e^2\, de \\ &= \frac{1}{A}\left[\frac{e^3}{3}\right]_0^A \\ &= \frac{A^2}{3} \\ E_{RMS} &= \frac{A}{\sqrt{3}} \end{aligned} \qquad \text{(Eq 4)}$$

*73, Doug Smith, KF6DX;* **kf6dx@arrl.org**

**In Search of Amateur Innovation**

If you experiment with radio as a ham, please read this through to the end and try to help me out. My name is Greg Lapin, N9GL, and I am a member of the FCC Technological Advisory Council. The TAC was formed by the Office of Engineering and Technology at the FCC to help keep the Commission informed about communication trends. It is made up of an erudite set of people, mostly from the communi-

cations industry. I've included a list of TAC members and their titles at the end of this letter.

I've been a ham since 1970. I credit our hobby with putting me on the path to a PhD degree in electrical engineering and for much of the work that I do as a consultant. I have found that people in industry have had good experiences working with hams and generally trust engineers from our hobby to provide innovative engineering services.

Unfortunately, that view is not always held at the higher levels in companies. I see a trend at places like FCC TAC meetings where suggestions are made that the spectrum that is assigned to Amateur Radio would be better allocated in other ways. As one corporate CTO put it to me regarding complaints from amateurs that their weak-signal work was being interfered with by Part-15, 2.4-GHz wireless networking, "The public good is better served by allowing millions of people to connect to the Internet from anywhere rather than by a few amateurs playing around."

Fortunately, the FCC has not bought this argument so far. However, the pressure on them is intense. The ARRL is already making admirable attempts at convincing Congress and the FCC that Amateur Radio is a valuable commodity, but most of their arguments seem to be based on the use of Amateur Radio to provide necessary communications in emergencies.

The word amateur often serves as an impediment to having what is done on the ham bands taken seriously. When I looked in a dictionary for the word amateur, there were two conflicting definitions that explain attitudes about our service:

> "amateur: 1) one who pursues an activity or is devoted to a study purely for intrinsic reward rather and monetary gain; 2) one who is unskilled in a given area or activity."

Many people in the communications industry seem to identify our avocation with the second definition. I work as a consultant in the communications industry and in my experience working with professionals, the members of our avocation who can be described with the first definition are usually more capable than many of the people who develop communications techniques for a living. Often, the best professionals in the companies that I have worked for hold Amateur Radio licenses.

There is another very important aspect to Amateur Radio that does not get the press it deserves. Experimentation with new communication techniques by amateurs over the course of the history of radio has provided the basis of much of what we consider to be modern communications. When I started to compile a list to report at a TAC meeting, there were very few details that anyone could give me. In the broadest terms, these were the developments in communications in which hams had an integral part: experimentation with ionospheric propagation in the early days of radio, early development of the use of frequencies above HF, early development of mobile radio equipment, initial experimentation with SSB, various antenna designs, the first non-military communications satellite, low-Earth-orbit satellites, experience with linked repeater networks that affected the design of cellular telephone networks, packet radio as the first wireless digital networks, early experimentation with digital signal processing and exploration of new modes of VHF propagation.

You may notice that much of the innovation by amateurs in that list is decades old. The question asked by many in the communications industry is, "What has Amateur Radio done for us lately?"

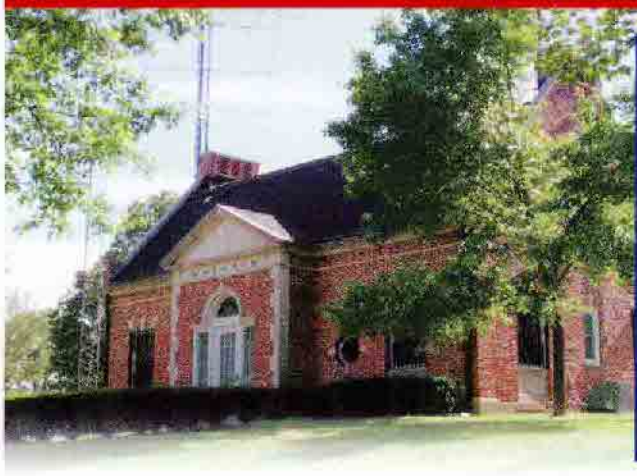
Thus, the point of my appeal, I don't believe that the Amateur Radio community has been effective at letting the rest of the technical world know the answer to that question. We all suspect that a lot of experimentation is taking place on the ham bands, but we don't hear of it often enough. I'd like to change that. Let me know about what you are working on. How is existence of the Amateur Radio Service helping to make your investigations possible? I'd like to report on your work; even it if never amounts to anything. If you try something that fails, it is still important because it adds to our knowledge of what is and isn't feasible. The big company that gets recognition for developing new communications techniques may have been able to do so only because of the experience that was gained by radio amateurs. Let's tell the world about our part in it!

I'd like to compile your experiences in experimentation on the amateur bands for two reasons: to report them to bodies such as the FCC TAC and to report them to the amateur community. Please get in touch with me by e-mail at **g.lapin@ieee.org** with your stories and experiences.—*Gregory D. Lapin, PhD, PE, N9GL;* **g.lapin@ieee.org**; *Chairman, ARRL RF Safety Committee;* **www.arrl.org/rfsafety**; *Member, FCC Technological Advisory Council,* **www.fcc.gov/oet/tac**

## *Next Issue in* QEX/Communications Quarterly

Are you getting what you expect from your guyed antenna installation? In the next issue, Dick Weber, K5IU, explores the interactions between his antennas and his guy wires. Dick uses modeling software to predict behavior and compares his predictions with actual measurements—how neat! The exercise reveals certain pitfalls and sets limits on the length and proximity of conductive guy sections.

# The dream is ALIVE...



**The dream began with a 1936 flood...**

... and the 1937 purchase of seven acres of land in Connecticut...

... then the inaugural CQ in July 1938...

... through the radio silence of World War II...

... and post-war code practice sessions...

... to 21st century technology.

**The traditions and dreams of Amateur Radio are alive at W1AW every day!**

**You can ensure that the Hiram Percy Maxim Memorial Station will be on the air for the next generation!**

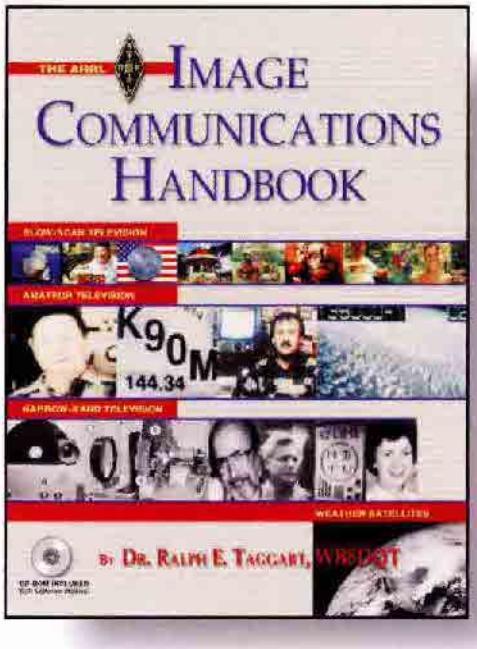
**Send your contribution to the W1AW Endowment Fund today!**

For more information, contact
**Mary M. Hobart, K1MMH**
Chief Development Officer
**ARRL**
225 Main Street
Newington CT 06111-1494
**Telephone: 860-594-0397**
**Email: mhobart@arrl.org**
**Web: www.arrl.org/endoww1aw.html**

**Now Shipping!**

The ARRL
# IMAGE
COMMUNICATIONS
HANDBOOK

Explore the possibilities of using Amateur Radio to **see and talk with other hams!** With home computers, widely available software, and gear many hams already own, it's easier than ever to enjoy the imaging modes: Narrow-Band Television (NBTV), Amateur Television (ATV), Slow-Scan Television (SSTV), and Weather Satellite Imaging (WEFAX).




***The ARRL Image Communications Handbook***

**by Dr. Ralph E. Taggart, WB8DQT**

Book includes CD-ROM with software utilities.

**ARRL Order No. 8616—$25.95***

*shipping: $6 US (UPS)
$8.00 International (surface)

Sales tax is required for orders shipped to CA, CT, VA, and Canada.

**Available from ARRL Dealers EVERYWHERE.**

**YOU'RE....**
**ON THE AIR**

**ARRL** *The national association for AMATEUR RADIO*
**www.arrl.org/shop**

225 Main Street, Newington, CT 06111-1494 **tel: 860-594-0355 fax: 860-594-0303**
In the US call our toll-free number **1-888-277-5289** 8 AM-8 PM Eastern time Mon-Fri.

# Great Audio!


MADE IN USA

**800-833-7373**
**www.tentec.com**

**$1189**

**jupiter video available**
All about this innovative rig!
30 min.—$10.00
(refunded with Jupiter purchase)

Great sound on the air can be yours at the turn of a knob with JUPITER. 18 selectable SSB transmit bandwidths to a maximum of 3.9 kHz deliver the finest sounding audio in amateur radio. Connect your favorite microphone, pick a bandwidth that suits you and listen to the comments roll in. Whether you're DXing on the high bands or ragchewing on 75, JUPITER can make you the best sounding op on the band. Find out why thousands of other hams have recently bought Ten-Tec—call or email us today for a complete information package.

**Model 307C External Speaker**
**$98.00**

**302J Remote Encoder/Keypad**
**$139.00**

**705 Desk Microphone**
**$99.95**

**963 Switching Power Supply**
**$169.00**

0Model 701, Accessory Hand Mic, not shown ($28)
538AT, Internal Auto Antenna Tuner, not shown ($299)

TEN-TEC

1185 Dolly Parton Parkway
Sevierville, TN 37862
Sales Dept: 800-833-7373
Sales Dept: sales@tentec.com
Monday - Friday 8:00 - 5:30 EST
*We accept VISA, Mastercard, Discover, and American Express*

Office: (865) 453-7172 • FAX: (865) 428-4483
Repair Dept.: (865) 428-0364 (8 - 5 EST)

S&H cost for Jupiter in 48 states is $16. With Power Supply, $21.