PerlAPRS

An Automated Control Application for APRS Networks

Richard Parry, P.E., W9IF
rparry@qualcomm.com
http://people.qualcomm.com/rparry

ABSTRACT

PerlAPRS is an application which can monitor both local TNC received APRS packets and remote Internet APRS packets and perform an automated action based on criteria specified by the user. The criteria that PerlAPRS uses is the callsign of the station and its location specified as a Maidenhead Grid Square. Other requirements specified by the user increase functionality of the program in real world applications. The actions executed can be written in any language, but UNIX style shell scripts are ideally suited for this purpose. Scripts can be developed to perform functions such as automatic notification via email as well as logging. PerlAPRS is freely distributed under the GNU licensing agreement.

KEYWORDS

Packet Radio, APRS, Linux, UNIX, Perl

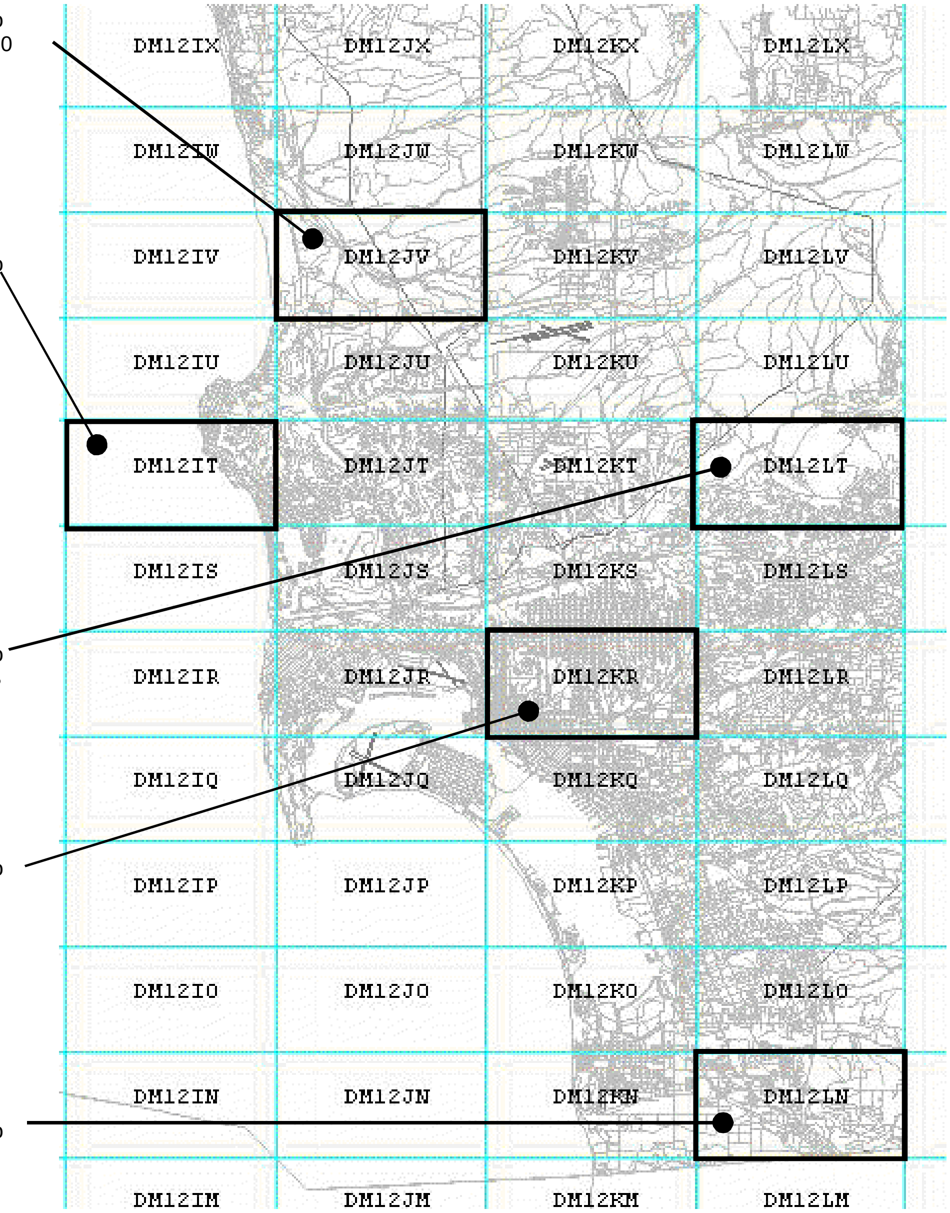
INTRODUCTION

The Automatic Position Reporting System[1] is one of the most popular facets of amateur radio today. It is a marriage of several cutting edge technologies including the Global Positioning System (GPS), amateur packet radio, and the global Internet. It incorporates satellite technology, wireless networks, and both analog and digital communication. The applications to support the APRS protocol are also sophisticated. They provide the user with an easy to use interface into the APRS world. Software such as MacAPRS for the Apple Macintosh, WinAPRS for Windows 95/NT, and APRSdos for DOS machines provide powerful and elegant solutions. With this software and support system, it is possible to display any APRS network. Other support software for APRS includes the work of Steve Dimise, K4HG, who extended the concept for the promulgation of APRS packets to the Internet with javAPRS. In addition, Steve Boyle, KD6WXD; and Dale Heatherington, WA4DSY, developed APRS servers for the Internet which allow users to remotely connect to the server and examine remote APRS networks.

These programs provide flexibility, functionality, and a highly visual means for tracking APRS activity. However, they are passive in that they provide predominately monitoring functionality. They do not provide the ability to control. For example, if you wish to know when an APRS tracker escorting marathon runners reaches a specific location, you need more than monitoring capability, you need control functionality. It is this ability that PerlAPRS provides.

---

[1] The APRS formats are provided for use in the amateur radio service. Hams are encouraged to apply the APRS formats in the transmission of position, weather, and status packets. However, APRS is a registered trademark of Bob Bruninga who reserves the ownership of these protocols for exclusive commercial application and for all reception and plotting applications. Other software engineers desiring to include APRS protocols in their software for sale within or outside of the amateur community will require a license from him.



Execute cmd2.sh up to 2 times when KI6MP-10 is heard in DM12JV and reset every 24 hours

Execute cmd1.sh up to 3 times when KC6VVT-9 is heard in DM12IT and reset every 24 hours

Execute cmd4.sh up to 5 times when KE6PHB is heard in DM12LT and reset every hour

Execute cmd3.sh up to 3 times when KD6AZU is heard in DM12KR and reset every 3 hours

Execute cmd5.sh up to 2 times when any station is heard in DM12LN and reset every hour

Figure 1 San Diego, California

PerlAPRS is a program written in the Perl computer language. Perl runs on all popular computer platforms today including MacOS, Windows3.1/95/NT, AMIGA, Unix, Linux, and many more. In addition, since Perl is compiled at run time, there is no need for a version precompiled or packaged for a specific platform. Also, since all source code is included for PerlAPRS, the user may easily alter the program to meet specific needs and is encouraged to do so. However, most users will find modifying the shell scripts rather than the program should meet most requirements.

PerlAPRS examines incoming packets from an APRS network and executes commands when a callsign and location match the criteria specified by the user. Location criteria is specified using grid squares. For example, when KC5PVL enters grid square DM12LW, a computer command specified by the user can be automatically executed.

Figure 1 shows grid squares overlaid on the city of San Diego, California, it will serve as the basis for the examples in this paper. The map shows several grid squares that are targeted for an action when the criteria specified in the *callsign.dat* file is met.

THE CALLSIGN FILE

When PerlAPRS starts, it reads the user’s callsign database file with the default filename of *callsign.dat*. This file provides the list of callsigns that PerlAPRS is to search for. The file consists of one or more lines of text as shown in Figure 2. A separate line (record) is required for each callsign. Each line of the file is further broken into five fields, a separate field for each parameter.

```
KI6MP-10     cmd2.sh       DM12JV       2            1440
KC6VVT-9     cmd1.sh       DM12IT       3            1440
KD6AZU       cmd3.sh       DM12KR       3            180
KE6PHB       cmd4.sh       DM12LT       5            60
*            cmd5.sh       DM12LN       2            60
```

Figure 2 Example Callsign File

The first field indicates the callsign that PerlAPRS is to listen for. In the example, PerlAPRS will listen for KI6MP-10, along with KC6VVT-9, KD6AZU, and KE6PHB. The asterisk character, shown on the last line of the example, is a wildcard that means “any” callsign.

The second field indicates the command that will be executed when the callsign is heard. It can be any computer command, however, as we will discuss later, shell scripts are powerful and easily implement commands. In the example, *cmd2.sh* will be executed when KI6MP-10 is heard.

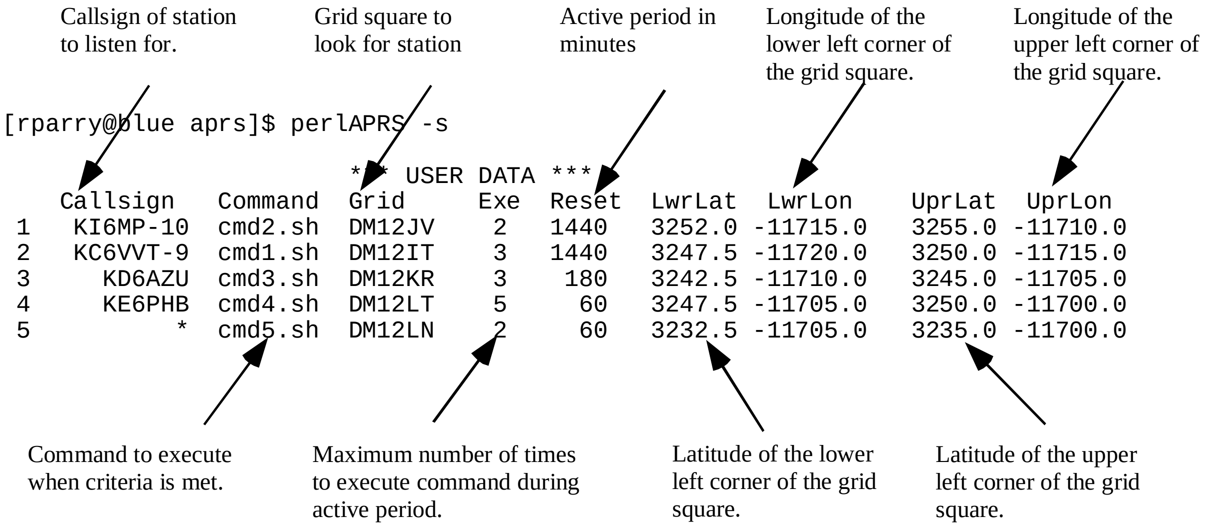
The third field represents the grid square in which the callsign must be heard. Returning to the example, PerlAPRS is listening for KI6MP-10 in grid square DM12JV.

The fourth field is provided to limit the number of times the command is executed during an active period. This parameter is necessitated by the repetitive nature of APRS packets. For example, if a station continues to broadcast packets while located within the grid square, the command would be executed each time a packet is heard. Since some APRS packets (e.g., mobile) are transmitted every few minutes, in many cases it would be undesirable to have the command executed repetitively in a short period. For this reason, the value is typically a small number (e.g., 1-5). However, indicating a large value will cause the command to be executed virtually without limit. Conversely, if one wishes to disable execution, setting the value to zero essentially disables the command without removing it from the database. In the example, the command, *cmd2.sh*, will be executed no more than 2 times during an active period.

The last field is provided to allow the user to specify the active period. The active period is the time expressed in minutes in which PerlAPRS is actively listening for the specified callsign. This is important, since without a means of resetting the execution counter, it would be inconvenient to leave the program running for an extended period (e.g., many weeks).

The following scenario may help to illustrate the need for specifying the active period. Assume we wish to leave PerlAPRS running indefinitely. Also assume we don’t want to execute a command every time a packet is heard since this could be hundreds of time during a 24 hour period. If we set the execution counter to a small value, we will limit the number of times the command is executed. However, once that count is reached, commands will no longer be executed. The active period parameter is therefore provided to allow the user to specify when the execution counter specified in field 4 is to be reset. Returning to our example, we see that a command will be executed no more than 2 times in a 24 hour (1440 minutes) period for KI6MP-10. At the end of the 24 hour period, the counter is reset and the command can again be executed up to 2 times during the next active period.

When PerlAPRS begins it reads in the callsign file and shows the original information provided by the user along with the conversion of the grid square to latitude and longitude[2]. To completely describe the grid square, requires the latitude and longitude of the lower left and upper right points of the square. These corner points are used by PerlAPRS to determine if the station is within the grid square.



```
[rparry@blue aprs]$ perlAPRS -s

                        *** USER DATA ***
    Callsign   Command  Grid     Exe  Reset  LwrLat  LwrLon    UprLat  UprLon
 1   KI6MP-10  cmd2.sh  DM12JV    2   1440   3252.0 -11715.0   3255.0 -11710.0
 2   KC6VVT-9  cmd1.sh  DM12IT    3   1440   3247.5 -11720.0   3250.0 -11715.0
 3     KD6AZU  cmd3.sh  DM12KR    3    180   3242.5 -11710.0   3245.0 -11705.0
 4     KE6PHB  cmd4.sh  DM12LT    5     60   3247.5 -11705.0   3250.0 -11700.0
 5          *  cmd5.sh  DM12LN    2     60   3232.5 -11705.0   3235.0 -11700.0
```

Callsign of station to listen for.
Grid square to look for station
Active period in minutes
Longitude of the lower left corner of the grid square.
Longitude of the upper left corner of the grid square.
Command to execute when criteria is met.
Maximum number of times to execute command during active period.
Latitude of the lower left corner of the grid square.
Latitude of the upper left corner of the grid square.

Figure 3 Output from PerlAPRS based on user's callsign file

SHELL SCRIPTS

When an APRS packet is heard, the command specified by the user is executed. It is important to emphasize that virtually any command can be executed, one is not limited to shell scripts. However, they are simple to write, flexible, and powerful. Scripts should meet the needs of most users. Alternatively, one can use Perl scripts which are even more powerful, yet still easy to write.

How to write shell scripts is a subject all by itself, and for this reason only a few examples are provided here. You don't have to be a software engineer to write scripts, but a knowledge of UNIX commands is important. The examples below were developed on a Linux system. These commands are not expected to work on other systems without customization. They are provided here as examples for illustrative purposes.

The following shell script will make a sound by sending the audio file *chirp.wav* to the audio output port of the system.

```
#!/bin/bash
cat /sounds/chirp.wav > /dev/audio
```

Shown below is a simple shell script to send email.

```
#!/bin/bash
echo “Match for KK5SU” | mail rparry@qualcomm.com -s “perlAPRS notification”
```

---

[2] There is a common misunderstanding that for APRS applications latitude and longitude is specified as: degrees, minutes, and seconds when it should be: degrees, minutes, and decimal minutes. For example, 100 degrees, 40 minutes, and 30 seconds is written as 10040.500 and not 10040.30.

STARTING PerlAPRS

PerlAPRS is invoked from the command line like most UNIX style commands. Note that in the examples below, several command line arguments may be passed to the program which allow the user to alter PerlAPRS defaults.

`perlAPRS -h`
`perlAPRS -help`
	Display a *help* screen.

`perlAPRS -v`
`perlAPRS -version`
	Display the current *version* of the software.

`perlAPRS -s`
`perlAPRS -show`
	Normally PerlAPRS will not provide any output. The “show’ option allows the user to see the progress of PerlAPRS. The *show* option will display only valid APRS packets that contain a position (latitude and longitude).

`perlAPRS -d`
`perlAPRS -debug`
	The *debug* option forces PerlAPRS to display packets that do not contain a valid position. This option is primarily for program debugging purposes.

`perlAPRS -d -s`	This example shows how PerlAPRS can be made to display both packets with and without latitude and longitude information.

`perlAPRS -p /dev/cua2`
`perlAPRS -port /dev/cua2`
	The default *port* that PerlAPRS will open is “/dev/cua1”. However, you can specify an alternate serial port using this option. For this example, a Linux serial port name is indicated. For other platforms, consult the system’s documentation.

`perlAPRS -p www.wa4dsy.radio.org:14579`
`perlAPRS -p sboyle.slip.netcom.com:14579`
`perlAPRS -p www.ne1h.radio.org:14579`
	If an Internet address is specified, PerlAPRS will open a socket to the address and obtain TNC data from the Internet. The examples listed are three presently know APRS servers. The text preceding the colon is the host name. The number following the colon is the port number which is fixed at 14579 for APRS applications

`perlAPRS -p trip.tnc`	A third specification for the *port* option is not actually a port, but a data file. If you have saved raw TNC packets to a text file, PerlAPRS will open the file rather than opening a serial port or an Internet communication’s socket. This last variation of the port command is included for completeness, its main purpose is to allow PerlAPRS testing using known packets.
`perlAPRS -f callsign2.dat`	If a command *file* is not specified on the command line, PerlAPRS will use the default filename *callsign.dat*. However, as shown in the example, the user can force an alternate command file to be called using the -f option.
`perlAPRS &`	This example shows how most users will run PerlAPRS. In this example, PerlAPRS will not show any output. It is also run as a background process by adding the ampersand character (&).

HOW IT WORKS

Figure 4 shows a sample output from PerlAPRS with the “-s” (show) option on. When a packet arrives, the callsign of the originating station is extracted along with the station’s latitude and longitude. This information is then compared with each callsign in the database created by the user. If a match is found, the command is executed.

The first line in the example below shows a packet from originating station KD6AZU. PerlAPRS extracts the callsign, latitude, and longitude, and displays them on the line following the packet. Since this is a valid APRS posit packet, PerlAPRS will search the callsign database looking for a match for KD6AZU. As shown in the example, the first two attempts at a match fail. The third comparison is a match shown in bold print for illustrative purposes. The command, *cmd3.sh*, is then executed and the execution counter is incremented. Note that the first line of the match included the time that the packet was heard along with the maximum execution count specified by the user (e.g., 3). The second line of the match shows the time that the execution counter will be reset, the present value of the counter (e.g., 1) and the name of the execution command. When a second and third APRS packet arrives from KD6AZU, a match occurs and the command is executed again. When the fourth packet arrives, the command is not execution since the maximum execution count has been reached. No additional matches for KD6AZU will cause command execution. However, by the time the fifth packet arrives, the execution counter has been reset by the timeout and command is executed again.

```
Packet= KD6AZU>APRS,KD4DLT-7,N4NEQ-2,WIDE*:@042327/3243.70N/11707.70W/0
             KD6AZU       3243.700    11707.700
        -  KI6MP-10       DM12JV
        -  KC6VVT-9       DM12IT
        *    KD6AZU       DM12KR Sun Aug 10 15:56:13 1997    3
                                 Sun Aug 10 15:57:13 1997    1      cmd3.sh
        -    KE6PHB       DM12LT
        -         *       DM12LN
Packet= KD6AZU>APRS,KD4DLT-7,N4NEQ-2,WIDE*:@042327/3243.70N/11707.70W/0
             KD6AZU       3243.700    11707.700
        -  KI6MP-10       DM12JV
        -  KC6VVT-9       DM12IT
        *    KD6AZU       DM12KR Sun Aug 10 15:56:23 1997    3
                                 Sun Aug 10 15:57:13 1997    2      cmd3.sh
        -    KE6PHB       DM12LT
        -         *       DM12LN
Packet= KD6AZU>APRS,KD4DLT-7,N4NEQ-2,WIDE*:@042327/3243.70N/11707.70W/0
             KD6AZU       3243.700    11707.700
        -  KI6MP-10       DM12JV
        -  KC6VVT-9       DM12IT
        *    KD6AZU       DM12KR Sun Aug 10 15:56:34 1997    3
                                 Sun Aug 10 15:57:13 1997    3      cmd3.sh
        -    KE6PHB       DM12LT
        -         *       DM12LN
Packet= KD6AZU>APRS,KD4DLT-7,N4NEQ-2,WIDE*:@042327/3243.70N/11707.70W/0
             KD6AZU       3243.700    11707.700
        -  KI6MP-10       DM12JV
        -  KC6VVT-9       DM12IT
        *    KD6AZU       DM12KR Sun Aug 10 15:56:44 1997    3
                                 Sun Aug 10 15:57:13 1997    3      cmd3.sh
        -    KE6PHB       DM12LT
        -         *       DM12LN
Packet= KD6AZU>APRS,KD4DLT-7,N4NEQ-2,WIDE*:@042327/3243.70N/11707.70W/0
             KD6AZU       3243.700    11707.700
        -  KI6MP-10       DM12JV
        -  KC6VVT-9       DM12IT
        *    KD6AZU       DM12KR Sun Aug 10 15:57:14 1997    3
                                 Sun Aug 10 15:58:14 1997    1      cmd3.sh
        -    KE6PHB       DM12LT
        -         *       DM12LN
```

Figure 4 Output from PerlAPRS

MAIDENHEAD GRIDS[3]

PerlAPRS relies on the use of grid squares to specify location. It accepts grid square parameters in 2, 4, or 6 letter formats. A 2 letter grid square covers such a large geographic area that the entire United States can be described in only 8 grid squares. A four letter grid square is smaller, but still represents a very large area (approximately the size of Connecticut). The 6 letter grid square is much smaller and is well suited for many applications in metropolitan areas.

It is not possible to describe the area of a grid square for a given format (e.g., 2, 4, or 6 letters) since they vary with their location on the earth. To illustrate the point, Figure 5 shows the size of a 2 and 4 letter grid square for the northern and southern portions of the United States. Even larger variations in grid square size occur between the poles and the equator.

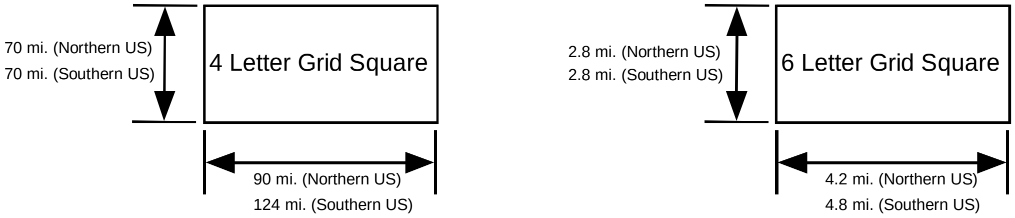


Figure 5 Grid Square Comparisons

Note that only the longitudinal distance varies between the southern and northern portion of the United States. The reason for this apparent anomaly stems from the fact that lines of latitude are parallel to each other and therefore separated by a constant distance. Lines of longitude are not parallel, they meet at the poles and are farthest apart at the equator.

CONCLUSION

PerlAPRS was developed to expand the usefulness of APRS to automated unmanned operations. It is an application that should prove useful in circumstances that require a specific action to a predefined packet specification. Using shell scripts or other languages developed by the user in conjunction with PerlAPRS should provide the framework for developing and extending APRS to many unique applications.

ACKNOWLEDGMENT

Thanks to Bob Bruninga, WA4APR; for permission to use the APRS trademark. Special thanks also to Keith Sproul, WU2Z; Mark Sproul, KB2ICI; and Steve Dimise, K4HG; for their pioneering work in this area.

---

[3] See ARRL web page listed in the bibliography for a more detailed explanation of this topic.

SYSTEM REQUIREMENTS

Linux was the development platform for PerlAPRS, however, it will work on any platform that supports perl version 5.002 or later. In addition, since Perl is available on virtually every popular computer platform, PerlAPRS should be able to be implemented easily. Any limitations are more likely to be with the platform's ability to support shell scripts. However, as emphasized in this paper, any computer language can be used to developing commands.

DISTRIBUTION

PerlAPRS is a freeware program available under the GNU General Public License, Version 2, June 1991, Free Software Foundation, Inc. 625 Massachusetts Avenue, Cambridge, MA 02139. It may be downloaded from the author's home pages at: **http://people.qualcomm.com/rparry/perlAPRS**. Further information regarding the necessary files to download and system requirements are included there.

BIBLIOGRAPHY

1. Bruninga, Bob, "Automatic Packet Reporting System (APRS)," *73*, December 1996, pp. 10-19.
2. Dimse, Steve, "javAPRS: Implementation of the APRS Protocols in Java," *ARRL and TAPR 15th Digital Communications Conference Proceedings*, Seattle, Washington, September 1996, pp. 9-14.
3. Ford, Steve, "Where Am I," *QST*, April 1994, pp. 86-88.
4. Horzepa, Stan, "APRS Tracks: RELAY, WIDE, and Other Paths," *Packet Status Register*, Fall 1996 - Issue #64, pp. 30-31.
5. Horzepa, Stan, "APRS Tracks: Alias Envy," *Packet Status Register*, Summer 1996 - Issue #63, pp. 25-26.
6. Horzepa, Stan, "APRS Tracks," *Packet Status Register*, Spring 1996 - Issue #62, pp. 16-17.
7. Horzepa, Stan, "Getting On Track with APRS," American Radio Relay League, Newington, CT.
8. Parry, Richard, "Position Reporting with APRS," *QST*, June 1997, pp 60-63.
9. Wilson, Mark, "QST Compares: GPS-Compatible TNCs," *QST*, October 1995, pp. 68-71.

WEB SOURCES

1. The ARRL web page **http://www.arrl.org/locate/gridinfo.html** provides an excellent source of information on Maidenhead grid squares. The page also allows one to interactively determine a gird square from the latitude and longitude provided by the user.
2. To join the APRS mailing list, send email to **listserv@tapr.org** with **subscribe aprssig FirstName LastName** in the body of the message.