The Need for Standard Application-Programming Interfaces (APIs) in Amateur Radio

Are you tired of requesting your favorite ham software developer to support your favorite rig or device?

By Lawrence G. Dobranski, VA3LGD/VE3TVV

The Problem

Some recent discussions on Internet e-mail reflectors supporting some of the popular contesting and logging software reinforced the lack of programming standards within the Amateur Radio community. When a new radio or other product is released, this lack of standardization causes developers to scramble; they must modify their programs to interface to the new equipment. Many software developers do not provide Amateur Radio software as their primary occupation. Finding the specifications, getting access to the equipment and verifying the interfaces can be a continual and significant hardship.

14 Sandhead Terrace
Nepean, Ontario, K2J 1L4
va3lgd@amsat.org

The Solutions

Two solutions exist for this problem. The first is to have ham-radio equipment manufacturers develop standard interfaces and command sets. Given the ham-radio equipment manufacturers' inability to agree on the wiring of the microphone connector, the possibility of getting a standard command set developed, approved and used is unlikely.

The second solution is based on a similar problem that has already been solved in the software-development world. Today's operating systems provide many features to application developers. The features and services are usually accessed through an *application-programming interface* (API). APIs provide a set of function calls that application developers use. The software that implements these calls performs the lower-level functions of the device or operating system.

What is an API?

To understand what an API is, let's review how one works through an example. Most contesting and logging programs use the computer interface on Amateur Radio transceivers. Through this interface, they read and set frequency, band and other information. To set the radio's frequency, the program must convert the data to a numerical format the radio can understand, format the data within an appropriate command structure and send the result to the radio. [Acknow-

ledgment from the transceiver may also be required.—*Ed.*] Similar operations are performed to set or read other rig data. Today, these programs must support many different rigs from many different manufacturers, each with its own command set and data format. The protocols may be quite different, and the command sets mutually exclusive.

Instead of the application software composing the radio command directly, it might call a standard software function instead—HAM_API_Rig_SetFrequency(*x*)—where *x* is the desired frequency. This API call is translated by a radio-specific library into the radio's command format. When new radios are released, a new radio-specific library is developed, rather than modifying the application software. With the addition or update of the new library, all existing applications that use the HAM API would then be able to interface to the radio.

Where in Amateur Radio Would We Use Them?

Amateur Radio is becoming increasingly computerized. In many of our shacks, computers are interfaced to our rigs, TNCs, rotators, voice keyers, CW keyers, antenna switches, GPSs, etc. When software is developed to aid the amateur, it must be built to support specific equipment. If Amateur Radio APIs existed, then we would only need to develop specific libraries. The application would no longer have to be modified.

Table 1 lists a sample of the API functions that could be developed for amateur use.

Two Sample APIs

To see the effectiveness of APIs, Tables 2 and 3 define samples that might be used for rig control and interfacing to a PacketCluster. The style used in the definition depicts the API as a set of functions. It could be defined in terms of object-oriented-programming constructs as well.

How Do We Develop Them?

If this approach to computer control of amateur equipment is acceptable, interested amateurs must develop working groups to author the respective APIs. These working groups could discuss their development—using the Internet, for example—and author the various libraries. Once an API is developed, it would not be considered a reference standard until two unrelated applications use the API to control two different devices. A test suite is then developed to verify that future API implementations meet the standard. This conformance test ensures the user that the API implementation will work with their application.

Before API standard development begins, a standard naming convention for Amateur Radio APIs should be developed. For example, a proposed naming convention is as follows:

HAM_API_*xxx*_*yyyy*(). *xxx* is the API name (ie, rig, tnc, rotor) and *yyyy* is the function name. Variables and constants are named in a similar way.

How Do We Implement Them?

Linux

Linux and all UNIX derivatives support run-time libraries. A chapter in *Linux Application Development*[1] describes *Linux's* shared libraries and how to implement them. Each Ham API device-specific library would be implemented as a shared library.

Windows

Win16 (the formal name for the Win 3.1X environment) and Win32 (Win 9X and NT) provide support for run-time libraries. In the Windows environment, these libraries are called Dynamic Link Libraries (DLLs). Their file name extension is ".dll". A good portion of the Windows operating system is implemented in DLLs. Each Ham API device-specific library would be implemented as a DLL.

DOS

The Microsoft *DOS* operating environment presents an interesting challenge when trying to implement

[1] *Linux Application Development*, by Michael K. Johnson and Erik W. Troan, published by Addison Wesley Longman Inc, 1998.

Table 1
Proposed API Classifications

Amplifier Control
Antenna Switch
Call book interfaces
CW Contest Keyers
Digital Voice Keyers
GPS Data
PacketCluster
Rig Control
Rotor Control
Satellite Trackers
TNC Control

Table 2
A Sample API for Rig Control

Function	*Description*
HAM_API_Rig_getName()	Returns the rig name and model number
HAM_API_Rig_getCaps()	Returns—in a standard data structure—information about the rig's capabilities: mode, frequency range, output power, etc
HAM_API_Rig_selectRig()	Sets the active rig for subsequent commands. Use when more than one rig is controlled by the computer
HAM_API_Rig_getSettings()	Returns—in a standard data structure—current rig settings: frequency, mode, split, etc
HAM_API_Rig_getFrequency()	Returns the rig frequencies
HAM_API_Rig_setFrequency()	Sets the rig frequency
HAM_API_Rig_setEventFunction()	Sets the function to be executed if a rig generated event happens, ie, frequency changed from rig's front panel
HAM_API_Rig_getEvent()	Returns the event that caused the setEventFunction to be activated
HAM_API_Rig_setRIT()	Sets the receive incremental tuning (RIT)
HAM_API_Rig_setXIT()	Sets the transmit incremental tuning (XIT)
HAM_API_Rig_setMode()	Sets the rig's mode
HAM_API_Rig_getMode()	Get the rig's mode

Table 3
A Sample API for Interfacing to a PacketCluster

Functional	*Description*
HAM_API_Packet_Cluster_login()	Login in to the PacketCluster
HAM_API_Packet_Cluster_setName()	Set the operator's name
HAM_API_Packet_Cluster_setQTH()	Set the operator's QTH
HAM_API_Packet_Cluster_doSetCommand()	Sends a set command to the PacketCluster. The command is contained in a standard data structure that also contains the result of the command
HAM_API_Packet_Cluster_doDirCommand()	Sends a Dir command to the PacketCluster. The command is contained in a standard data structure that also contains the result of the command
HAM_API_Packet_Cluster_doShowCommand()	Sends a Show command to the PacketCluster. The command is contained in a standard data structure that also contains the result of the command
HAM_API_Packet_Cluster_delete()	Sends a command to delete a mail message
HAM_API_Packet_Cluster_send()	Sends a mail message
HAM_API_Packet_Cluster_announce()	Sends an announcement. The type of announcement is contained in standard data structure
HAM_API_Packet_Cluster_quit()	Sends the command to log off the node
HAM_API_Packet_Cluster_dx()	Announces a DX station
HAM_API_Packet_Cluster_reply()	Reply to a read message
HAM_API_Packet_Cluster_talk()	Enter talk mode
HAM_API_Packet_Cluster_type()	Enter a command to display a file. File contents are returned in the data structure
HAM_API_Packet_Cluster_upload()	Uploads a bulletin file
HAM_API_Packet_Cluster_wwv()	Gets the solar flux
HAM_API_Packet_Cluster_read()	Sends a command to read a message into the data structure.

standard libraries supporting the API. No one standard run-time-library module has emerged. Standard linking libraries are used at compile and link time, but often, no real run-time library module exists. Instead, software developers have made use of the architecture of the Intel iAPX-86 family, for which *DOS* was developed. They use the same method by which *DOS* communicates with underlying basic input/output system (BIOS) firmware and software; that is, via software interrupts and terminate-and-stay-resident (TSR) techniques.

The Intel iAPX-86 architecture provides support for up to 256 software interrupts. Like their close cousins, hardware interrupts, software interrupts are invoked by asserting an interrupt request (IRQ). Instead of being generated via hardware, software interrupts are requested through software instructions. For example, *DOS* provides a function for printing a character to the standard output device. It is invoked by the following fragment of iAPX-86 assembly language code:

```
mov   al,32   ; move an ASCII 32 (space)
              ; Into the <AL> register
mov   al,dl   ; in <DL> for DOS call
mov   ah,2    ; destination in standard output
int   21h     ; execute the DOS library call
```

The HAM_API library can be developed in a similar way. A suitable, unused interrupt in the DOS architecture would have to be chosen. To allow portability across machines, this value should be set by a *SET* command at boot time. For example, the AX register pair would provide 256 different API families, each with 256 different functions. The API TSR would be a dispatcher that loads and invokes specific APIs as required, as configured by *SET* commands.

Where Do We Go from Here?

To ensure that this approach is suitable for Amateur Radio, discussion is needed. Comments and observations are needed from application developers, product developers and amateurs on the feasibility of this approach. Once agreement on its viability is reached, ARRL-sponsored working groups should be created to develop the respective API descriptions. These working groups need not meet physically to develop the standard, but can use the Internet and Amateur Radio for communications.

Once the API is completed, volunteers would develop reference implementations for use by application developers. If these are successful, the working group develops conformance-testing criteria to certify that API implementations meet the standard. The ARRL then publishes the standard.

After publication, the working group convenes to maintain the standard on a regular basis. As APIs are implemented, lessons will be learned and improvements made in the functions and descriptions.

Lawrence G. Dobranski, VA3LGD, has a BS (with honors) in Engineering-Physics from Dalhousie University in Halifax, Nova Scotia, and a MS (Engineering) in Physics from Queen's University in Kingston, Ontario. He is presently a Senior Consultant with the EXOCOM Group of Companies in Ottawa, Ontario, specializing in Information Technology Security. Lawrence has been involved in the standards development of APIs for Information Technology Security Services. Lawrence's interests lie in the technical side of ham radio. He operates mainly HF mobile, with some dreams of serious contesting. □□