A PACKET RADIO API

David A. Beyer
Rooftop Communications
Los Altos, California

Mark G. Lewis
SRI International
Information, Telecommunications, and Automation Division
Telecommunications and Distributed Processing Program
Menlo Park, California

ABSTRACT

A variety of organizations participating in DARPA's Global Mobile Information Systems (GloMo) research program are currently designing and testing advanced wireless modem hardware, simulation components, and software protocols for distributed packet radio networks. Such efforts promise to support the efficient, reliable, and secure communication of multimedia traffic over rapidly deployed, multihop, wireless networks that serve as seamless extensions of the Internet.

In order for the participants to share research results and foster collaboration, we developed an interface specification to permit a mix-and-match approach that will enable the various network control protocols to be run on both the simulation and hardware systems currently being developed. The motivation for defining such an interface is to provide the flexibility to control various parameters such as power and data rate for future adaptive modems.

This common radio application programmer's interface (API) is intended to be language-, operating-system-, and platform-independent; extensible for supporting new or unique radio features; and available for adoption by the wider community. In this paper, we recognize the underlying capabilities of typical digital radios. Next, we categorize these capabilities into command, status, and measurement variables, and asynchronous events. We then show how these features are specified in the API, with specific examples of implementation in the "C" programming language.[*]

PURPOSE

Advanced software protocols for distributed packet radio networks are being designed and tested by such organizations as Rooftop Communications (Rooftop),[†] University of California at Santa Cruz (UCSC), SRI International (SRI), Bolt Beranek and Newman Inc. (BBN), and the University of California at Los Angeles (UCLA). Additionally, organizations such as Hughes Electronics Corporation (Hughes), UCLA, Virginia Polytechnic Institute, ITT, Utilicom, Inc., and GEC-Marconi Hazeltine Corporation (Hazeltine) are developing next-generation, highly-programmable digital radios and antennas to provide reliable and flexible wireless links for such networks. These future networks promise to support the efficient, reliable, and secure multimedia traffic over rapidly deployed, multihop wireless infrastructures. These ad hoc networks can also serve as seamless extensions of the Internet.

In order for these diverse organizations to share research results and foster collaboration, SRI, Rooftop Communications, and others in the GloMo community have developed an Application Programmer's Interface (API) for a radio that provides interface specification to permit a mix-and-match approach to the various network control protocols on the simulation and hardware systems currently being developed.

The Radio API was developed to facilitate both collaboration and independent development of the network protocols and digital radio modem hardware and continues to evolve in this capacity as well. Its intent is to allow protocol software and digital radio modems to be easily integrated, or "mixed and matched," into distributed packet radio products (or *Internet Radios*).

Specifically, our Radio API is intended to:

- Define a minimum, platform-independent interface specification between digital radio modems and network protocols
- Foster cross-organization collaboration between protocol and digital radio developers
- Permit protocol implementation and testing in the absence of actual radio hardware
- Provide standard methods for permitting radio-specific extensions
- Permit easy protocol porting between multiple radios, and vice versa.

By adhering to the Generic Device Driver specification discussed below, the independent yet source-compatible development of protocol software and device drivers for new radios may be achieved. Although encouraged when possible, adherence to the Generic Device Driver specification is not a strict requirement for many networks as particular platforms may require a different software implementation of the primitives described in the Radio API.

ARCHITECTURE

The Radio API is positioned a layer called the Transceiver Frame Control Interface (TFCI). The Radio API has been defined to avoid restricting how radio device functions are

[*]This work was performed under DARPA/CECOM Contracts DAAB07-95-C-D157 (Wireless Internet Gateways [WINGs], DAAB07-96-C-D010 (Commercial Distributed Packet Radio), and DAAB07-95-C-D318 (Global Mobile Information Systems {GloMo}).

[†]All product or company names mentioned in this document are the trademarks of their respective holders.

0-7803-4249-6/97/$10.00 © 1997 IEEE

actually implemented. Thus, these functions may be performed by analog or digital hardware, by communications controllers attached to or within the embedded microcontroller, or by device-driver software; most likely, however, some combination of these is involved.

Specifically, the Radio API assumes that the following general operations are provided by radio devices (that is below the Radio API).

- RF and IF radio stages (mixers, filters, power amplifiers, and low-noise amplifiers)
- Modulation
- Baseband spreading (direct sequence and/or fast frequency hopping)
- Preamble generation, detection, and synchronization
- Framing (start and stop flags, and zero-bit insertion)
- Error detection and/or correction (CRC computation, error control coding)

The Radio API assumes that the following operations are performed by the software protocols (i.e., above the Radio API).

- Media Access Control (MAC) protocols (channel scheduling and synchronization and avoidance of hidden terminal collisions)
- Link-layer protocols (reliable delivery of packets between neighbors or of local broadcast packets, fair sharing of link resources among neighbors, and discovery and authentication of new neighbors)
- Network-layer protocols (efficient routing without persistent loops despite mobility and dynamics, routing and queuing according to traffic service and priority requirements, efficient multicasting, and security of network control traffic)
- Internet-layer protocols (wireless-to-wired Internet routing issues, network management, and Internet-compatible interfaces).

FUNCTIONALITY

This section introduces the logical functionality of the Radio API and covers primitives (the basic elements of the API) and packet handling.

Primitives

The Radio API is defined by three basic types of *primitives*, described as follows.

- **Commands**. These are asynchronous protocols-to-device primitives for performing immediate, typically nonpersistent actions. Example command primitives include: `start packet transmissions`, `reset radio`, and `drop receive capture`.
- **Variables**. These are persistent radio state or long-term measurement primitives that support one or more of the `set`, `get`, or `increment` synchronous access operations. Control variables include the raw-channel bit rate, coding rate, center frequency, transmit power, and carrier-detect threshold. Measurement variables include received signal strength and noise level.
- **Signals**. These are asynchronous device-to-protocols primitives for reporting recent, typically nonpersistent events. The radio device should support the selective enabling and disabling of each of these signals by the protocol software. Example signal primitives include signals for `packet transmission complete`, `packet received`, `receive carrier detected`, and `receive capture detected`.

Data Packet Handling

The following objectives guided the definition of th handling of data packets across the Radio API.

- Simplify the job of the radio device driver programmer as much as possible
- Support the transmission of uninterrupted packet bursts
- Avoid packet copy operations
- Support the use of standard, serial-communications controllers that use arrays of pointers to contiguous frame buffers.*

The Radio API communicates user data in the form o packets that are identified by a start pointer and a length Though not a strict requirement, the radio should be able t handle queues of such packets on both the transmit an receive sides. Each individual packet should be received b the receiving node(s), and then passed up to the receivin protocol(s) as the same, undivided individual packet (rather than merging packets together or passing up a serie of portions of packets).

All packet buffers are owned by the protocols. Protocols ar thus responsible for allocating, freeing, and informing th radio device of the identities of packet buffers to use fo transmit and receive operations. Asynchronous signal (described below) are used to return transmit and receiv packet buffers to the protocols. The radio must also accep and store an individual *protocol buffer handle* with eac buffer, to be returned to the protocols when its associate buffer is returned through a signal (upon the completion o a packet transmission or reception). The protocol softwar can use this protocol buffer handle to hold buffer-specifi context information for each packet buffer passed down t the radio.

Uninterrupted multiple packet burst transmission can b ensured using the radio device variable `XmtBurstCnt`. Th protocols increment this variable to indicate the number o packets remaining in the burst. At the start of each packe transmission, the radio device lowers this variable. Th radio stops transmitting if the variable upon the completion of a packet transmission is zero. The protocols may modify this variable using an increment operation by incrementing the variable for packets that have yet to be handed down to the radio device before the start of, and/or during, the packet

*Implementations for such controllers are available in ICs, ASIC modules, and controllers on embedded microprocessors such as Motorola's 68360.

burst. The increment operation instructs the radio device to add the (possibly negative) value passed by the protocols to the current value of the variable. The radio device, however, never sets this variable to a number less than zero.

When transmitting a burst of packets, radio-dependent *idle* flags should be transmitted by the radio device if the next packet is not yet available, to permit the protocols to retain control of the multiple-access radio channel.

PRIMITIVE DESCRIPTIONS

This section identifies and describes the command, variable, and signal primitives, qualifiers, and return codes of the Radio API. Table 1 lists the standard return codes. Radio-specific header files may define extensions to this set. Table 2 summarizes the names, degree of requirement and qualifiers for each of the Radio API's logical primitives. Each command, variable, and signal primitive should be tagged with a return code that the radio device can use to indicate the result or status of the operation.

Table 1. Radio API Return Codes

Return Code	Description
RadioRetOk	Operation successful
RadioRetFail	General request failure
RadioRetNotInit	Radio device not initialized
RadioRetTimeOut	Request timed out
RadioRetMemOut	Memory exhausted
RadioRetHwFail	Hardware failure
RadioRetInvVersion	Invalid API version number
RadioRetInvInitData	Invalid initialization data
RadioRetInvCtlBlockPtr	Invalid control block pointer
RadioRetInvState	Operation not permitted now
RadioRetInvCmd	Invalid command
RadioRetInvVar	Invalid variable
RadioRetInvSig	Invalid signal
RadioRetInvDev	Invalid device pointer
RadioRetInvPtr	Invalid pointer argument
RadioRetInvSize	Invalid size argument
RadioRetInvQual	Invalid qualifier
RadioRetInvParam	General invalid parameter
RadioRetPktRcvFail	Packet failed to be received
RadioRetPktXmtFail	Packet failed transmission
RadioRetPktRcvError	Error in received packet

Table 2. Summary of Radio API Primitives

Commands	Requirement	Qualifiers
RadioCmdReset	M	
RadioCmdXmtPkt	M	chan
RadioCmdRcvPkt	M	chan
RadioCmdDropCapture	H*	chan
RadioCmdNativeConsole	O	
Variables		
RadioVarVersion	M	get
RadioVarName	M	get
RadioVarXmtBurstCnt	H	get/inc, chan
RadioVarMacAdr	H	get, chan
RadioVarQPkts	H	get, chan, xmt/rcv
RadioVarBitRate	H	get/set, chan, xmt/rcv
RadioVarXmtPower	H	get/set, chan
RadioVarFreq	H	get/set, chan, xmt/rcv
RadioVarCarrierThresh	H	get/set, chan
RadioVarRcvSignal	H	get, chan
RadioVarRcvNoise	H	get, chan
RadioVarCode	H[2]	get/set, chan, xmt/rcv
RadioVarMaxPkts	H	get, chan, xmt/rcv
RadioVarLoopbackMode	H	get/set, chan
RadioVarCodeRate	D[2]	get/set, chan, xmt/rcv
RadioVarCodeOffset	D[2]	get/set, chan, xmt/rcv
RadioVarFecRate	D	get/set, chan, xmt/rcv
RadioVarQBytes	O	get, chan, xmt/rcv
RadioVarSleepMode	O	get/set chan
Signals		
RadioSigAll†	M	chan
RadioSigRcvPkt	M	isr, chan
RadioSigXmtPkt	M	isr, chan

Table 2. Summary of Radio API Primitives

Commands	Requirement	Qualifiers
RadioSigError	M	`isr, chan`
RadioSigCarrierActive	H	`isr, chan`
RadioSigCarrierInactive	H	`isr, chan`
RadioSigCaptureActive	H[2]	`isr, chan`
RadioSigCaptureInactive	H[2]	`isr, chan`
RadioSigXmtActive	D	`isr, chan`
RadioSigRcvActive	D	`isr, chan`
RadioSigXmtInactive	O	`isr, chan`
RadioSigRcvInactive	O	`isr, chan`

M: Mandatory; H: Highly Desirable; D: Desirable; O: Optional.

*For direct-sequence spreading radios.

†Used by protocols to enable all supported signals at once.

Qualifiers

Each primitive can be *qualified* to give more specific instructions such as specifying the radio channel to which the command should be applied (for radios that support multiple channels), and specifying to which radio section the operation should be applied (e.g., `xmt` or `rcv`). Of course, these qualifiers will only be relevant to radios that support such capabilities.

`get` Indicates that the primitive should support `get` operations. If only `get` (and not `set` or `inc`) is specified, then the variable is read-only.

`set/inc` Indicates that the primitive should support `set` or `increment` operations. If only `set` or `inc` (and not `get`) are specified, then the variable is write-only. Increment operations instruct the radio device to add the (possibly negative) value passed to the current value. Note that variables can support either `set` or `inc`, but not both.

`xmt/rcv` Indicates that the primitive should be supported for both the transmitter and receiver sections individually (for radios that can support it).

`chan` Indicates that the primitive should be supported on a channel-specific basis for radios that support multiple simultaneous channels).

`isr` Indicates whether the primitive is running within hardware interrupt or foreground software processor modes in signal callbacks.

RADIO SPECIFIC CHARACTERISTICS

SRI's Radio API is intended to detail the required interface functionality between software protocols and radio hardware. However, because of the wide variety of radio implementations that are possible below the API, the radio must be accompanied by documentation and/or a header file that details the precise limitations, variable mappings, and extensions to this API for that specific radio.*

Examples of required radio-specific information include the mappings of variable values to actual real-world values, the time it takes for the radio to change frequencies or switch from transmit to receive mode and vice versa, the length of the preamble at the start of a packet burst, and the usefulness of the PN code parameter (which may be meaningless in a radio intended for slow frequency hopping). If the differences in these characteristics between radios are large enough, swapping proven, entirely different software protocol modules may be required. In other cases in which the differences are sufficiently small, the only requirements may be a radio-dependent configuration file with constant parameters, and tables defining the particular characteristics of the radio in use.

For example, consider the interpretation of the transmission power variable. In one case, a radio device may interpret the value of the `RadioVarXmtPower` variable as a 32-bit integer representing the actual power level in units of dBm. In another case, the radio device may not actually document the interpretation of the variable value, but may instead provide the following macros in its radio-specific header file to permit the protocols to do the conversion between the variable value and dBm:

```
RADIO_POWER_VAR_TO_DBM(varValue)
RADIO_POWER_DBM_TO_VAR(dbmValue)
```

With further experience, we expect *de facto* standard methods of handling these differences to emerge, at which time they will be added to the Radio API (and `rad_api.h`).

GENERIC DEVICE DRIVER

This section summarizes the application of the Radio API to an implementation of a standard C-language-based Generic Device Driver.† All of the primitives previously detailed have a simple, one-to-one mapping to Generic Device Driver entry point functions (which are available in a standard file `dev.h`).

The protocols use two entry points for the command and variable primitives as follows.

retCode = **DevCmd** (void *radioDev, uint32 num, uint32 quals,
    void *data, uint32 dataLen);

retCode = **DevVar** (void *radioDev, uint32 num, uint32 quals,
    void *data, uint32 dataLen);

*Requiring, instead, that all of these characteristics be made available for read access through the Radio API would be overly burdensome on the developer of the radio device driver, particularly when linking a radio-dependent-characteristics header file into the protocol software.

†The embedded and simulated communication devices drivers implemented in Rooftop's C++ Protocol Toolkit (CPT) conform to this Generic Device Driver specification.

Generic Device Drivers must also be able to signal the protocol software by calling a protocol-specified callback function when signal events occur. The prototype for this callback is as follows.

```
void(*DevSig)(void *protoDev,
        uint32 sigNum, uint32 quals,
    void *data, uint32 dataLen, uint32 retCode);
```

The `DevCmd` and `DevVar` functions should return one of the standard return codes detailed above, or a radio-specific return code. A return code is also delivered to the protocols in the `retcode` argument in calls to `DevSig`. We describe the purposes for each of the other arguments to the above functions as follows.

`radioDev` The `radioDev` argument allows the protocol software to supply the radio with an opaque handle in each call to device entry points previously supplied by the radio to the protocols. This pointer can provide context information to the radio device driver. For instance, this may be useful if the same device-driver code is being used to control multiple actual hardware devices.

`protoDev` The `protoDev` argument in the signal callback provides a similar opaque handle back to the protocol software with each signal.

`num` The `num` arguments identify the specific command, variable, or signal. Enumerated types are defined in `rad_api.h`, with names identical to the names of the primitive in the preceding section.

`data` `dataLen` The `data` and `dataLen` arguments are used to pass data or their pointers across the interface. The format for this data for each variable, command, and signal primitive is described in `rad_api.h`. Radio-specific extensions to the data must be described in the corresponding radio-specific header file.

`quals` The `quals` arguments specify the applicable qualifiers, listed in the preceding section for each primitive. This is represented as a bit mask with bits for `set`, `inc`, `get`, `xmt` and/or `rcv` direction, channel number, error flag, and error type. Macros defined in `dev.h` facilitate the use of these qualifiers while hiding the actual bit manipulation from the programmer. The upper 16-bits of these qualifiers are also reserved for radio-dependent extensions to the qualifiers.

As detailed in Table 3 below, the packet primitives use a `RadioDevPktInfo` structure defined in `rad_api.h` as

```
typedef struct _radioDevPktInfo {
      DevPktInfopktInfo;
      uint32      typeId;
      uint32      macAdr;
      uint32      errStatus;
      uint32      power;
} RadioDevPktInfo;
```

where `DevPktInfo` is defined within `dev.h` as follows.

```
typedef struct _devPktInfo {
      uint32typeId;
      bytep       buf;
      uint32      buflen;
      void        *bufHandle;
 } DevPktInfo;
```

The `RadioDevPktInfo` structure can easily be extended to carry radio-specific information by simply defining a new type which derives from the `RadioDevPktInfo` structure in the same way that `RadioDevPktInfo` was derived from `DevPktInfo`.

These API structures, defined solely to communicate information across the API (such as `RadioDevPktInfo`), are themselves always owned by the calling module. Therefore, the calling module may overwrite the data immediately following its return from the function. The called function must be sure to copy any data it intends to use later before returning from the function.

Other Generic Device Driver Entry Points

Five additional Generic Device Driver entry-points have simple responsibilities, and three others are unused. They are summarized in Table 3. The result returned from each function are defined by the return codes discussed previously.

Table 3. Generic Device Entry Points

Entry Point	Description
DevCmd	Execute a command primitive
DevVar	Change a variable primitive
(*DevSig)	Signal primitive from device
DevInit	Initialize API and device
DevOpen	Activate the device
DevClose	Deactivate the device
DevSigEnable	Selectively enable signals
DevIdle	Allow device to do idle work
DevRead	Not used for Radio API
DevWrite	Not used for Radio API
DevFlush	Not used for Radio API

FURTHER INFORMATION

Additional information about the Radio API and its sample code is available at the following WWW sites:

`http://www.glomo.sri.com/radioapi`
`http://www.rooftop.com.`