Radio Device API

Version: 17 August 1998; v2.03

Authors:[1] Dave Beyer (dave@rooftop.com), Thane Frivold, John Hight, Darren Lancaster, and Chane Fullmer of Rooftop Communications

Mark Lewis (lewis@erg.sri.com) SRI International

1 Purpose

Advanced software protocols for distributed packet radio networks are being designed, tested, and fielded by a variety of organizations including: Rooftop Communications, University of California, Santa Cruz, SRI International, Bolt Beranek and Newman, and the University of California, Los Angeles. Additionally, a variety of organizations including Hughes, UCLA, Virginia Polytechnic Institute, ITT, Utilicom and Hazeltine are developing next generation, highly-programmable digital radios and antennas to provide the reliable and flexible wireless links for such networks. These future networks promise to support efficient, reliable, and secure communication of multimedia traffic over rapidly-deployed, multihop wireless infrastructures, that can serve as seamless extensions of the Internet.

This Radio Device API was developed, and continues to evolve, to facilitate both collaboration and independent development of the network protocols and digital radio modem hardware. The 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, this Radio Device API is intended to:

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

This Radio Device API is defined using the API Framework specified in the "API Framework for Internet Radios" document.[2] Following this framework, the API is defined primarily by a set of generic *"primitives"* that can be mapped to various software or hardware implementations, as appropriate for the particular hardware/software environment. Many of the primitives for this API are inherited from the "Core Packet API" defined in the framework document.

An example software interface implementation called the "Generic Device Driver" specification is presented in an appendix. Other successful implementations include a mapping to the Unix IOCTL mechanism,[3] and a serial message passing implementation.[4]

---

[1] This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Small Business Innovative Research Program (SBIR). Refer to the *Acknowledgments* section for details

[2] See the "API Framework for Internet Radios" document, available through the http://www.rooftop.com and http://www.erg.glomo.com web-sites.

[3] Contact Mark Lewis (lewis@erg.sri.com) for specification.

[4] Specification defined by 9 April 1998 email by Fred Templin (templin@erg.sri.com) titled "An Encoding of Radio API Primitives for the ISI APT Radio via the SLIP Protocol."

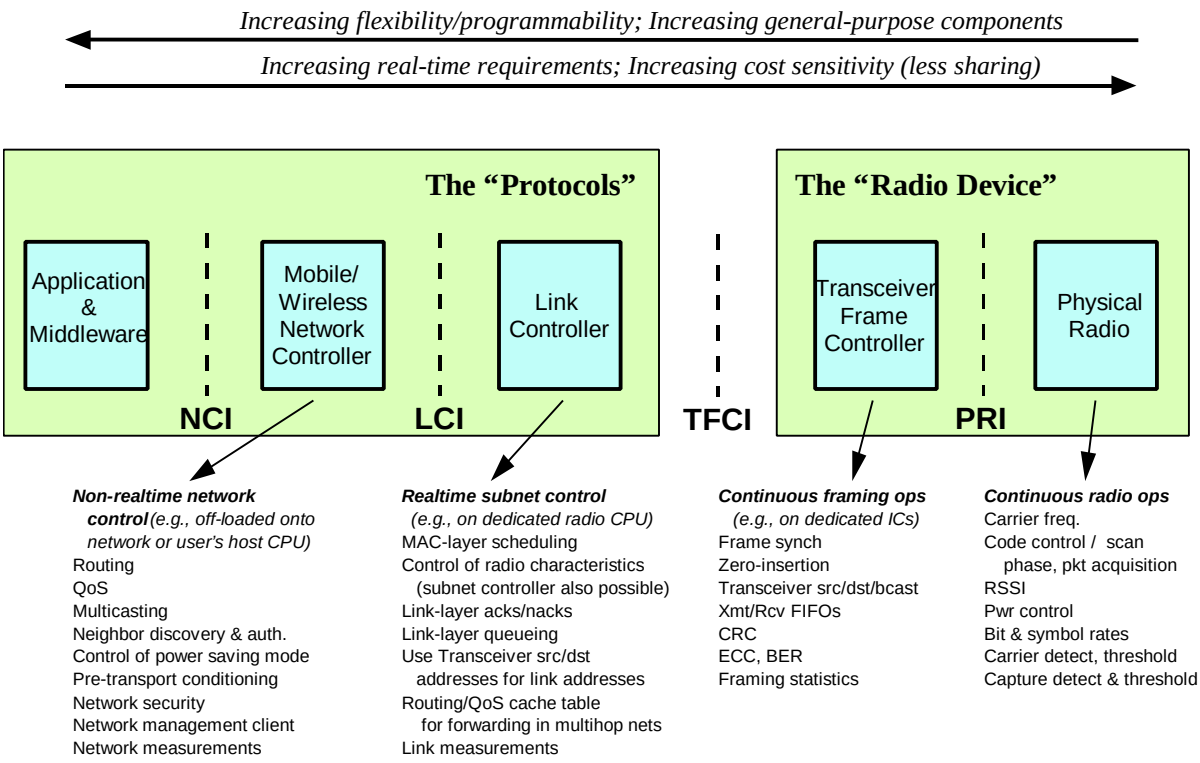

Figure 1: Position of the Radio Device API

2 Architecture

The position of this "Radio Device API" is the same as the "Transceiver Frame Control Interface" (TFCI) in Figure 1. The Radio Device API has been defined with the intent to avoid restricting how the "radio device" functions are actually implemented. Thus, these functions may be performed in analog or digital hardware, in communications controllers attached to or within the embedded microcontroller, or in device driver software, or most likely, some combination of these.

Specifically, the Radio Device API assumes that the following general operations are provided by the radio device (i.e., below" the Radio Device API).

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

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

- Media Access Control (MAC) protocols ("channel" scheduling and synchronization, avoidance of hidden terminal collisions),[5]

[5] Refer to the "Radio Device API Addendum: Support for TDMA Radios" for extensions which permit the precise scheduling of packet transmissions and receptions to occur by the radio device, under the direction of the protocols.

- Link-layer protocols (reliable delivery of packets between neighbors or of local broadcast packets, fair sharing of link resources among neighbors, discovery and authentication of new neighbors),
- Network-layer protocols (efficient routing free of loops despite dynamics, service-based routing and queuing, efficient multicasting, security of network control traffic), and
- Internetwork-layer protocols (wireless-to-wired Internet routing issues, network management, Internet-compatible interfaces).

Refer also to the companion document "Physical Radio Interface (PRI) Specification" for the specification of the PRI interface in Figure 1 for radios with synchronous serial data interfaces. This document may be of more use then the present document for those developing physical radio modules without serial frame controllers (leaving the framing functions to a serial communications controller running on the protocol's embedded microprocessor, for example).

3 Logical Functionality

The logical functionality of the Radio Device API is defined using the API Framework defined in the "API Framework for Internet Radios" document. This Radio Device API inherits from, and extends, the "Core Packet API" defined in that API Framework document.

In accordance with this API Framework, the logical functionality of the Radio Device API is defined by logical API primitives, qualifiers, and return codes. These are briefly summarized in this section. (Refer to the API Framework document for further information.) In addition, this section discusses how the basic packet handling procedures for this Radio Device API extend that of the Core Packet API.

3.1 Primitives, Qualifiers and Return Codes

The Radio Device API is defined by four basic types of *primitives*, as described in the following table and illustrated in Figure 2.

Commands	Asynchronous protocols-to-device primitives for performing immediate, typically non-persistent actions. Example command primitives include: start packet transmissions, reset the radio, and drop receive capture.
Variables	Persistent radio state or long-term measurement primitives that support one or more of the set, get, increment, and clear synchronous access operations. Control variables include the raw channel bit rate, coding rate, center frequency, transmit power, and carrier-detect threshold. Measurement variables include the received signal strength and noise level.
Responses	The synchronous device response to a protocol's command or variable operation. For a software based implementation (such as the Generic Device Driver implementation), this is typically handled using the return value from the command or variable function call (thus the dotted line in the diagram below). For a packet- or shared buffer-based implementation, the response could be returned in a separate packet or buffer, or by setting a field in a shared buffer and switching an ownership flag.
Signals	Asynchronous device-to-protocols primitives for reporting recent, typically non-persistent events. The radio device should support the selective enabling and disabling of each of these signals by the protocol software. Examples include signals for packet transmission complete, packet received, receive carrier detected, and receive capture detected.

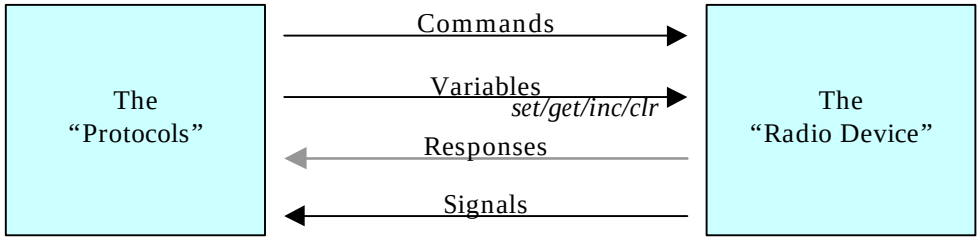

Figure 2: Basic Types of API Primitives

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 which radio section (e.g., xmt or rcv) the operation should be applied. Of course, these qualifiers will only be relevant to radios that support these capabilities.

The RadioVarSignalEnable variable is used by the protocols to enable or disable the generation of individual signals by the radio device. All signals can be enabled or disabled at once using the RadioSigAll code (a code in the Signal space defined solely for this purpose).

The Radio Device API also defines a set of return codes to provide a standard means for the radio device to indicate the success or failure status in each Response to Command and Variable operations, and in each asynchronous Signal delivered to the protocols. Examples of these codes include RadioRetOk (request accepted or performed successfully), RadioRetFail (general request failure), and RadioRetInvCmd (command is invalid or has not been implemented by this radio device).

3.2 Data Packet Handling

The following objectives guided the definition for the Core Packet API (which is inherited by this API):

- Simplify the job of the radio device driver programmer to the extent possible.
- Avoid packet copy operations.
- Support the use of standard, serial-communications controllers within the logical "Transceiver Frame Controller" of Figure 1 that use arrays of pointers to contiguous "frame buffers".[6]

As described in the Core Packet API definition, the Radio Device API communicates user data in the form of packets, identified by a start pointer and a length. Though not a strict requirement, the radio should be able to handle queues of such packets on the transmit and receive sides. A packet information structure accompanies each packet. This packet information structure includes fields for the packet buffer start and stop pointers, a protocol buffer handle, device address fields, bit error information, the precise transmit and receive times stamped by the lower-modules of the transmitting and receiving nodes, and a list of (variable, value) pairs to permit packet-specific tuning of transmit characteristics, or to report packet-specific measurements on received packets.[7] Asynchronous signals (RadioSigXmtPkt and RadioSigRcvPkt) are used to return transmit and receive packet buffers and accompanying packet information structures back to the protocols. The reader is referred to the Core Packet API for further information on the definition of these core packet-handling capabilities.

In addition, packet-handling extensions introduced in this Radio Device API were designed to:

- Support transmission of uninterrupted packet bursts.

[6] Implementations for such controllers are available in IC's, ASIC modules, and controllers on embedded microprocessors such as Motorola's 68360.
[7] See the RadioDevPktInfo structure in the Generic Device Driver implementation for an example (Section A.1).

Uninterrupted transmission of multi-packet bursts can be ensured formally using the variable *"RadioVarXmtBurstCnt"* introduced by the Radio Device API. Multi-packet bursts allow the protocols to transmit a number of consecutive packets without incurring the overhead of framing preambles between packets despite possible processing delays that may delay the delivery of these consecutive packets from the protocols to the radio device. Also, multi-packet bursts ensure that the transmitter remains "on" so that neighboring nodes will be able to detect that the channel is occupied (via the *RadioSigCarrier* and/or *RadioSigCapture* signals), and can refrain from transmitting interfering packets. Multi-packet bursts must be destined for a single receiver or set of receivers, and should generally use a single set of radio transmit characteristics (since there will be no synchronization preambles for each packet other than the first).

To ensure a multi-packet burst, the protocols increment this *RadioVarXmtBurstCnt* variable to indicate the number of packets in the upcoming burst, or remaining in an active burst. At the start of each packet transmission, the radio device decrements this variable. The radio devices turns off the transmitter only if this variable is zero upon the completion of a packet transmission and there are no packets in the radio device's transmit queue. The protocols may modify this variable, using an "increment" operation, before the start of, and/or during, the packet burst, but only by incrementing for packets that have yet to be handed down to the radio device. If the variable is incremented to a positive number after the radio has already started shutting down the transmitter, the radio should complete the shut down, and then treat the variable as having been incremented during an idle or receive state (i.e., wait for the next *RadioCmdXmtPkt* to turn the transmitter back on). The increment operation instructs the radio device to add the (possibly negative) value passed by the protocols to the current value of the variable. However, the radio device 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 while the next packet is not yet available.

In the default transmit mode (see *RadioVarXmtMode*), packet transmission commands take precedence over packet receptions.[8] Therefore, in the default transmit mode, if the radio device receives a command to transmit a packet (on a given channel for multichannel radios)[9] while it is in the midst of receiving a packet (on that channel), it aborts the packet reception and activates its transmitter circuitry immediately, unless the radio can support simultaneous receive and transmits on this same channel.

If a variable primitive operation is performed during a packet reception that affects the radio's receive operation, the radio device should implement the change immediately, dropping the incoming packet if necessary. If a variable primitive operation is performed during a packet transmission that affects the radio's transmit operation, the radio device should wait for the end of the transmission before implementing the change. However, if a change to the transmit characteristics are attempted during a formal packet burst (i.e., while *RadioVarXmtBurstCnt* > 0) the radio may a) ignore the change; b) do whatever is necessary to implement the changes before transmitting the next packet in the burst (e.g., inserting another preamble if necessary); or c) return *RadioRetPktXmtFail* (if the change was attempted using the packet's information structure), or *RadioRetInvState* (if the change was attempted by a variable primitive operation). Actual operation should be documented in the radio-specific header file.

[8] This gives the MAC layer within the protocol software complete control over when to schedule packet transmissions. For instance, the MAC layer may know that the intended recipient for the incoming packet is some other distant node, who would not be interferred by a simultaneous transmission by this node.

[9] In this document, the term "radio channel" and "channel number" refers to a logically separate modem/IF/RF chain that can be used independently of other modem/IF/RF chains across the API.

3.3 Precedence for Radio Waveform Characteristics

In accordance with the API Framework, the following precedence is used by the radio device to determine the current settings for the radio's waveform characteristics:

Highest precedence	Characteristics specified in a packet information structure for an active packet transmission or reception attempt (according to the packet information structure associated with the packet being transmitted or being filled with received data).
Lowest precedence	Characteristics specified by the persistent radio state variables (RadioVar... primitives).

4 Radio Device API Definition

4.1 Qualifiers

Each primitive in the Radio Device API can be qualified by one or more of the qualifiers listed in this section. The following qualifiers are inherited from the Core API defined in the API Framework document.

get	Primitive should support "get" operations.
set	Primitive should support "set" operations.
inc	Primitive should support "increment" operations.
clr	Primitive should support "clear" operations.
info	Used to retrieve capability information for variable primitives.
isr	Tagged to signals running within an interrupt or high-priority thread.

In addition, the Radio Device API introduces the following qualifiers:

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

4.2 Primitives

This section identifies and describes the command, variable, response, and signal primitives of the Radio Device API. Each primitive is labeled with Mandatory, Highly desirable, Desirable, or Optional, indicating the degree of requirement by the protocols. A "Data" field indicates the generic input and/or output data communicated across the API by each primitive.

4.2.1 Commands

The command primitives inherited from the Core Packet API, along with their aliases, include the following.

Command, Alias	**Rqmt**	**Qualifiers**	**Data**
CmdReset, RadioCmdReset	M		
CmdNativeConsole, RadioCmdNativeConsole	O	*chNum*	User cmd string and response string.
CmdProcExec, RadioCmdProcExec	O	*chNum*	Diagnostic, or other procedure, to exec.
CmdXmtPkt , RadioCmdXmtPkt	M	*chNum*	Pkt buf & its protocol buf handle
CmdRcvPkt, RadioCmdRcvPkt	M	*chNum*	Pkt buf & its protocol buf handle

Note the addition of the "chNum" qualifier to some of these inherited primitives. In addition, the following inherited command has other extensions specific to this Radio Device API.

RadioCmdXmtPkt Core Packet API Command

Requirement:	Mandatory
Qualifiers:	*chNum*
Data:	(See Core Packet API)
Description:	(See Core Packet API.) Also, see RadioVarXmtMode for how the packet should be handled by the radio device when a carrier is detected on the channel.

The new commands introduced by this API are the following.

RadioCmdDropCapture Command

Requirement:	Highly desirable (for DS radios only)
Qualifiers:	*chNum*
Data:	
Description:	Commands the radio to drop code-synchronization of an incoming chip stream. Receiver should return to “code-synchronization-search” mode.

4.2.2 Variables and Groups

This section lists the radio state variables that may be available to the protocols. The variable primitives inherited from the Core Packet API along with their aliases, include:

Variable, Alias	**Rqmt**	**Qualifiers**	**Data**
VarVersion, RadioVarVersion	M	*get*	Version string (e.g., "2.0.1; 2 July 1998")
VarName, RadioVarName	M	*get*	Name string given to lower module
VarClass, RadioVarClass	H	*get*	API class identifier
VarStatus, RadioVarStatus	H	*get, chNum*	Number indicating device status
VarSigEnable, RadioVarSigEnable	H	*get/set, chNum*	Signal number to enable or disable
VarGroupSelect, RadioVarGroupSelect	H	*set, chNum*	groupClassId, groupInstanceNum
VarGroupValues, RadioVarGroupValues	H	*get (set) , chNum*	groupClass & instance, {var, value} pairs
VarGroupClassName, RadioVarGroupClassName	H	*get, chNum*	Group class name string
VarGroupClassSize, RadioVarGroupClassSize	H	*get, chNum*	Group class size (number of variables)
VarMacAdr, RadioVarMacAdr	H	*get/set, clr, chNum*	MAC address / mask for this packet device
VarQPkts, RadioVarQPkts	H	*get, xmt/rcv, chNum*	No. of packets in queue
VarBitRate, RadioVarBitRate	H	*get/set, xmt/rcv, chNum*	Raw channel bit rate
VarMaxPkts, RadioVarMaxPkts	H	*get, xmt/rcv, chNum*	Max number of packet buffers
VarTestMode, RadioVarTestMode	H	*get/set/clr, chNum*	Test (e.g., loopback) mode
VarMtu, RadioVarMtu	H	*get, chNum*	Max. packet buffer size in bytes.
VarGroupClassInstances, RadioVarGroupClassInstances	H	*get, chNum*	returns # of instances given groupClassId
VarGroupClassDefine, RadioVarGroupClassDefine	O	*set, chNum*	groupClassId, name, size, instances, var list
VarGroupDefineNumMax, RadioVarGroupDefineNumMax	O	*get, chNum*	# of dynamically-defined groups supported
VarPktHeadLen RadioVarPktHeadLen	O	*get, chNum*	Number of bytes
VarPktTailLen, RadioVarPktTailLen	O	*get, chNum*	Number of bytes
VarQBytes, RadioVarQBytes	O	*get, xmt/rcv, chNum*	Total no. of bytes in queue
VarMaxMacAdrs, RadioVarMaxMacAdrs	O	*get, chNum*	Max. no. of rcv MAC address / masks

Note the addition of the "chNum" qualifier to most of these inherited primitives.

The new variables introduced by this API are the following.

RadioVarXmtBurstCnt Variable

Requirement:	Highly desirable
Qualifiers:	*get/inc, chNum*
Data:	Holds the number of packets yet to be sent in current burst. The *inc* operation provides a reservation for a no. of additional packets to be sent in current burst, or the (neg.) number that were previously reserved, but will not be sent.
Description:	A variable used by the protocol to inform the radio of the number of packets *additional* packets remaining to be transmitted in an uninterrupted burst of packets.

RadioVarXmtPower Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set, chNum*
Data:	Transmission power.
Description:	The RF transmission power in terms of units or a table index as defined within the radio-specific header file.

RadioVarFreq Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set, chNum, xmt/rcv*
Data:	Center frequency.
Description:	The RF center frequency in terms of units or a table index as defined within the radio-specific header file.

RadioVarCarrierThresh Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set, chNum*
Data:	Carrier-detection threshold
Description:	The receive carrier detection threshold, in terms of units or a table index as defined within the radio-specific header file.

RadioVarRcvSignal Variable

Requirement:	Highly Desirable
Qualifiers:	*get, chNum*
Data:	Receive signal power
Description:	The current receive signal power measurement, in terms of units or a table index, and averaged over some period, as defined within the radio-specific header file.

RadioVarRcvNoise Variable

Requirement:	Highly Desirable
Qualifiers:	*get, chNum*
Data:	Receive noise power
Description:	The current receive noise power measurement, in terms of units or a table index, and averaged over some period, as defined within the radio-specific header file.

RadioVarCode Variable

Requirement:	Highly Desirable (for DS radios)
Qualifiers:	*get/set, chNum, xmt/rcv*
Data:	PN code
Description:	The pseudo-random code in terms of units or a table index as defined within the radio-specific header file.

RadioVarXmtMode Variable

Requirement:	Highly Desirable
Qualifiers:	*set/get,chNum*
Data:	The transmit mode that the radio device should use among {RadioXmtModeCarrierIgnore (the default), RadioXmtModeCarrierWait, RadioXmtModeCarrierFail}
Description:	When operating in the CarrierIgnore mode the radio will immediately transmit any packet passed for transmission regardless of the receive signal state of the channel. When in the CarrierFail mode, the radio will first measure the RadioRcvVarSignal value, and if above the RadioVarCarrierThresh threshold, will immediately return a Response with the RadioRetPktXmtFailCarrier return code.[10] When in the CarrierWait mode, the radio will transmit the packet as soon as RadioVarRcvSignal is below RadioVarCarrierThresh. Upon reset or power-up, the radio will enter the RadioXmtModeCarrierIgnore state, where it will remain until changed by a RadioVarXmtMode:set operation.

RadioVarSilentMode Variable

Requirement:	Desirable
Qualifiers:	*get/set, chNum*
Data:	on/off flag
Description:	Controls receive-only-mode. No packet transmissions are permitted while in silent mode (including any synchronization control packets automatically sent by the radio device).

[10] Also, in this RadioXmtModeCarrierFail mode, if for some reason RadioVarRcvSig becomes >= RadioVarCarrierThresh after sending the RadioRetOk response to the protocols but before starting the actual transmission, then RadioSigXmtPkt should be sent with the RadioRetPktXmtFail return code.

RadioVarCodeRate Variable

Requirement: Desirable (for DS radios)
Qualifiers: *get/set, chNum, xmt/rcv*
Data: PN code rate
Description: The pseudo-random code rate (e.g., chips/sec) in terms of units or a table index as defined within the radio-specific header file.

RadioVarCodeOffset Variable

Requirement: Desirable (for DS radios)
Qualifiers: *get/set, chNum, xmt/rcv*
Data: PN code offset
Description: The offset in a pseudo-random code sequence, in terms of units or a table index as defined within the radio-specific header file.

RadioVarSymbolRate Variable

Requirement: Desirable
Qualifiers: *get/set, chNum, xmt/rcv*
Data: Modulation symbol rate
Description: The modulation symbol rate (e.g., symbols/sec) in terms of units or a table index as defined within the radio-specific header file.

RadioVarModulationType Variable

Requirement: Desirable
Qualifiers: *get/set, chNum, xmt/rcv*
Data: Modulation type
Description: The modulation type (e.g., BPSK, QPSK, …) in terms of a table index as defined within the radio-specific header file.

RadioVarFecRate Variable

Requirement: Desirable
Qualifiers: *get/set, chNum, xmt/rcv*
Data: Error control coding rate
Description: The error control coding rate, in terms of units or a table index as defined within the radio-specific header file.

RadioVarPowerMode Command

Requirement: Optional
Qualifiers: *set/get chNum, xmt/rcv*
Data: Power mode to enter; one of {RadioPowerUp, RadioPowerStandby, RadioPowerSleep, RadioPowerDown} (or radio-specific extension).
Description: Used to put the radio into specific power-saving mode.

The Radio Device API also defines the following read-only variable group:

RadioVarGroupOpConsts Variable Group

Qualifiers	*get, chNum*	
Description:	Used to discover the (default) operational constants of this radio device.	
Variables:	RadioVarPreambleLen	in bits
	RadioVarXmtFrameLen	in bits
	RadioVarXmtBurstMaxLen	in bytes
	RadioVarFreqIndexNum	no. of frequency indexes
	RadioVarFreqChangeDelay	in usecs
	RadioVarXmtRampUpDelay	in usecs
	RadioVarXmtRampDownDelay	in usecs
	RadioVarRcvCaptureLostDelay	in usecs
	RadioVarRcvCarrierOffDelay	in usecs
	RadioVarRcvSsCodingGain	in dB
	RadioVarRcvCaptureThresh	in dBm
	RadioVarRcvCaptureSNThresh	in dB
	RadioVarRcvMaintainSyncSNThresh	in dB
	RadioVarRcvInterferenceThresh	in dBm

4.2.3 Signals

The signal primitives inherited from the Core Packet API, along with their aliases, include:

Signal, Alias	Rqmt	Qualifiers	Data
SigAll, RadioSigAll	M	*chNum*	
SigError, RadioSigError	M	*isr, chNum*	Number indicating the error.
SigStatus, RadioSigStatus	H	*isr, chNum*	Number indicating the new status.
SigRcvPkt, RadioSigRcvPkt	M	*isr, chNum*	Rcv'd pkt buf & its proto buf handle
SigXmtPkt, RadioSigXmtPkt	M	*isr, chNum*	Xmt'd pkt buf & its proto buf handle
SigProcResults, RadioSigProcResults	D	*isr, chNum*	Results of a CmdProcExec.

Note the addition of the "chNum" qualifier to each of these inherited signals. The Radio Device API also introduces the following signals.

RadioSigCarrierActive Signal

Requirement:	Highly Desirable
Qualifiers:	*isr, chNum*
Data:	
Description:	A signal indicating that the received signal strength (RadioVarRcvSignal) has risen above the carrier threshold (RadioVarRcvCarrierThresh).

RadioSigCarrierInactive Signal

Requirement: Highly Desirable
Qualifiers: *isr*, *chNum*
Data:
Description: A signal indicating that the received signal strength (RadioVarRcvSignal) has fallen below the carrier threshold (RadioVarRcvCarrierThresh).

RadioSigCaptureActive Signal

Requirement: Highly Desirable (for DS radios)
Qualifiers: *isr*, *chNum*
Data:
Description: A signal indicating that the receiver has synchronized to an incoming code sequence.

RadioSigCaptureInactive Signal

Requirement: Highly Desirable (for DS radios)
Qualifiers: *isr*, *chNum*
Data:
Description: A signal indicating that the receiver has lost synchronization to an incoming code sequence.

RadioSigXmtActive Signal

Requirement: Desirable
Qualifiers: *isr*, *chNum*
Data:
Description: A signal generated when the transmitter becomes active, at the start of a single- or multiple-packet transmission.

RadioSigRcvActive Signal

Requirement: Desirable
Qualifiers: *isr*, *chNum*
Data:
Description: A signal generated when the receiver becomes active, and ready to start receiving a new packet. This should occur sometime after a RadioCmdReset commands, or after a RadioSigXmtInactive signal is generated. This may also occur upon completion of other radio transitions such as changing frequencies.

RadioSigXmtInactive Signal

Requirement: Optional
Qualifiers: *isr*, *chNum*
Data:
Description: A signal generated when the transmitter becomes inactive, at the end of a packet or packet burst.

RadioSigRcvInactive Signal

Requirement:	Optional
Qualifiers:	*isr, chNum*
Data:	
Description:	A signal generated when the receiver becomes inactive. This should occur after receiving a RadioCmdXmtPkt command, and sometime before a RadioSigXmtActive signal is generated. This may also occur when starting other radio transitions such as changing frequencies.

4.3 Return Codes

The Radio Device API inherits the return codes defined in the Core Packet API, and introduces no new return code. Table 1 lists the standard, RadioReturn codes. Radio-specific header files may define extensions to this set.

Table 1: Radio Device API, RadioReturn Codes	
RetOk, RadioRetOk	Operation accepted or successfully performed.
RetFail, RadioRetFail	General request failure.
RetNoInit, RadioRetNoInit	Lower module not initialized.
RetTimeOut, RadioRetTimeOut	Request timed out.
RetMemOut, RadioRetMemOut	Lower module is out of memory.
RetHwFail, RadioRetHwFail	Hardware failure. A ***strong*** suggestion that the upper module should issue a CmdReset command.
RetInvVersion, RadioRetInvVersion	Invalid API version number (or version not supported)
RetInvInitData, RadioRetInvInitData	Invalid initialization data
RetInvCtlBlockPtr, RadioRetInvCtlBlockPtr	Invalid control block pointer (used for device driver implementations)
RetInvState, RadioRetInvState	Operation not permitted in current state.
RetInvCmd, RadioRetInvCmd	Invalid command or command not implemented by this lower module.
RetInvVar, RadioRetInvVar	Invalid variable or variable not implemented by this lower module.
RetInvSig, RadioRetInvSig	Invalid signal or signal not implemented by this lower module.
RetInvDev, RadioRetInvDev	Invalid "device" pointer (used for context by some implementations)
RetInvPtr, RadioRetInvPtr	Invalid pointer argument.
RetInvSize, RadioRetInvSize	Invalid size argument.
RetInvQual, RadioRetInvQual	Invalid qualifier.
RetInvParam, RadioRetInvParam	General invalid parameter error.
RetInvGroupClass, RadioRetInvGroupClass	Invalid group class identifier.
RetInvGroupInstance, RadioRetInvGroupInstance	Invalid group instance number.
RetPktRcvFail, RadioRetPktRcvFail	Pkt failed to be received, returning packet buffer. Pkt buffer is being returned before any rcv has completed for this buffer.
RetPktXmtFail, RadioRetPktXmtFail	Pkt failed to be transmitted, returning packet buffer. Pkt buffer is being returned before any xmt attempt has completed.
RetPktXmtFailCarrier, RadioRetPktXmtFailCarrier	Packet failed to be transmitted due to sensed carrier, pkt buf is being returned.
RetPktXmtFailOverflow, RadioRetPktXmtFailOverflow	Packet failed to be transmitted due overflow of device xmt queue.
RetPktXmtFailUnderrun, RadioRetPktXmtFailUnderrun	Packet failed to be transmitted due to underrun of radio xmt queue (used for radios which are put into persistent "transmit modes").
RetPktRcvError, RadioRetPktRcvError	Other error in rcv'd pkt; typically reported with SigRcvPkt accompanying a rcv'd pkt buffer with errors.
RetPktXmtError, RadioRetPktXmtError	Other error in xmt pkt; typically reported with SigXmtPkt accompanying a xmt'd pkt buffer with some xmt error detected.

4.4 Summary of Primitives

Table 2 summarizes the names, degree of requirement (M-Mandatory, H-Highly desirable, D-Desirable, O-Optional), qualifiers, and data for each of the Radio Device API's logical primitives.

Table 2: Summary of Radio Device API Primitives

Commands	Rqmt	Qualifiers	Data
RadioCmdReset	M		
RadioCmdXmtPkt	M	*chNum*	Pkt buf & its protocol buf handle
RadioCmdRcvPkt	M	*chNum*	Pkt buf & its protocol buf handle
RadioCmdDropCapture	H*	*chNum*	Drop sync capture of incoming signal
RadioCmdProcExec	O	*chNum*	Test or other radio procedure to execute
RadioCmdNativeConsole	O	*chNum*	Cmd string and user-response string

Signals	Rqmt	Qualifiers	Data
RadioSigAll[11]	M	*chNum*	
RadioSigError	M	*isr*, *chNum*	Error string
RadioSigRcvPkt	M	*isr*, *chNum*	Rcv'd pkt buf & its proto buf handle
RadioSigXmtPkt	M	*isr*, *chNum*	Xmt'd pkt buf & its proto buf handle
RadioSigStatus	H	*isr*, *chNum*	New status
RadioSigCarrierActive	H	*isr*, *chNum*	
RadioSigCarrierInactive	H	*isr*, *chNum*	
RadioSigCaptureActive	H*	*isr*, *chNum*	
RadioSigCaptureInactive	H*	*isr*, *chNum*	
RadioSigProcResults	D	*isr*, *chNum*	Results of internal test or other procedure.
RadioSigXmtActive	D	*isr*, *chNum*	
RadioSigRcvActive	D	*isr*, *chNum*	
RadioSigXmtInactive	O	*isr*, *chNum*	
RadioSigRcvInactive	O	*isr*, *chNum*	

* For direct-sequence spreading radios.

[11] Used by protocols in conjunction with RadioVarSignalEnable to enable or disable all supported signals at once; never generated by the radio device.

Variables	Rqmt	Qualifiers	Data
RadioVarVersion	M	*get*	Version string
RadioVarName	M	*get*	Name string
RadioVarClass	H	*get*	API class identifier
RadioVarStatus	H	*get, chNum*	Number indicating device status
RadioVarXmtBurstCnt	H	*get/inc, chNum*	No. of pkts yet to be sent in burst
RadioVarMacAdr	H	*get/set, clr, chNum*	MAC address / mask for this radio
RadioVarQPkts	H	*get, chNum, xmt/rcv*	No. of packets in queue
RadioVarBitRate	H	*get/set, chNum, xmt/rcv*	Raw channel bit rate
RadioVarXmtPower	H	*get/set, chNum*	Transmission power
RadioVarFreq	H	*get/set, chNum, xmt/rcv*	Center frequency
RadioVarCarrierThresh	H	*get/set, chNum*	Carrier-detect threshold
RadioVarRcvSignal	H	*get, chNum*	Receive signal power
RadioVarRcvNoise	H	*get, chNum*	Receive noise power
RadioVarCode	H*	*get/set, chNum, xmt/rcv*	PN code
RadioVarMaxPkts	H	*get, chNum, xmt/Rcv*	Max number of packet buffers
RadioVarTestMode	H	*get/set, chNum*	Test (e.g., loopback) mode
RadioVarMtu	H	*get, chNum*	Max. packet buffer size in bytes.
RadioVarXmtMode	H	*get/set, chNum*	Xmt mode for radio (carrier sense or not)
RadioVarGroupSelect	H	*set, chNum*	groupClassId, groupInstanceNum
RadioVarGroupValues	H	*get (set) , chNum*	groupClass & instance, {var, value} pairs
RadioVarGroupClassName	H	*get, chNum*	Group class name string
RadioVarGroupClassSize	H	*get, chNum*	Group class size (number of variables)
RadioVarGroupClassInstances	H	*get, chNum*	returns # of instances given groupClassId
RadioVarSigEnable	H	*get/set, chNum*	Signal number & TRUE or FALSE
RadioVarSilentMode	D	*get/set, chNum*	On / off flag
RadioVarCodeRate	D*	*get/set, chNum, xmt/rcv*	PN code rate
RadioVarCodeOffset	D*	*get/set, chNum, xmt/rcv*	PN code offset
RadioVarSymbolRate	D	*get/set, chNum, xmt/rcv*	Modulation symbol rate
RadioVarModulationtype	D	*get/set, chNum, xmt/rcv*	Modulation type
RadioVarFecRate	D	*get/set, chNum, xmt/rcv*	Error control coding rate
RadioVarPktHeadLen	O	*get, chNum*	Number of bytes
RadioVarPktTailLen	O	*get, chNum*	Number of bytes
RadioVarQBytes	O	*get, chNum, xmt/rcv*	Total no. of bytes in queue
RadioVarPowerMode	O	*get/set chNum*	Power mode for radio device
RadioVarMaxMacAdrs	O	*get, chNum*	Max. no. of rcv MAC address / masks
VarGroupClassDefine	O	*set, chNum*	groupClassId, name, size, instances, var list
VarGroupDefineNumMax	O	*get, chNum*	# of dynamically-defined groups supported

5 Acknowledgments

The development of this API Framework was initially supported by the Small Business Innovation Program (SBIR) through Rooftop's Commercial Distributed Packet Radio project (contract no. DAAB07-96-C-D010). It's continuing evolution has been supported by the Defense Advanced Research Projects Agency (DARPA) through the Global Mobile (GloMo) program's Wireless Internet Gateways (WINGS) project (contract no. DAAB07-95-C-D157), SRI International's GloMo program, and the Adaptive Signal Processing and Networking (ASPEN) program (contract no. F30602-97-C-0314). WINGS is a collaborative effort by the University of California, Santa Cruz (the prime contractor) and Rooftop Communications, and ASPEN is a collaborative effort by Raytheon Corp. (prime contractor) and Rooftop. This document has also benefited from the constructive review and feedback from other members of the GloMo contractor community, Government reviewers, and commercial radio employees, including in particular:[12]

BBN / GTE Internetworking	Ram Ramanathan, Martha Steenstrup, Greg Lauer, David Li, Regina Rosales Hain	*SRI International*	(Authors plus) Fred Templin, Elin Klaseen, Ambatipudi Sasty
CECOM	Jay Staba	*UC Berkeley*	Bob Broderson
Hazeltine	Jim Limardo	*UCLA*	Charles Chien, Walt Boring,
ISI	Mike Gorman, Jack Wills	*Univ. of California, Santa Cruz*	JJ Garcia-Luna-Aceves
ITT	Lester Matheson	*Univ. of Kansas*	Gary Minden
Raytheon	George Vardakas, Jason Erickson, Kevin Burns, Dale Feikema, Jim Thomas, James Tsusaki, Edwin Lee	*Univ. of Texas*	Heinrich Foltz, James McLean
Rockwell	Jim Stevens	*Utilicom Inc.*	Nuno Bandeira, Steve Schapel, Steve Wrolstad
Rooftop Communications	(Authors plus) Bich Nguyen	*Virginia Tech.*	Scott Midkiff, Nattavut Smavatkul, Francis Dominique, Anwarul Hannan

[12] This is an undoubtedly incomplete list, acknowledging the people who have provided contributions and/or constructive review & feedback during the development of this still-evolving API. However, this list is not intended to suggest complete agreement on, nor total endorsement of, the current API by those listed.

A. Generic Device Driver Implementation Mechanism Overview

This section summarizes the application of the Radio Device API to a software implementation of a standard C-language-based "Generic Device Driver."[13] All of the primitives detailed above have a simple, one-to-one mapping to one of three Generic Device Driver entry point functions. The section contains the following subsections:

- Commands, Variables and Signals
- Other Generic Device Driver Entry Points

A.1 Commands, Variables and Signals

Generic Device Drivers must provide entry points for accepting commands from the protocols and accepting requests to set or get radio variables. The prototypes for these two entry points (which are available in a standard file *dev.h*) are:

```
uint32    (*DevCmdFp)    ( void * radioDev, uint32 num, uint32 quals,
                           void * data, uint32 dataLen );

uint32    (*DevVarFp)    ( void * radioDev, uint32 num,  uint32 quals,
                           void * data,   uint32 dataLen );
```

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:

```
void      (*DevSigFp)    ( void * protoDev, uint32 sigNum, uint32 quals,
                           void * data, uint32 dataLen, uint32, RadioRetCode);
```

The (*RadioDevCmdFp) and (*RadioDevVarFp) functions should, RadioReturn one of the standard, RadioReturn codes detailed previously, or a radio-specific, RadioReturn code. Also, a, RadioReturn code is delivered to the protocols in the, RadioRet*code* argument in calls to (*RadioDevSigFp). The following table describes the purposes for each of the other arguments to the above functions.

radioDev *protoDev*	The *radioDev* argument allows the protocol software to supply the radio with an opaque "handle", previously supplied by the radio to the protocols, in each call to device entry points. 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. 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 "primitive" names in the preceding section.
quals	The q*uals* arguments specify the applicable qualifiers, as listed in the preceding section for each primitive. This is represented as a bit mask with bits for Set, Inc, or Get, Xmt and/or Rcv direction, channel number, error flag, error type, etc. Macros defined in *dev.h* (and listed in Table 5) facilitate the use of these qualifiers while hiding the actual bit manipulation from the programmer. Also, the upper 16-bits of these qualifiers are reserved for radio-dependent extensions to the qualifiers.
data	The *data* and *dataLen* arguments are used to pass data, or pointers to data,

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

dataLen across the interface. The format for this data for each variable, command, and signal is described in the table below, as well as in *rad_api.h*. Radio-specific extensions to the data must be described in the corresponding radio-specific header file.

As detailed in the table below, the packet primitives use a RadioDevPktInfo structure defined in *rad_api.h* . Currently, the radio device does not extend the DevPktInfo structure (defined in *dev.h* for all generic packet devices), and uses the charGroup and DevChar structure described below to control the xmt or rcv radio characteristics on an individual packet basis. For consistency (and in case basic Radio Device fields are required to extend DevPktInfo in the future), RadioDevPktInfo is defined in *rad_api.h* as:

```
typedef   DevPktInfo   RadioDevPktInfo;        // signature of RadioDevPktInfo is also set to
                                                // be equal to that of DevPktInfo
```

DevPktInfo is defined within *dev.h* as:

```
typedef struct _devPktInfo {
        uint32              signature;      // Identifies this type for debugging

        bytep               buf;            // points to start of buffer
        uint32              bufLen;         // len of buffer or of data in buf
        void    *           bufHandle;      // holds "protocol buf handle"

        uint32              macAdr;         // dst or src MAC adr for xmt/rcv pkts
        uint32              errStatus;      // 0 - no errors; 1 – uncorrectable bit errors;
                                            // 2 – totally correctable bit errors. Other error
                                            // codes defined in device-dependent hdr file

        uint32              rcvTime_s;  [14] // Timestamp for reporting the precise rcv
        uint32              rcvTime_us;     //   time at the receiving node
        uint32              xmtTime_s;      // Timestamp for controlling and reporting the
        uint32              xmtTime_us;     //   precise transmit time at the transmitting node

        uint32              charModsNum;    // Number of device characteristics modifiers
                                            // in charMods[] array
        DevChar             charMods[DEV_PKT_CHAR_MODS_MAX];
                                            // List of device characteristics used for this
                                            // pkt, or selection of a variable group.
} DevPktInfo;
```

[14] *Deleted "RadioVarRcvTOA variable, since this functionality should be covered by time_s and time_us in the DevPktInfo struct.*

Where the DevChar structure is used to communicate the radio waveform transmit characteristics or receive measurements for individual packet transmissions or receptions.

```
typedef struct devChar {
        uint32          name;       / / The (enumerated) name of the
                                    / / characteristics variable primitive
                                    / / or VarGroupSelect
        uint32          value;      / / The new or measured value
                                    / / for this characteristic, or the encoded
                                    / / classId and instanceNum for a group.
} DevChar;
```

Variable primitives that are relevant for controlling or measuring radio device characteristics include:

- RadioVarBitRate,
- RadioVarXmtPower,
- RadioVarFreq,
- RadioVarRcvSignal,
- RadioVarRcvNoise,
- RadioVarCode,
- RadioVarCodeOffset,
- RadioVarCodeRate, and
- RadioVarFecRate.

Note that the memory pointed to by the "chars" variable (if any) must remain valid for the radio device for the same period that the DevPktInfo structure (or derived type) is valid.

The following default radio MAC broadcast address is also defined in *rad_api.h*:

```
#ifndef RADIO_MAC_BROADCAST_ADR
#define RADIO_MAC_BROADCAST_ADR  0xFFFFFFFF
#endif
```

The RadioDevPktInfo structure can easily be extended to carry radio-specific information simply by defining a new type which "derives" from the RadioDevPktInfo structure in the following way:[15]

```
typedef struct tRadioDevPktInfo {
        RadioDevPktInfo     radioPktInfo;   / / must be first
        uint32              signature;      / / Identifies this type for debugging
        uint32              slotNum;        / / xmt or rcv slot number of pkt sched (or
                                            / / T_RADIO_SLOT_NUM_UNUSED)
        uint32              slotOffset;     / / offset into slot in usecs (or
                                            / / T_RADIO_SLOT_OFFSET_UNUSED)
        uint32              rcvDuration;    / / The duration, in slots, during which
                                            / / reception should be attempted (or
                                            / / T_RADIO_SLOT_RCV_DURATION_UNUSED)
} TRadioDevPktInfo;
```

Unless specifically indicated otherwise, "API structures," defined solely to communicate information across the API (such as DevPktInfo) are themselves always owned by the calling module and must ensure that the data in the structure remains valid until the protocols receive the appropriate signal (e.g., RadioSigPktXmt or RadioSigPktRcv), RadioReturning control of the structure memory to the protocols.

Table 3 lists the specific use of the *data* and *dataLen* arguments for each primitive in the Radio Device API. For the variable primitives, the usage is defined according to whether the operation is a set, get, or inc.

[15] This example provides a glimpse into the TDMA Addendum to the Radio Device API.

So that the called function called can determine the type and length of any data argument passed, and perform consistency checking, the *dataLen* is always set to one of following:

- length of the memory (structure or string plus any terminating characters) pointed to by the *data*, or
- either 0 or **DEV_ ARG_LEN_UNUSED** (defined in dev.h) indicating that the data argument is not used.

Unused data arguments can either be set to (void *) 0 or **DEV_ARG_PTR_NULL**.

The RadioVarMacAdr variable is used to specify the MAC address and mask pair(s) for this radio device, or set to the radio-specific MAC broadcast address to receive all packets (*RADIO_MAC_BROADCAST_ADR*). This variable uses the RadioDevMacAdrInfo structure to convey the MAC address information:

```
typedef struct {
    uint32   macAdrLen;    /* Number of bytes */
    void   * macAdr;
    void   * macAdrMask;
} RadioDevMacAdrInfo;
```

For a specific MAC address, the mask should be set to all 1's. Incoming packet frames are only accepted if the destination address in the frame header is the MAC broadcast address, or is equal to the MAC address of the receiving node.

Table 3: Data Argument Usage By Generic Radio Devices

Commands	***data***	***dataLen***
RadioCmdReset RadioCmdDropCapture	NULL	NULL
RadioCmdXmtPkt RadioCmdRcvPkt	Ptr to RadioDevPktInfo	sizeof (RadioDevPktInfo)
RadioCmdProcExec	Ptr to int32 with code for procedure	sizeof (int32)
RadioCmdNativeConsole	Ptr to Null-term. cmd string (on input) and overwritten cmd string (output).	Max len of user-response string + 1 (bytes)

Variables	***data***	***dataLen***
RadioVarXmtBurstCnt RadioVarMacAdr RadioVarBitRate RadioVarXmtPower RadioVarFreq RadioVarCarrierThresh RadioVarCode[16] RadioVarTestMode RadioVarMtu RadioVarXmtMode RadioVarSilentMode RadioVarCodeRate RadioVarCodeOffset RadioVarSymbolRate RadioVarModulationType RadioVarFecRate RadioVarPowerMode RadioVarMaxMacAdrs RadioVarGroupDefineNumMax RadioVarGroupClassSize RadioVarGroupClassInstances	*get:* Ptr to int32 for result *set:* Ptr to int32 with new value *inc*: Ptr to int32 with new value	sizeof (int32) sizeof (int32) sizeof (int32)
RadioVarGroupSelect	*set-only:* Ptr to int32 with groupClassId, groupInstanceNum encoded in upper and lower 16 bits	sizeof (int32)
RadioVarGroupValues RadioVarGroupClassDefine	*set/get:* Ptr to array of {variable, value} pairs.	sizeof (int32) * 2 * length of array
RadioVarGroupClassName	*set/get:* Ptr for radio returned, Null-terminated string. Passed down with first (int32) encoded with groupClassId, groupInstancesNum	Max. len of, Radio returned string + 1 (bytes)
RadioVarClass RadioVarStatus RadioVarQPkts RadioVarMaxPkts RadioVarQBytes RadioVarRcvSignal RadioVarRcvNoise RadioVarPktHeadLen RaidioVarPktHeadLen	*(get-only)* Ptr to int32 for result	sizeof (int32)
RadioVarVersion RadioVarName	*(get-only)* Ptr for Radio returned, Null-terminated string	Max. len of, Radio returned string + 1 (bytes)
RadioVarMacAdr	*Ptr to RadioDevMacAdrInfo*	Sizeof (*struct*)
RadioVarSignalEnable[17]		

[16] For radios using codes greater than 32-bit codes, a structure should be used and defined in the radio-specific header file.

[17] The RadioVarSignalEnable variable is implemented using the (*DevSignalEnableFp) entry point. See Section 0.

Signals	***data***	***dataLen***
RadioSigCaptureActive RadioSigCaptureInactive RadioSigXmtActive RadioSigXmtInactive RadioSigRcvActive RadioSigRcvInactive	NULL	NULL
RadioSigRcvPkt RadioSigXmtPkt	Ptr to RadioDevPktInfo	sizeof (RadioDevPktInfo)
RadioSigError	Null-terminated string	string len + 1 (bytes)
RadioSigStatus	int32 with new status	sizeof (int32)
RadioSigProcResults	int32 wth procedure results code	sizeof (int32)
RadioSigCarrierActive RadioSigCarrierInactive	int32 rcv power measurement	sizeof (int32)

A.2 Other Generic Device Driver Entry Points

Of the remaining eight Generic Device Driver entry-points, three are unused, and the other five have simple responsibilities. A small skeletal driver (file *my_rad.c*) is available to provide default functions for these other entry points, as well as templates and example code for the three functions discussed above.

The arguments and responsibilities for the remaining device driver entry points are described in the following. The result, RadioReturned from each function are defined by the, RadioReturn codes discussed previously.

```
uint32    (*DevInitFp)    ( uint32 version, void * protoDev,  void * initData,
                            DevSigFp sig,  DevCntrlBlk * control );
```

This prototype defines the radio device's initialization function. This is the only function known by name to the protocols. The actual prototype to this function (which must match the above prototype) should be included in the radio's device-specific header file. The *version* argument gives the protocol's idea of what version of the API is being used. The *protoDev* argument is the protocol's opaque handle, to be used in subsequent calls to the signal callback function. The *initData* argument is not currently used for Radio Device API devices, but is available for radio-specific extensions. The *sig* argument is a pointer to the protocol's signal callback function. The *control* argument points to the device's "control block" which contains pointers for each of the device's entry-point functions. The radio device must fill these in with actual pointers to its entry points before, RadioReturning. Also, an entry in this control block must be initialized with the radio's opaque *radioDev* handle. This handle will be passed down by the protocols in each subsequent call to radio entry points. (Refer to the *dev.h* file for details on the device control block*).*

```
uint32    (*DevOpenFp)    ( void * radioDev, bytep name, bytep mode );
uint32    (*DevCloseFp)   ( void * radioDev );
```

The open and close entry points are used to activate and deactivate the device. They should perform whatever radio-specific operations are necessary. Also, the *name* argument should be stored, and used for future RadioVarName variable queries. The *mode* argument is not used.

```
uint32    (*DevIdleFp)    ( void * radioDev );
```

The idle entry point is called when the protocols are idle. This permits the radio's device driver to perform "idle-time" processing, if any.

```
uint32    (*DevReadFp)    ( void * radioDev, bytep buffer, uint32 * len );
uint32    (*DevWriteFp)   ( void * radioDev, bytep buffer, uint32 * len );
uint32    (*DevFlushFp)   ( void * radioDev );
```

The read, write, and flush entry points are not used for radio devices. Their functions pointers should be set to (void *) 0 in radio devices' control block.

Table 4: Generic Device Driver Prototypes

Return	Name	Arguments
	Device Initialization	
uint32	(*CpwDevInitFp)	(uint32 version, void * protoDev, void * initData, CpwDevSigFp sig, CpwCntrlBlk * control);
	Signal callback	
void	(*CpwDevSigFp)	(CpwDevice device, uint32 num, uint32 qual, void * data, uint32 dataLen, uint32, RadioRetCode);
	Device Entry points	
uint32	(*CpwDevOpenFp)	(void * radioDev, bytep name, bytep mode);
uint32	(*CpwDevCloseFp)	(void * radioDev);
uint32	(*CpwDevReadFp)	(void * radioDev, bytep buffer, uint32 * len);
uint32	(*CpwDevWriteFp)	(void * radioDev, bytep buffer, uint32 * len);
uint32	(*CpwDevFlushFp)	(void * radioDev);
uint32	(*CpwDevCmdFp)	(void * radioDev, uint32 num, uint32 qual, void * data, uint32 dataLen);
uint32	(*CpwDevVarFp)	(void * radioDev, uint32 num, uint32 qual, void * data, uint32 dataLen);
uint32	(*CpwDevIdleFp)	(void * radioDev);

Table 5: Radio Device Qualifier Macros

Radio Device Macro	Description
RADIO_QUAL_GET	Specifies a *get* variable operation
RADIO_QUAL_SET	Specifies a *set* variable operation.
RADIO_QUAL_INC	Specifies a *increment* variable operation.
RADIO_QUAL_CLR	Specifies a *clear* variable operation.
RADIO_QUAL_INFO	Specifies a "info" variable operation.
RADIO_QUAL_ISR	Specifies that the signal function is being called as a "software" signal (otherwise it's being called in a hardware interrupt).
RADIO_QUAL_XMT	Specifies that the operation or signal refers only to the output (xmt, write) direction.
RADIO_QUAL_RCV	Specifies that the operation or signal refers only to the input (rcv, read) direction.
RADIO_QUAL_CH_ENCODE (index)	Returns a qualifier encoded with the specified channel number.
RADIO_QUAL_CH_DECODE (qual)	Extracts the channel number value from the qualifier.