API Framework for Internet Radios

Version: 11 September 1998; v2.03

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

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

1 Purpose

Software protocols for advanced wireless 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 Raytheon, 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 API Framework was developed to facilitate both collaboration and independent development of individual modules for these systems. The intent is to allow these modules to be easily integrated, or "mixed and matched," into advanced, wireless networking systems (or *Internet Radios*).

Specifically, this API Framework is intended to:

- Introduce a concise, platform-independent, language and methodology that can be used to define the interface between "upper" and "lower" modules in a system,
- Provide standard methods for permitting module-specific extensions,
- Foster cross-organization collaboration between developers of these modules, and
- Facilitate porting of various modules among multiple platforms.

The API Framework is based on the definition of a set of generic *"primitives"* that can be mapped to various software and hardware implementations, as appropriate for the particular system environment. For a particular API, these primitives define the functional interface between a "lower module" that provides a service, and an "upper module" that is a user of that service. Physical communication of API primitives across an interface is then defined by specification of an implementation mechanism. For example, a C-based software interface implementation is defined by the "Generic Device Driver" specification.[2] Other successful implementations include a mapping to the Unix IOCTL mechanism,[3] and a serial message passing implementation.[4]

In summary, there are three levels of specification for these APIs:

1. *API Framework*; a set of consistent, API-independent tools (including the various types of API "primitives") for defining the APIs (see Section 2).
2. *API Definition*; the implementation-independent description of the API (see Sections 3, 4, and the API-specific definition documents which extend the "Core" APIs in this document).[5]
3. *API Implementation Mechanism*; the description of how any given API Definition can be mapped to a particular physical hardware or software implementation.[6]

---

[1] This work was supported by the U.S. Government. Refer to the *Acknowledgments* section for details.

[2] Available via http://www.rooftop.com , click on Radio Interface.

[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."

[5] The "Radio Device API" document is an example.

2 API Framework

This section introduces the framework used to define the GloMo APIs. It consists of the following three major components:

- Primitives (the basic operations of the API), and
- Qualifiers (flags and other action specifiers that are applicable to many of the primitives),
- Return Codes (status codes returned from the lower module to the upper module).

Implementation mechanisms should provide 32-bit fields for the identification of these primitives and return codes, and also a 32-bit field for the selection of appropriate qualifiers for each operation.

In addition, API definition inheritance, API information structures, and API compatibility are discussed.

2.1 Primitives

The basic information element of each API is called a "primitive". There are four types of *primitives*, as described in the following table and illustrated in Figure 1.

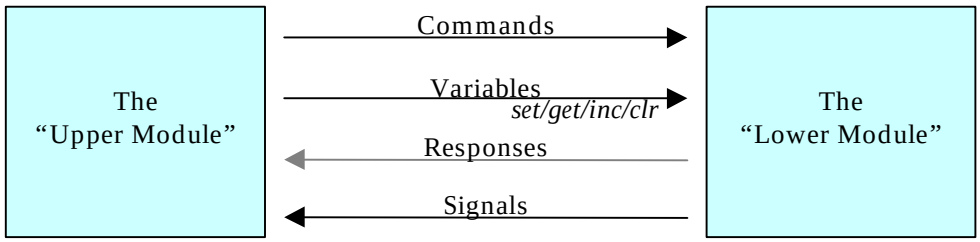

Figure 1: Basic Types of API Primitives

2.1.1 Commands

Commands are asynchronous upper-to-lower module primitives for performing immediate, typically non-persistent actions. Commands often result in an immediate Response followed later by one or more Signals from the lower module.

2.1.2 Variables

Control state characteristics and measurement status information of the lower module are communicated using *Variable* primitives. Variables support one or more of the set, get, increment, or clear synchronous access operations (see Core API Qualifiers in Section 3.1).

Also, *variable groups* allow for the upper module to access a group of variables with one operation. (See "Core API Variables and Variable Groups," Section 3.2.2).

Individual or grouped variable operations that are not associated with some transient event control the "persistent" state of the variable. Certain other transient variable operations (such as those associated with a time slot or a packet transmission or reception, see Section 4.1) control the "transient" state of the variable. The transient state of a variable takes precedence over the persistent state. However, once the event associated with the transient state is no longer active (e.g., the corresponding packet completes transmission), the state of the variable is returned to its persistent state.

2.1.3 Responses

Responses report the synchronous lower-module result to an upper module's command or variable operation. For a software-based implementation (such as the "Generic Device Driver" implementation),

[6] The "Generic Device Driver" document describes how the primitives, qualifiers, and return codes for any API following this framework are mapped to a C-based, function-call software interface.

this is typically handled using the return value from the Command or Variable function call. 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. For Commands, the Response often indicates whether or not the Command has been received correctly and can be acted upon, with one or more Signals reporting the result of the action later.

2.1.4 Signals

Signals are asynchronous lower-to-upper module primitives for reporting recent, typically non-persistent events. The lower module should support the selective enabling and disabling of each of its supported signals through the API. (See the VarSignalEnable Core API variable in Section 3.2.2).

2.2 Qualifiers

Each primitive can be *qualified* to give more specific instructions such as specifying the "channel" or specifying which section (e.g., xmt or rcv) the operation should be applied. Of course, individual qualifiers will only be relevant to lower modules that support the corresponding capabilities.

A special *info* qualifier defined in the Core API is used with variable operations to allow the upper module to learn the capabilities (e.g., read / write support, range of valid values, default value) of the lower module with regards to a particular variable (see Section 3.1.1).

2.3 Return Codes

Each API also defines a set of *return codes* to provide a standard means for the lower module to indicate the success or failure status in each Response to Command and Variable operations, and in each asynchronous Signal delivered to the upper module.

2.4 API Definition Inheritance

The method used to implement an API must allow for each of the above categories of primitives, qualifiers, and return codes to be extended, to permit definition of APIs to modules that inherit and then extend the API to a less-specific (more abstract) module class. For instance, a common header file[7] can be used to assign the numbering ranges for the primitives for each API, as well as assign numeric identifiers for each primitive in these APIs. A constant named "API_END_<*API Name*>_<*Primitive Type*>" can then be used to serve as the starting number for API-specific extension primitives to a particular API, at least until these extension primitives are formally added to the base API class, or until this API extension is formally recognized as a derived class with its own primitive numbering range. Also, locally derived classes can be defined using numbering ranges above the constant "API_START_USER."

The following figure presents the API hierarchy for modules be ng defined to date within the GloMo program following this API framework. (This document only includes the definitions of the "Core API" and the "Core Packet API." Note that unless a module needs particular extensions, it may be implemented using only a "Core" specification. An example is Ethernet and SLIP Framing modules that may be implemented using only the Core Packet API.)

[7] C- and C++-compatible header files with primitive range assignments for various APIs, and with actual primitive assignments for the Core and Radio Device API primitives are available through the www.rooftop.com web-site.

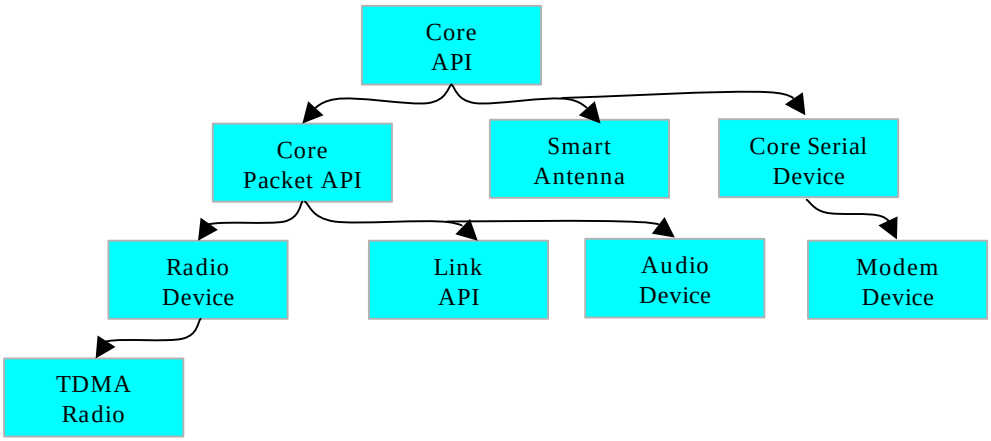

Figure 2: API Inheritance Hierarchy

2.5 API Information Structures

In many cases, API operations involve mainly the passing of an identifier for the primitive (Command, Variable, Signal, or Response), and possibly, one single-word parameter value. However, in other cases, API primitives operations require the communication of an information structure with multiple, possibly multi-word parameters from the upper module to the lower module or vice versa. Examples include communication of packet buffers for transmission or reception, transceiver characteristics to be used for particular transmissions or receptions, or operations that reference multiple variables (e.g., variable "groups"). For these cases, the following rules apply to the API implementation mechanism used to communicate these structures:

- For Command operations, the "access rights" of the API information structure, and any information referenced by it, is passed from the upper module to the lower module. The upper module should make no changes to the contents of this structure while the lower module owns the access rights to it. Access rights to the structure is then returned to the upper module using either a Signal operation (e.g., for packet transmission or reception operations), or a Response to a Command.[8]

- As is the case for the API primitives, return codes, and qualifiers, the implementation mechanism must also define API information structures in a way that facilitates extensions. For instance, a C++-based implementation mechanism could use object "classes" to define information structures and use derived classes for extensions. A C-based implementation could use "structs" for the information structures, and then include these structs at the start of new structs for extensions. For instance a C-based implementation of a packet-based API might define the following information structure:

```
typedef struct {
  uint32    signature;    // Unique ID of this struct type
  bytep     buf;          // points to start of packet buffer
  uint32    bufLen;       // len of data in packet buffer

  ...
} PktInfo;
```

This structure could then be extended to add transmission timing by doing the following:

[8] For software-based implementations, these information structures are typically "owned" by the upper module. Thus, the upper module is responsible for allocating and freeing these packet information structures after access rights have been returned.

```
typedef struct {
  PktInfo    pktInfo;      // Parent's information structure
  uint32     signature;    // Unique ID for this new struct type
  uint32     xmtTime_s;    // Transmit time, seconds
  uint32     xmtTime_us;   // Transmit time, usecs

  …
} TimedPktInfo;
```

In this implementation, the type or extended-type of each information structure can be uniquely identified and verified using the combination of: 1) which primitive the information struct is associated with; 2) the length of the information struct (passed as an argument), and 3) the unique signature in the information struct itself. (Merely association of the information structure with a particular primitive will typically be sufficient when only a parent's base information structure type is being used.)

2.6 Degrees of API Compatibility

The API Framework distinguishes varying degrees of compatibility between upper and lower modules. Differences between modules are categorized into the following three main types:

1. *Derivation conflict* — The upper and lower modules' level of support (in the API's Definition inheritance tree) differ. For example, the upper module can take advantage of radios supporting the "TDMA Radio" API, but the lower module only supports the "Radio Device" API. This can be detected by the upper module using the Core API's "VarClass" variable (see Section 3.2.2).
2. *Version conflict* — The upper and lower modules were created using different versions of the API Definition. This can be detected by the upper module using the Core API's "VarVersion" variable primitive (see Section 3.2.2).
3. *Mechanism conflict* — The upper and lower modules were created using different physical implementation mechanisms. Of course, this will be detected during the process of attempting to integrate the two modules.

In the first two cases, effective operation may still be possible, either by having the upper-module adapt to using only the limited capabilities of the lower-module (when the upper module supports a more recent, or more derivation-specific, API), or by having the upper-module function normally although it would only utilize a subset of the lower module's capabilities (when the lower module supports a more recent, or more derivation-specific, API). In the third case, some development effort would be required. This may involve either:

- development of an "API mapper" to convert and forward the primitive operations between the two API implementation mechanisms, or
- redesign of either the upper or lower module to adopt the implementation mechanism of the other.

3 Core API Definition

This section gives the logical definition of the Core API. It is divided according to the major API components:

- Qualifiers
- Primitives
- Return Codes

3.1 Core API Qualifiers

The following table lists the Core API qualifiers, defined for all APIs.

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	Indicates that the primitive should support *set* or *increment* operations. If only *set* (and not *get*) is specified, then the variable is write-only.
clr	Indicates that the primitive should support *clear* operation. *Clear* will return the state of variables to their default state.
inc	Indicates that the primitive should support the *increment* operation. If only *inc* (and not *get*) is specified, then the variable is write-only. Increment operations instruct the lower module to add the (possibly negative) value passed to the current value. Note that while in a given mode, variables typically support either *set* or *inc* but not both.[9]
info	Used with any variable primitive to retrieve the lower module's capabilities with respect to a particular variable. No other qualifier is permitted in conjunction with the use of the *info* qualifier
isr	Indicates whether running within hardware interrupt or a "foreground" software processor mode in Signal primitives handled as function callbacks.

For variable primitives that support both *get* and *set* (or *inc*) operations, a single variable primitive operation can be used to perform both operations at once. In this case, the operation sets the variable according to the data passed, and returns the value the variable had prior to the set.

3.1.1 The *info* qualifier -- Implementation-dependent Capabilities

The *info* qualifier can be used with any variable primitive to retrieve the lower module's capabilities with respect to a particular variable. No other qualifier is permitted in conjunction with the use of the *info* qualifier. For each *info* operation on a particular variable, the lower module returns the following information:

- The type of variable equal to one of {VarTypeSingle, VarTypeRange, VarTypeTable}.
  - VarTypeSingle variables only have a single value (and thus are typically read-only).
  - VarTypeRange variables can have a range of values, and

[9] There are exceptions, such as initially setting, and then incrementally modifying, the network time through the API to a time-aware packet communication device.

  - VarTypeTable variables may be set to a particular index into a table, corresponding to particular setting for this variable.

- The read / write permissions of the variable equal to one of {VarPermitRead, VarPermitWrite, VarPermitReadWrite}.
- A flag indicating whether this variable can be accessed only through variable group operations on the appropriate group(s) (as is typically the case when a variable's setting is dependent on the setting of one or more other variables).
- The default value of the variable (or default index for VarTypeTable variables).
- The allowed values for VarTypeRange and VarTypeTable variables, specified as follows:
  - For VarTypeRange variables, the minimum and maximum allowed values, and the resolution (or "step-size").
  - For VarTypeTable variables, a list of the allowed settings, indexed (from 0) in the same way that these variables should be referred to in subsequent set or get operations. For example, if the variable "VarXmtPower" can be set to one of the following settings {0, 10, 20, 40} dBm, then setting the variable to 2 will cause the XmtPower to be set to 20 dBm.

3.2 Core API Primitives

This Section lists the Core API primitives, defined for all APIs. Each primitive is labeled with Mandatory, Highly desirable, Desirable, or Optional, indicating the degree of requirement. The "Data" field indicates the generic input and / or output data communicated across the API by each primitive.

3.2.1 Commands

The Core API defines the following command primitives:

CmdReset		Command
	Requirement:	Mandatory
	Qualifiers:	
	Data:	
	Description:	A command used to reset the lower module. The lower module should "play dead" until receiving its first CmdReset command following power-up.
CmdNativeConsole		Command
	Requirement:	Optional
	Qualifiers:	
	Data:	String to be delivered to lower module's "console," and the returned response string.
	Description:	Send a command string to the lower module's native "console" and return the user-response string, if any. The lower module's native console typically refers to the ASCII command interpreter that may be available to the user by connecting a dumb terminal (or terminal emulator) directly to a serial port attached to the lower module.

CmdProcExec Command

Requirement:	Optional
Qualifiers:	
Data:	Diagnostic or other procedure to execute.
Description:	Used to direct the lower module to execute a specific built-in test, or other built-in procedure. The results of the test are returned asynchronously via a SigProcResults signal.

3.2.2 Variables and Variable Groups

The Core API defines the following variable primitives:

3.2.2.1 Name, Version, and SigEnable Variables

VarVersion Variable

Requirement:	Mandatory
Qualifiers:	*get*
Data:	Returned string
Description:	A read-only variable that provides a type and version string of the lower module. This string is typically "hard-wired" into each particular release of this lower module, incrementing the version between releases. The format is an ASCII string of numbers separated by periods, optionally followed by a semicolon and free-form text. For example, `2.0.1; 2 July 1998 rev`

VarName Variable

Requirement:	Mandatory
Qualifiers:	*get*
Data:	Returned string
Description:	A read-only variable that provides the name of this API, possibly appended by an "API instance" name provided by the upper module during initialization or configuration. The format is an ASCII string identifying the API, optionally followed by a semicolon and the name provided by the upper module during initialization: For example, `Radio Device API; Utilicom 2020 driver 1`

VarClass Variable

Requirement:	Highly Desirable
Qualifiers:	*get*
Data:	Class identifier
Description:	A read-only variable that provides the class of this API. The format is API-dependent. For C & C++ APIs, the identifier is a bit-mask with bits indicating the classes that this API is a member of in the inheritance tree. For example, the Radio Device API will be a member of the Radio Device API, Core Packet API, and Core API classes.

VarStatus Variable

Requirement:	Highly Desirable
Qualifiers:	*get*
Data:	Lower-module status
Description:	A read-only variable that provides the lower-module's overall status. A significant change in status can be reported by the lower-module using the *SigStatus* signal.

VarSignalEnable Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set*
Data:	For a set: the Signal code, or SigAll, and TRUE or FALSE to enable or disable the signal. For a get: the Signal code with TRUE or FALSE being returned.
Description:	Allows the upper module to enable or disable individual signals, or to check on the enabled state of signals. The special SigAll Signal code can be used to enable or disable all signals at once.

3.2.2.2 Variables for Managing Variable Groups

Variable groups permit more efficient and convenient reference to lists of API variables. Variable groups are composed of lists of {variable name, value} pairs. Each variable group belongs to a particular variable group *class*. Each variable group class has a unique numeric identification and name (accessible using VarGroupClassName), and specifies the precise set and ordering of the individual variables that make up variable group instances of that class (or "class instances"). VarGroupClassSize gives the number of variables that comprise this variable group class. Individual variable groups are then identified using the variable group class ID along with the appropriate variable group class instance number, which ranges from 0 to VarGroupClassInstances (for that variable group class) – 1. Two class instances are always defined for every variable group class, and are not included in the above VarGroupClassInstances count:

- VarGroupInstanceDefault: a read-only group that lists the default values of all the variables in the specified variable group class.
- VarGroupInstanceCurrent: a read/write group (for classes that permit write operations) that lists or sets the current "persistent" state of the variables in this group.

A variable group class that supports only the Default and Current class instances will have VarGroupClassInstances equal to 0. There are three main uses for variable groups:

1. To provide a convenient way to access or select the (often long) list of behavior and/or configuration characteristics of the lower module. These group classes are defined only by the lower module.

   For instance, for a radio device, a read-only behavior group class might include the time required to change frequencies (RadioVarFreqChangeDelay), and the typical time required to obtain preamble synchronization to an incoming packet (RadioVarPreambleSyncTime); and a read/write configuration group class might include the modulation selection (RadioVarModulationType) and the scrambling algorithm selection (RadioVarScramblingMode). Also, multiple instances of a configuration group class may be

pre-defined by the lower module to allow the upper module to easily select a particular configuration.

2. To group mutually dependent variables into a read-only variable group class, with the individual instances representing the combinations of settings for this variable group. These group classes are also only defined by the lower module.

   For instance, for a radio device, the transmit bit rate (RadioVarBitRate), direct-sequence chipping rate (RadioVarCodeRate), and the number of chips per bit (RadioVarCodingGain) may be inter-dependent variables. Using groups, the specific radio device API can provide a variable group class called something like "Radio Group Xmt Rates" which consists of these three variables, and then provide a set of class instances that specify the permitted alternatives.

3. To permit the upper module to dynamically define custom variable group classes, and class instances, to permit more efficient reference to a related list of variable settings with one operation. These group classes are defined only by the upper module.

   For instance, for network protocols controlling a radio device, the protocols may determine an optimal set of radio transmit parameters to use when communicating with each of it's neighboring network nodes. This parameter list might include transmit power (RadioVarXmtPower), error-correcting code rate (RadioVarFecRate), and direct-sequence spreading code (RadioVarCode). By defining a variable group class (using VarGroupClassDefine) named "Radio Group Neighbor Xmt Characteristics", and then setting individual instance of this class for each of its neighbors (using VarGroupValues), the protocols can easily switch to the proper parameters simply by selecting the proper instance of this new variable group class (using VarGroupSelect).

VarGroupDefineNumMax is a read-only variable that returns the maximum number of dynamically-defined groups supported by the lower module.

VarGroupSelect Variable

Requirement:	Highly Desirable
Qualifiers:	*set*
Data:	groupClassId, groupInstanceNum
Description:	Commands the lower module to switch to the variable characteristics specified by the identified variable group. The group instance number may be an integer from 0 to VarGroupClassInstances - 1, or one of VarGroupInstanceDefault, or VarGroupInstanceCurrent. This *VarGroupSelect* primitive can be used anywhere a single-value variable primitive can be used across the API. Thus, the API implementation must encode the {groupClassId, groupInstanceNum} in a way to permit it to be passed in place of the value for primitive operations.[10]

VarGroupValues Variable

Requirement:	Highly Desirable
Qualifiers:	*get* (and *set* for writeable groups)

[10] For example, the Generic Device Driver implementation mechanism encodes the class & instance into a 32-bit integer using the upper 16 bits for the class ID and the lower 16 bits for the instance number.

Data: groupClassId, groupInstanceNum and the list of {variable name, variable value} pairs corresponding to this class.
Description: Although the variable names are actually not required (since the variable class defines this list and ordering of variables), they are included here as a consistency check.

VarGroupClassName Variable

Requirement: Highly Desirable
Qualifiers: *get*
Data: string
Description: Returns the name of this variable group class.

VarGroupClassSize Variable

Requirement: Highly Desirable
Qualifiers: *get*
Data: Input the groupClassId and returns the number of variables in this variable group class.
Description: Used to retrieve the number of variables in (i.e., the size of) the specific variable group class.

VarGroupClassInstances Variable

Requirement: Highly Desirable
Qualifiers: *get* (and *set* for dynamically-defined classes)
Data: Input groupClassId, and returns the number of instances for this class.
Description: Used to retrieve the number of variable groups defined for this variable group class.

Optional variable group primitives to support upper-module definition of new variable groups include the following:

VarGroupClassDefine Variable

Requirement: Optional
Qualifiers: *set*
Data: groupClassId, className, classSize, classInstances, and the list and ordering of variables that define this variable group class.
Description: groupClassId may be an integer from 0 to VarGroupDefineNumMax.

VarGroupDefineNumMax Variable

Requirement: Mandatory
Qualifiers: *get*
Data: The maximum number of dynamically-defined groups
Description: The lower module returns the maximum number of dynamically-defined groups supported by the lower module.

3.2.3 Signals

The Core API defines the following signal primitives:

SigAll Signal

Requirement:	Mandatory
Qualifiers:	
Data:	
Description:	A special Signal code used only in conjunction with VarSignalEnable variable operations. (See description of VarSignalEnable.)

SigError Signal

Requirement:	Mandatory
Qualifiers:	*isr*
Data:	A number indicating the error, as defined in the lower-module-specific header file.
Description:	A signal indicating that a lower-module error has occurred.

SigStatus Signal

Requirement:	Highly Desirable
Qualifiers:	*isr*
Data:	A number indicating the new lower-module status.
Description:	A signal indicating that a significant (but "non-Error") change has occurred to the status of the lower-module. The lower-module status can also be queried at any time by the upper-module using the *VarStatus* variable.

SigProcResults Signal

Requirement:	Desirable
Qualifiers:	*isr*
Data:	Procedure results
Description:	A signal generated at the completion of a diagnostic test, or other built-in procedure. The procedure may have been executed as a result of a CmdProcExec command, or due to some other event such as built-in tests executed upon power-up.

3.2.4 Summary of Core API Primitives

Table 1 summarizes the names, degree of requirement (M-Mandatory, H-Highly desirable, D-Desirable, O-Optional), qualifiers, and data for each of the Core API's logical primitives.

Table 1: Summary of Core API Primitives

Commands	Rqmt	Qualifiers	Data
CmdReset	M		
CmdNativeConsole	O		User cmd string and response string.
CmdProcExec	O		Diagnostic, or other procedure, to exec.

Variables	Rqmt	Qualifiers	Data
VarVersion	M	*get*	Version string (e.g., "2.0.1; 2 July 1998")
VarName	M	*get*	Name string given to lower module
VarClass	H	*get*	API Class identifier
VarStatus	H	*get*	Lower-module status
VarSigEnable	H	*get/set*	Signal number to enable or disable
VarGroupSelect	H	*set*	groupClassId, groupInstanceNum
VarGroupValues	H	*get (set)*	groupClass & instance, {var, value} pairs
VarGroupClassName	H	*get*	Group class name string
VarGroupClassSize	H	*get*	Group class size (number of variables)
VarGroupClassInstances	H	*get*	returns # of instances given groupClassId
VarGroupClassDefine	O	*set*	groupClassId, name, size, instances, var list
VarGroupDefineNumMax	O	*get*	# of dynamically-defined groups supported

Signals	Rqmt	Qualifiers	Data
SigAll[11]	M		
SigError	M	*isr*	Number indicating the error.
SigStatus	H	*isr*	Number indicating the new status.
SigProcResults	D	*isr*	Results of a CmdProcExec.

3.3 Core API Return Codes

The following table lists the Core API Return Codes, defined for all APIs.

[11] Used by protocols in conjunction with VarSignalEnable to enable or disable all supported signals at once; never generated by the lower-module.

RetOk	Operation accepted or successfully performed.
RetFail	General request failure.
RetNoInit	Lower module not initialized.
RetTimeOut	Request timed out.
RetMemOut	Lower module is out of memory.
RetHwFail	Hardware failure. A ***strong*** suggestion that the upper module should issue a CmdReset command.
RetInvVersion	Invalid API version number (or version not supported)
RetInvInitData	Invalid initialization data
RetInvCtlBlockPtr	Invalid control block pointer (e.g., used for generic device driver implementations)
RetInvState	Operation not permitted in current state.
RetInvCmd	Invalid command or command not implemented by this lower module.
RetInvVar	Invalid variable or variable not implemented by this lower module.
RetInvSig	Invalid signal or signal not implemented by this lower module.
RetInvDev	Invalid "device" pointer (used for context by some implementations)
RetInvPtr	Invalid pointer argument.
RetInvSize	Invalid size argument.
RetInvQual	Invalid qualifier.
RetInvParam	General invalid parameter error.
RetInvGroupClass	Invalid group class identifier.
RetInvGroupInstance	Invalid group instance number.

4 Core Packet API Definition

This section gives the logical definition of the Core Packet API. It too is divided according to the major API components:

- Qualifiers
- Primitives
- Return Codes

Because the primitives, qualifiers, and return codes in the Core Packet API extend those defined in the Core API, the Core Packet API is said to *inherit*, or *be derived from*, the Core API.

Also, to outline how packets are communicated across the Core Packet API, this section begins with a discussion on Packet Handling.

4.1 Packet Handling

The following objectives guided the definition for the handling of data packets across Core Packet APIs:

- Simplify the job of the lower module to the extent possible.
- Avoid packet copy operations.
- Support the use of standard, serial-communications controllers within the "lower-module" that use arrays of pointers to contiguous "frame buffers".[12]

Core Packet APIs communicate user data in the form of packet buffers, identified by a start pointer and a length. Though not a strict requirement, the lower module should generally be able to handle queues of such packet buffers on both the transmit and receive sides. Each individual packet buffer transmitted down across the source API should, if communicated properly, be passed up across the recipient's API as the same, undivided, individual packet (e.g., rather than merging packets together or passing up a series of portions of packets).

For software-based implementations, all packet buffers are "owned" by the upper module (as are any packet information or other structures used to pass information from the upper module to lower module). Thus, the upper module is responsible for allocating, freeing, and informing the lower module of the identities of packet buffers and packet information structures (see below) used for transmit and receive operations. The "access rights" for these packet buffers are passed to the lower module with a *CmdXmtPkt* or *CmdRcvPkt* operations. Asynchronous signals (described below) are then used to return the access rights to the transmit and receive packet buffers and the accompanying packet information structures back to the upper module. Also, the lower module must accept and store an individual *protocol buffer handle* with each packet buffer, to be returned to the upper module when its associated packet buffer is returned through a signal (e.g., upon completing a packet transmission or reception). The upper module can use this protocol buffer handle to hold buffer-specific context information for each packet buffer passed down to the lower module.

To allow the upper module to set transmit or receive characteristics to be used specifically for a particular packet, and to permit the lower module to report packet-specific measurements for receptions, a packet information structure should accompany each packet transmit and receive command (CmdPktXmt, CmdPktRcv), and their corresponding signals (SigPktXmt, SigPktRcv). The packet information structure should include fields for the following:[13]

[12] Implementations for such controllers are available in IC's, ASIC modules, and controllers on embedded microprocessors such as Motorola's 68360.

[13] See the DevPktInfo structure in the Generic Device Driver implementation for an example.

InfoPkt API Information Structure

Description: Used to manage buffers and transmit/receive characteristics for packet transmissions and receptions.

Field Name	*Field Description*	*Software Example*
Handle	Upper-module buffer handle	Opaque software pointer
Pkt buffer	Packet buffer information	Packet start pointer and length
Address	The address to use for the destination of transmitted packets, and to report the source of received packets (if supported by the lower module),	Src/Dst MAC address
Error status	Bit error and/or correction status	Enumerated error status
Timestamps	The precise transmit time for transmitted packets (reported by the lower module), and the precise transmit time (stamped in the packet by the transmitter) and receive time (reported by the lower module) for received packets.	Transmit timestamp in {secs, usecs}, and receive timestamp in {secs, usecs}.
Modifiers	A list of (variable, value) pairs to permit packet-specific tuning of transmit characteristics, or to report packet-specific measurements on received packets	Array of variable {name, value} pairs.

The following precedence is used by the lower module to determine the current settings for the transmit or receive module 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 state variables.

4.2 Core Packet API Qualifiers

For Core Packet APIs, the following qualifier is added to those defined for Core APIs:

xmt/rcv	Indicates that the primitive should be supported for both the transmitter and receiver sections individually, for lower modules that can support it.

4.3 Core Packet API Primitives

The tables in this subsection list primitives that extend the Core API primitives for all APIs that communicate information by data packets or buffers.

4.3.1 Commands

The following modifications or extensions are made to commands inherited from the Core API:

CmdReset	*Core API Command*
Requirement:	Mandatory
Qualifiers:	
Data:	
Description:	Any receive or transmit packet buffers should be returned to the upper module using the SigRcvPkt and SigXmtPkt signals with the RetPktRcvFail or RetPktXmtFail return code. If performed through a function call, the function should "block" until the reset operation has completed. The lower module should "play dead" until receiving its first CmdReset command following power-up.

This Core Packet API introduces the following commands:

CmdXmtPkt	Command
Requirement:	Mandatory
Qualifiers:	
Data:	A packet buffer and its associated protocol buffer handle.
Description:	Command to transmit a packet.

CmdRcvPkt	Command
Requirement:	Mandatory
Qualifiers:	
Data:	A packet buffer and its associated protocol buffer handle.
Description:	Command to pass a buffer to the lower module to be used for received packet data.

4.3.2 Variables

The following variables are introduced by this Core Packet API:

VarQPkts	Variable
Requirement:	Highly desirable
Qualifiers:	*get, xmt/rcv*
Data:	Returned number of packets.
Description:	A read-only variable indicating the current number of packets queued for transmission (default) or reception, depending on the xmt/rcv direction in the qualifier.

VarMacAdr Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set/clr*
Data:	The MAC address or MAC address / mask pair, and the MAC address index (if VarMaxMacAdrs is greater than 1).
Description:	This variable is used to specify the MAC address or MAC address / mask pair(s) for this lower module, or set to the lower-module-specific MAC broadcast address to receive all packets.

VarBitRate Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set, xmt/rcv*
Data:	bit rate
Description:	Raw bit rate on the communication channel.

VarMaxPkts Variable

Requirement:	Highly Desirable
Qualifiers:	*get, xmt/rcv*
Data:	Max number of receive or transmit packet buffers
Description:	A read-only variable indicating the maximum number of internal packet buffers that the lower module can handle for either transmission (maximum number of unsent packets) or reception (maximum number of empty receive packet buffers) depending on the *xmt/rcv* qualifier.

VarTestMode Variable

Requirement:	Highly Desirable
Qualifiers:	*get/set/clr*
Data:	Test mode.
Description:	Used for debugging, this variable indicates the current test mode of the lower module (such as a loopback mode). The test mode must be "disabled" by default after power-up or reset. The modes should be defined within the lower-module-specific header file. For example, different modes may cause a loopback test to occur at different stages within the lower module.

VarMtu Variable

Requirement:	Highly Desirable
Qualifiers:	*get*
Data:	Maximum Transmission Unit (packet buffer size) in bytes
Description:	Returns the maximum size for transmit or receive packet buffers in bytes. This number does not include any header or tailer bytes added to the packet by the lower module (see VarPktHeadLen and VarPktTailLen).

VarPktHeadLen Variable

Requirement: Optional (if not supported, then assumed to be 0)
Qualifiers: *get*
Data: Number of bytes
Description: A read-only variable used to indicate the number of bytes that the lower module requires to be available *before* the start pointer of each packet buffer passed down to the lower module in CmdPktXmt and CmdPktRcv commands. The lower module may use this packet buffer space for adding it's own packet header to outgoing packets, or to receive and process the packet headers from incoming packets. If this variable is not supported, then the upper module will assume that the lower module does not add its own header, or that a buffer copy is done prior to adding its header for outgoing packets (or after processing its header for incoming packets).

VarPktTailLen Variable

Requirement: Optional (if not supported, then assumed to be 0)
Qualifiers: *get*
Data: Number of bytes
Description: A read-only variable used to indicate the number of bytes that the lower module requires to be available *after* the (start pointer + len) of each packet buffer passed down to the lower module in CmdPktXmt and CmdPktRcv commands. The lower module may use this packet buffer space for adding it's own packet tailer to outgoing packets, or to receive and process the packet tailers from incoming packets. If this variable is not supported, then the upper module will assume that the lower module does not add its own tailer, or that a buffer copy is done prior to adding its tailer for outgoing packets (or after processing its tailer for incoming packets).

VarQBytes Variable

Requirement: Optional
Qualifiers: *get, xmt*
Data: Returned number of bytes.
Description: A read-only variable indicating the total number of bytes queued for transmission.

VarMaxMacAdrs Variable

Requirement: Optional
Qualifiers: *get*
Data: Returns the max. number of rcv MAC addresses.
Description: If the lower module can receive packets for multiple MAC address / mask combinations, this will return the max. number. This can be useful for efficient multicast protocols.

4.3.3 Signals

The following signals are introduced by this Core Packet API

SigRcvPkt Signal

Requirement:	Mandatory
Qualifiers:	*isr*
Data:	Rcv'd pkt buffer and associated protocol buffer handle
Description:	A signal generated when a packet has been received. The protocol buffer handle is equal to that used in the corresponding CmdRcvPkt command. This signal is also used, with the RetPktRcvFail return code, to return a receive packet buffer to the upper module before it has been filled in with received packet data (due, for example, to a CmdReset).

SigXmtPkt Signal

Requirement:	Mandatory
Qualifiers:	*isr*
Data:	Xmt'd pkt buffer and associated protocol buffer handle
Description:	A signal generated when a packet has completed transmission. The protocol buffer handle is equal to that used in the corresponding CmdXmtPkt command. This signal is also used, with the RetPktXmtFail return code, to return a transmit packet buffer to the upper module before the data has been actually transmitted (due, for example, to a CmdReset).

4.3.4 Summary of Core Packet API 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 Core Packet API's logical primitives (which are in addition to those defined the Core API).

Table 2: Summary of Core Packet API Primitives

Commands	Rqmt	Qualifiers	Data
CmdXmtPkt	M		Pkt buf & its protocol buf handle
CmdRcvPkt	M		Pkt buf & its protocol buf handle

Variables	Rqmt	Qualifiers	Data
VarMacAdr	H	*get/set, clr*	MAC address / mask for this packet module
VarQPkts	H	*get, xmt/rcv*	No. of packets in queue
VarBitRate	H	*get/set, xmt/rcv*	Raw channel bit rate
VarMaxPkts	H	*get, xmt/rcv*	Max number of packet buffers
VarTestMode	H	*get/set*	Test (e.g., loopback) mode
VarMtu	H	*get*	Max. packet buffer size in bytes.
VarPktHeadLen	O	*get*	Number of bytes
VarPktTailLen	O	*get*	Number of bytes
VarQBytes	O	*get, xmt/rcv*	Total no. of bytes in queue
VarMaxMacAdrs	O	*get*	Max. no. of rcv MAC address / masks

Signals	Rqmt	Qualifiers	Data
SigXmtPkt	M	*isr*	Xmt'd pkt buf & its proto buf handle
SigRcvPkt	M	*isr*	Rcv'd pkt buf & its proto buf handle

4.4 Core Packet API Return Codes

For Core Packet APIs, the following return codes are defined in addition to those in the Core API:

RetPktRcvFail	Packet failed to be received, returning access rights of packet buffer and any info structure. Packet buffer is returned before any reception has been completed for this buffer.
RetPktXmtFail	Packet failed to be transmitted, returning access rights to packet buffer and any info structure. Packet buffer is returned before any transmission attempt has completed.
RetPktXmtFailCarrier	Packet failed to be transmitted due to sensed carrier, returning pkt buf (when in a mode where receive carrier takes precedence over transmissions).
RetPktXmtFailOverflow	Packet failed to be transmitted due to overflow of module xmt queue.
RetPktXmtFailUnderrun	Packet failed to be transmitted due to underrun of module xmt queue (used for modules which are put into persistent "transmit modes").
RetPktRcvError	Other error in received packet; typically passed to the upper module with SigRcvPkt accompanying a received packet buffer with errors detected.
RetPktXmtError	Other error in transmitted packet; typically passed to the upper module with SigXmtPkt accompanying a transmitted packet buffer with some transmission error detected.

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 of the "GloMo Radio API Working Group."