REDLINK API DEVELOPMENT GUIDE

REVISION HISTORY

REV.	ECO NO.	DATE	DESCRIPTION
A	008001	01/04/14	Formal document release.
B	008198	05/16/14	Updated for RCP parameter set 5.1. Added new API functions: rcp_set_uint(), rcp_send(), and rcp_api_get_version
B1	008209	05/16/14	Recreated PDF. No content change.
C	008646	09/05/14	Updated for RCP parameter set 6.0. Added callbacks for new data types. Added ability to customize the API build. Added new parameter properties.
D	009408	04/08/15	Updated for RCP parameter set 6.10. Added many parameters. Added rcp_set_xx_relative functions.
E	009842	07/23/15	Updated for RCP parameter set 6.20. Added two application supplied functions, one composite parameter and hardware dependency section. See API change log for details of some function prototype changes.
F	010194	10/29/15	Updated for RCP parameter set 6.30. Added file transfer functions. Added new client provided functions.
F1	010231	11/05/15	Corrected typo in footer.

TABLE OF CONTENTS

DISCLAIMER

RED® has made every effort to provide clear and accurate information in this document, which is provided solely for the user's information. While thought to be accurate, the information in this document is provided strictly "as is" and RED will not be held responsible for issues arising from typographical errors or user's interpretation of the language used herein that is different from that intended by RED. All safety and general information is subject to change as a result of changes in local, federal or other applicable laws.

RED reserves the right to revise this document and make changes from time to time in the content hereof without obligation to notify any person of such revisions or changes. In no event shall RED, its employees or authorized agents be liable to you for any damages or losses, direct or indirect, arising from the use of any technical or operational information contained in this document.

For comments or questions about content in this document please send a detailed email to rcpsdk@red.com.

COPYRIGHT NOTICE

TRADEMARK DISCLAIMER

OBJECTIVE

The purpose of this document is to describe the structure and usage of the REDLINK® Command Protocol (RCP) Application Programming Interface (API). The API is provided to abstract certain complex aspects of using RCP into more atomic operations. This document covers the high-level usage and key concepts of the API. It is not intended as detailed documentation of the API function calls or the structure of RCP itself. That information is found in other documents listed in the REDLINK SDK Documents table below.

SCOPE

Applies to WEAPON®, RAVEN™, EPIC-X, EPIC-M, and SCARLET-X® cameras that have either MYSTERIUM-X® or RED DRAGON® sensors, with RCP Parameter Set 5.0 and later.

SUPPORTING DOCUMENTS

This document is part of the REDLINK Software Development Kit (SDK), which contains other supporting documents.

REDLINK SDK DOCUMENTS

REDLINK SDK DOCUMENTS	
DOCUMENT	**DESCRIPTION**
REDLINK API Development Guide	This document.
REDLINK Command Protocol: Reference Guide	Documentation of RCP Core protocol and parameters.
REDLINK SDK Source and Reference Applications	Zip archive of: SDK API and Core source code Example REDLINK iOS Application Application for iPhone using the API. Source only. Example REDLINK Android™ Application Application for Android using the API. Source only. Example REDLINK PC Application Qt® based Windows® and OS X® example application for camera remote control using the API. Source and executable. Example RCP Core Application Qt based Windows and OS X example application using only the core functions of RCP, not the API. Source and executable.

CONVENTIONS USED

This document uses hyperlinks to facilitate locating a topic that is usually discussed in a later section. These appear as red text.

The `courier` font is used for literal names or symbols taken from the code and in code snippets.

The term “PC” refers to a desktop or laptop type computer running Windows, OS X, or Linux®.

INTRODUCTION

The RCP API is for use by programmers to interface their applications with a RED DSMC® camera for status and control. This API generates and processes the low level RCP packets, allowing you to operate more naturally at the data level. Direct use of the RCP Core is still available, but the API and its documentation are meant to supplant using RCP directly.

The latest version of the REDMOTE® firmware uses the RCP API.

Throughout this document references are made to specific function calls and data types. This document does not go into details of the calls or data types. The detailed documentation of these is found in the auto-generated documentation in an html format. Use a browser to open the index.html file in the /rcp_api/doc folder.

API OVERVIEW

SDK COMPONENENTS

The API is a portion of the full REDLINK Software Development Kit (SDK). The API is built on top of, and requires the use of the RCP Core, also provided in the REDLINK SDK. The Core is still fully available to an application, but its direct use is discouraged except as described in later sections for use of the cList module. The REDLINK SDK source code is delivered as a zip archive of several folders. The figures 1 and 2 show the preferred usage of the SDK for C/C++ and Java applications. Other languages are not directly supported at this time. If using a language other than C, C++, or Java, it is highly recommended to create a wrapper for the API rather than trying to recreate the functionality from scratch. This will greatly ease the incorporation of updates to the SDK at a later time.

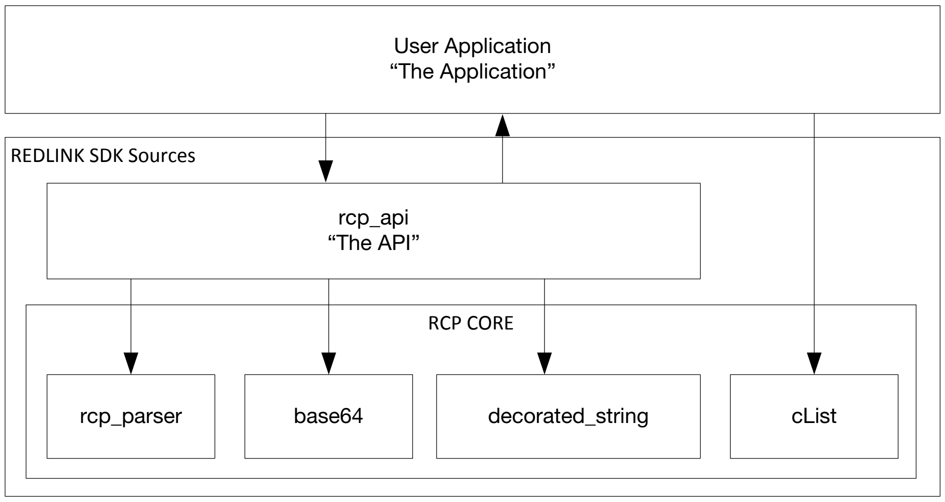


Figure 1. SDK and Application Relationship for C and C++

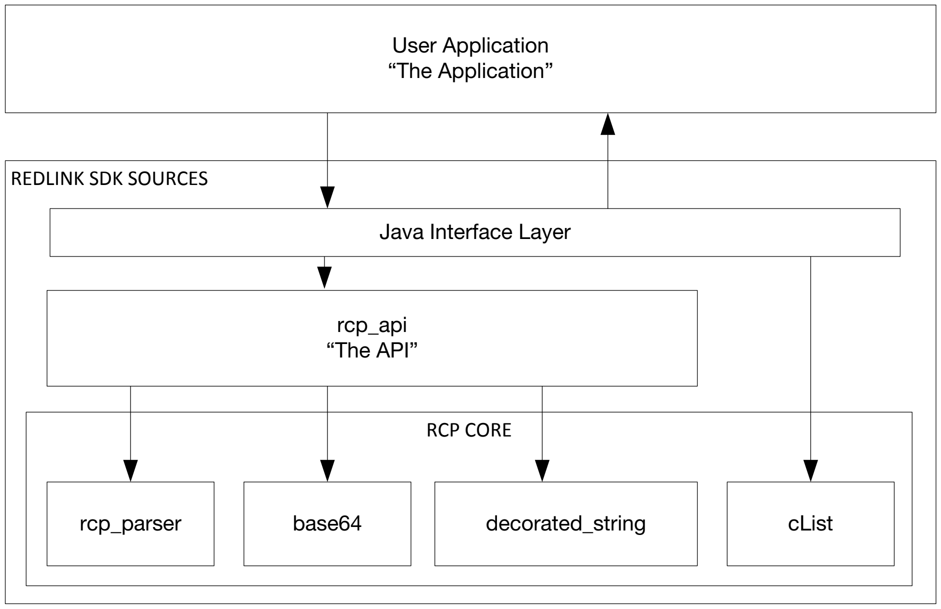


Figure 2. SDK and Application Relationship for Java

PHILOSOPHY OF USE – HOW TO USE IT CORRECTLY

The main purpose of the API is to encapsulate key concepts for using RCP correctly, but it is still possible to ignore these concepts and/or bypass the API. Some key things to keep in mind regarding RCP and the API are presented in the following sections.

MESSAGE TYPES

RCP is based around three essential message types: SET(and SET_LIST, SET_RELATIVE, SET_LIST_RELATIVE), GET(and GET_LIST), and CURRENT(and CURRENT_LIST). These are used respectively to send a value to the camera, request a value from the camera, and send a value from the camera. Various data types from a simple number to complex structures are supported and the API provides functions to facilitate their use. See the REDLINK Command Protocol: Reference Guide document for further details on the underlying protocol.

RCP IS ASYNCHRONOUS

RCP is not a command/acknowledge type of protocol. Agents using RCP can send messages anytime as needed. Sending a SET command to the camera does not result in an acknowledgement of receipt or execution. Executing the SET request generates a CURRENT response if the SET changes a value. You should not wait on a CURRENT message as an indication of any action happening because one may not come.

DRIVE THE UI FROM CURRENT MESSAGES

Camera parameters can be changed in other ways than just SET messages from the application. Changing settings directly on the camera user interface (UI), automatic operations in the camera, or another application also using RCP can change (or modify) them. This means that CURRENT messages can be generated independently of the application. Therefore, the application UI and processing should be driven by CURRENT messages from the camera. For example, do not do things like turn the record indicator on because the user pressed the record button. Instead, send out a message to toggle record in response to the user pressing the record button and turn the record indicator on in response to a CURRENT message from the camera indicating the record state is now `RECORD_STATE_RECORDING`.

THE CAMERA DATA IS KING

For parameters that are displayed to the user as a list of options to select, do not hard code the list in the application. Instead, the list should be retrieved using the GET_LIST command. This is to ensure the proper list is presented to the user based on camera type or other camera settings. It is also important to do this as late as possible when needed since the list may change dynamically. For example, get the list when the user tries to open a drop down not when the UI is initialized.

The camera provides display strings for most camera parameters. When available use the display string sent from the API to display values to the user rather than creating strings manually using the integer value. This will help ensure your application displays values the same way as the camera UI and help keep your application future proof.

THE API CACHES DATA

By default the API caches all data that can be cached. What this means is that as CURRENT messages are received, the API stores the values as well as forwarding them to the application. When an `rcp_get()` call is made, if the API has the latest value for the requested parameter, no GET message is actually sent. Instead the API immediately calls the appropriate application call back function with the cached value. This is particularly useful for applications that jump across tabs or views. They can be refreshed very quickly with minimal traffic to the camera. Internally, the API manages which parameters may be cached and the management of that data. The application should not cache any data sent from the API, instead it should rely on the API caching for efficiency.

To allow a smaller memory usage footprint, the caching feature can be disabled at build time. See API Build Configuration for this and other options.

An important exception to note is that the API does not cache thumbnail data. It is the job of the application to follow a best practices policy of caching thumbnails to avoid repeated transfers of large amounts of data.

THE API MANAGES HARDWARE DEPENDANCIES

The API maintains knowledge of hardware related features and allows the application to check this using the `rcp_get_is_supported()` function. The return for a given parameter is based on firmware version, hardware type, and connected accessories. Always use this feature for determining what should be presented to the user. Do not try to make this determination by other means.

PHYSICAL CONNECTION TO CAMERA

A camera connection is a point-to-point connection used to control and/or monitor the status of a given camera. The physical connection to a camera may be over serial port, wired gigabit Ethernet, or WiFi. The application must provide the creation and management of that connection. The API abstracts the camera connection by requiring the application to provide the callback function `send_data_to_camera_cb()` for the API to transmit to the camera. The API is completely connection agnostic.

The reception of data from the camera is managed directly in the application and requires the received data to be passed to the API in the call `rcp_process_data()`.

CAMERA DISCOVERY

In a network environment there may exist several cameras that can be connected to using the API. If the cameras have REDLINK Bridges, its possible there are multiple network capable interfaces even with just one camera. The Camera Discovery portion of the API is used to find and enumerate all the interfaces on the same sub network as the controlling application. This is not applicable to cameras connected by serial port. It is also an optional step. If the camera's IP address is already known this can be bypassed.

Discovery is implemented using broadcast UDP packets (on port 1112). The application must provide the call back `rcp_broadcast_data_to_cameras_cb()` to perform the UDP broadcasts. It is good practice to make sure you broadcast UDP packets out all the network interfaces on your device. In a separate thread, all incoming UDP data on port 1112 should be sent to the API via the `rcp_discovery_process_data()` function. Starting with RCP parameter set 6.0, the application must extract the interface's IP address from the UDP response header and pass that to the `rcp_discovery_process_data()` function along with the data from the UDP response. The data returned by the camera now also includes an enum of the interface that owns that IP address. This will be in the discovery list data. The application can then differentiate between a wireless connection to a REDLINK Bridge and a wired Ethernet connection on the same camera. Note that the location of the IP address in the discovery list data has moved. Check the auto generated documentation for details.

Psuedocode of one way to discover all the cameras on the network is shown below. It is possible the application may also continuously look for cameras and periodically provide an updated list. Figure 3 shows the interactions of the discovery process in a sequence diagram.

```
rcp_discovery_start()
rcp_discovery_process_data() for any received responses
delay(RCP_DISCOVERY_STEP_SLEEP_MS milliseconds)
loop (RCP_DISCOVERY_STEP_LOOP_COUNT times)
      rcp_discovery_step()
      rcp_discovery_process_data() for any received responses
      delay(RCP_DISCOVERY_STEP_SLEEP_MS milliseconds)
end_loop
rcp_discovery_get_list()
copy(data in list  to local memory)
rcp_discovery_free_list()
rcp_discover_end()
```

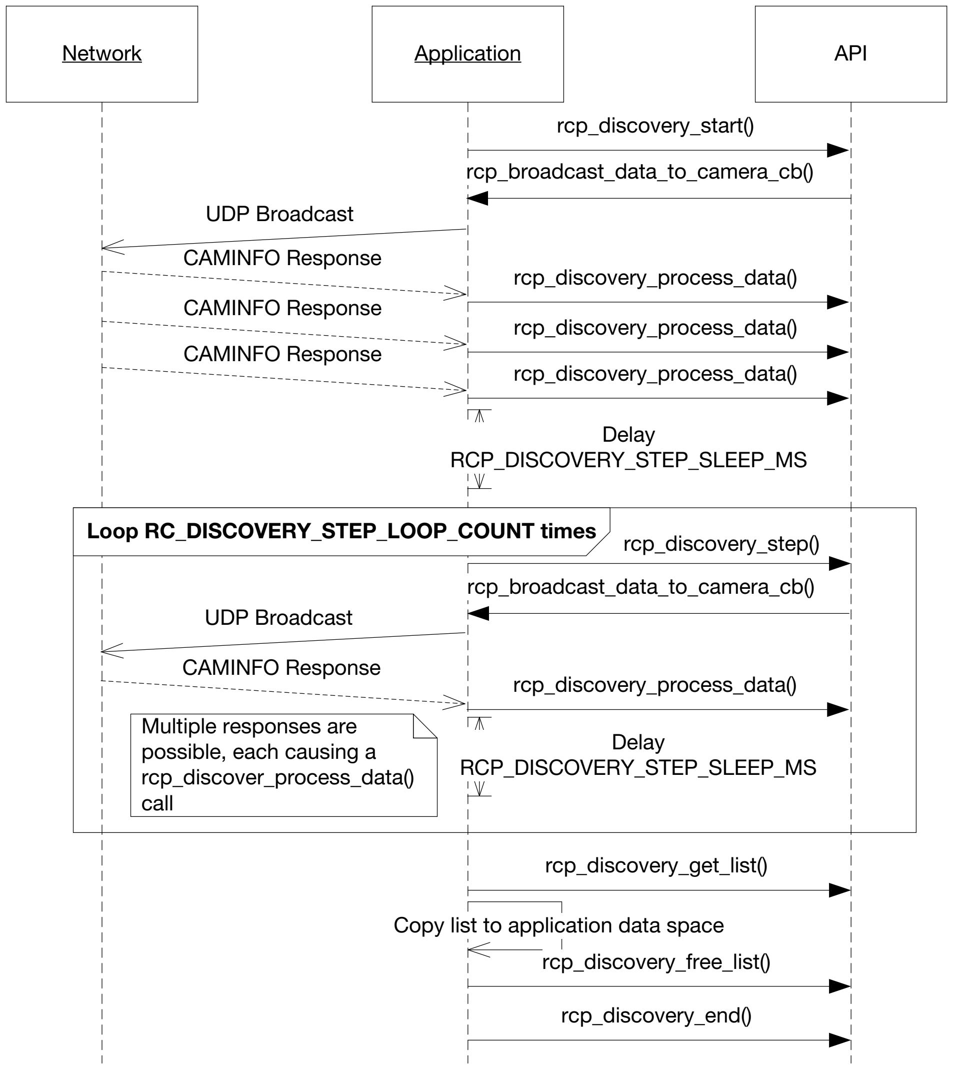

Network
Application
API
rcp_discovery_start()
rcp_broadcast_data_to_camera_cb()
UDP Broadcast
CAMINFO Response
rcp_discovery_process_data()
CAMINFO Response
rcp_discovery_process_data()
CAMINFO Response
rcp_discovery_process_data()
Delay
RCP_DISCOVERY_STEP_SLEEP_MS
Loop RC_DISCOVERY_STEP_LOOP_COUNT times
rcp_discovery_step()
rcp_broadcast_data_to_camera_cb()
UDP Broadcast
CAMINFO Response
rcp_discovery_process_data()
Multiple responses are possible, each causing a rcp_discover_process_data() call
Delay
RCP_DISCOVERY_STEP_SLEEP_MS
rcp_discovery_get_list()
Copy list to application data space
rcp_discovery_free_list()
rcp_discovery_end()

Figure 3. Discovery Process Sequence

CONNECTION CREATION

When creating a camera connection, the physical connection to the camera must first be established. Once this is done a call to `rcp_create_camera_connection()` may be made. When creating the connection, a number of callback functions must be provided to allow the API to communicate information back to the application code. See `rcp_camera_connection_info_t` for the list of callbacks required. That structure must be populated before calling `rcp_create_camera_connection()`. The `send_data_to_camera_cb()` callback will be used to send data out to the connected camera. It is the application's responsibility to send this data out the appropriate serial port or TCP/IP socket. Furthermore, all incoming data from the camera on the connection must be sent to the API via the `rcp_process_data()` function, passing in the camera connection object (which is returned from `rcp_create_camera_connection()`). Note that with multiple instances of camera connection objects (`rcp_camera_connection_t`) it is possible to connect to more than one camera (or interface) concurrently.

After creating a camera connection do not send any get or set requests to the camera until after the `RCP_CONNECTION_STATE_CONNECTED` state has been reached. The callback `state_cb()` is used by the API to report the connection state. Assuming a successful connection, this is where the application would begin communication with the camera. Unsuccessful connection conditions are discussed in the next section.

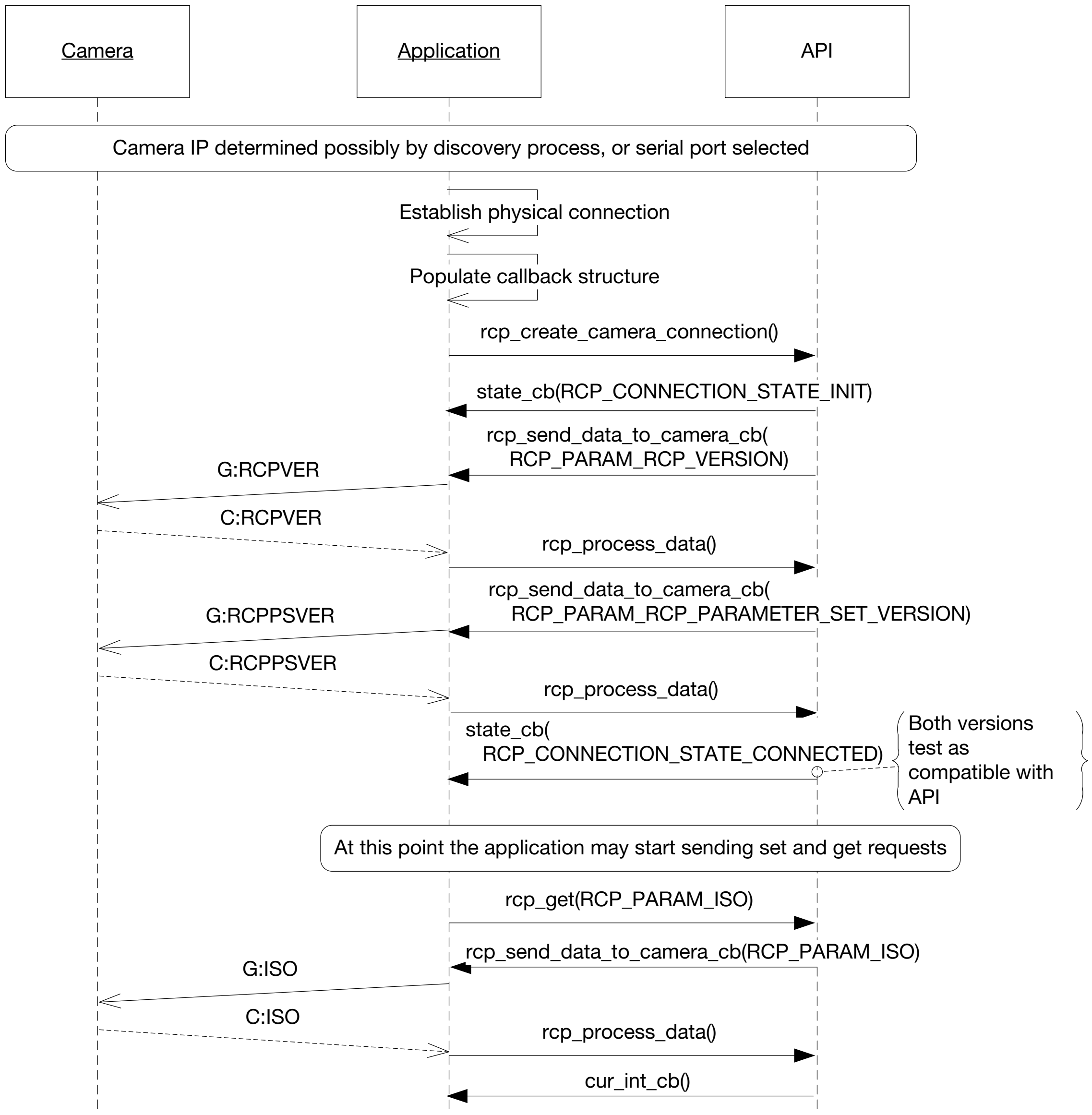

Camera
Application
API
Camera IP determined possibly by discovery process, or serial port selected
Establish physical connection
Populate callback structure
rcp_create_camera_connection()
state_cb(RCP_CONNECTION_STATE_INIT)
rcp_send_data_to_camera_cb(RCP_PARAM_RCP_VERSION)
G:RCPVER
C:RCPVER
rcp_process_data()
rcp_send_data_to_camera_cb(RCP_PARAM_RCP_PARAMETER_SET_VERSION)
G:RCPPSVER
C:RCPPSVER
rcp_process_data()
state_cb(RCP_CONNECTION_STATE_CONNECTED)
Both versions test as compatible with API
At this point the application may start sending set and get requests
rcp_get(RCP_PARAM_ISO)
rcp_send_data_to_camera_cb(RCP_PARAM_ISO)
G:ISO
C:ISO
rcp_process_data()
cur_int_cb()

Figure 4. Camera Connection Sequence

Once a camera connection is established the camera continuously sends out CURRENT messages, which in turn result in the appropriate callback calls being made to the application. For example, if the user changes the ISO parameter using the camera UI from 800 to 1000 the application will receive an integer type callback with the parameter `RCP_PARAM_ISO`, value of 1000, and an appropriate string to display (in general it is best to use this string generated by the API rather than creating your own from the integer value of the parameter). Periodic data is also continuously sent from the camera (such as `RCP_PARAM_TIMECODE`, `RCP_PARAM_POWER_VAL`, etc.) without being triggered by a GET or SET. Figure 4 shows the connection sequence.

The version of the API itself is available as a string from the call `rcp_get_api_version()`. This can be useful in an about page of the application.

Starting with RCP version 6.0, connection statistics are available. The function `rcp_camera_connection_stats()` can be called to fill a structure with the number of transmitted packets and total bytes and received packets and total bytes. Values are totals since the connection was established.

CONNECTION ERROR CONDITIONS

Several error conditions are possible when making a connection to the camera.

- An error occurs at the network (or serial port) level creating the physical connection. This must be detected and handled by the application code.
- An error occurs at the network (or serial port) level creating the camera connection. This could indicate an error in the `send_data_to_camera_cb()` callback provided by the application. The API does not access the physical connection. The application must detect these conditions and have the callback return an `RCP_ERROR_SEND_DATA_TO_CAM_FAILED` error code. The API will subsequently set the connection state to `RCP_CONNECTION_STATE_COMMUNICATION_ERROR`. This is applicable at any point there are errors in the physical communication such as loss of connection.
- A successful connection is made, but periodic CURRENT messages never come and the camera does not respond to SET or GET requests. Verify that the Enable External Control check box is set if using Ethernet access, or that the REDLINK Command Protocol is selected if using serial port. The application should be able to detect this by checking for a timeout on reception of periodic data such as the temperature that should be received once per second.
- The API reports the connection state as either `RCP_CONNECTION_STATE_ERROR_RCP_VERSION_MISMATCH` or `RCP_CONNECTION_STATE_ERROR_RCP_PARAMETER_SET_VERSION_MISMATCH`. The application must destroy the connection by calling `rcp_delete_camera_connection()` and destroying any physical connection structures as needed. The application should also alert the user that either the camera or the application might need updating.
- The API reports the connection state as `RCP_CONNECTION_STATE_COMMUNICATION_ERROR`. The application must delete the camera connection and destroy the physical connection. The deletion of the connection in response to the error state should only be performed from the state_cb() call. The application might then choose to try to re-establish the physical connection and recreate the API connection without notifying the user, possibly some limited number of attempts. If this error happens while trying to create the connection, the user should probably be notified that connection cannot be made.

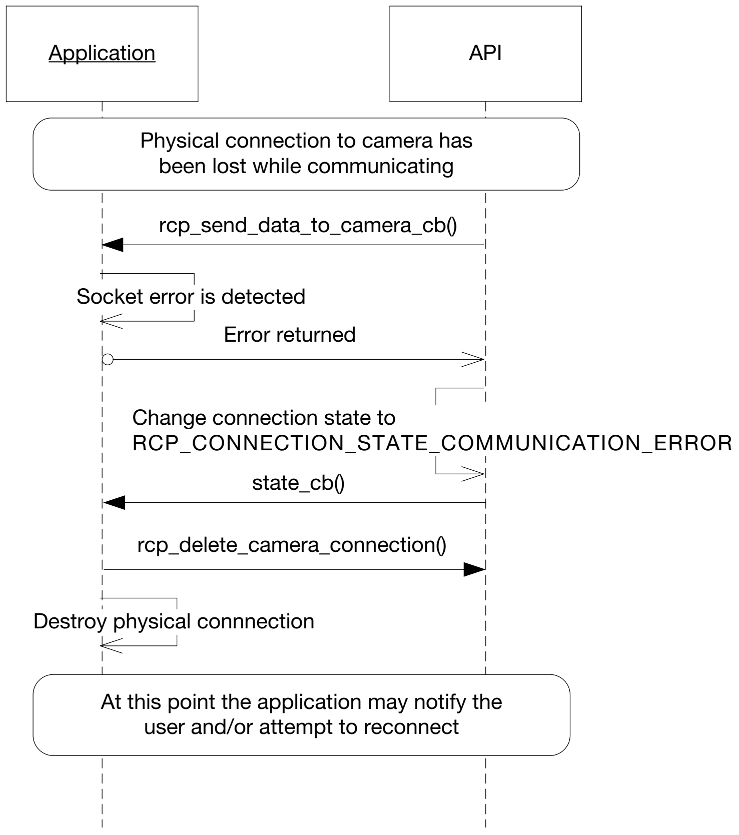

Application
API
Physical connection to camera has been lost while communicating
rcp_send_data_to_camera_cb()
Socket error is detected
Error returned
Change connection state to RCP_CONNECTION_STATE_COMMUNICATION_ERROR
state_cb()
rcp_delete_camera_connection()
Destroy physical connnection
At this point the application may notify the user and/or attempt to reconnect

Figure 5. Correct Connection Error Recovery Sequence

CALLBACK FUNCTIONS

Callback functions provide a way for the API to have the application do work for it (send data to the camera) or to notify the application of some event (CURRENT message data available or state change). There is one callback for sending data, one for UDP broadcast, one for signaling connection state, and one each for the various return data types. The application must create and provide the callbacks. With the exception of the UDP broadcast callback (`rcp_broadcast_data_to_camera_cb()`), callback function address and data are all defined in the structure of type `rcp_camera_connection_info_t`. The application must allocate and populate this structure to be passed in the `rcp_create_camera_connection()` call. Each callback allows for a pointer to data to be sent when the API calls the callback. For example, the `rcp_send_data_to_camera_cb()` could put a pointer to the socket it created for the physical connection. Then when the API calls the callback, it sends that pointer as one of the parameters of the call and the application can access the appropriate socket. This is especially useful if the application is managing multiple camera connections.

Note that throughout this document the callback functions are referred to by the names in the `rcp_camera_connection_info_t` structure. These are really just the names of pointers to the callbacks and the application may choose other names as desired.

DATA MANAGEMENT

DATA TYPES

The API is used to set and get the values of camera parameters, which are uniquely identified with the `rcp_param_t` enum. Only parameters defined in `rcp_param_t` can be accessed.

The data associated with a parameter can be one of several types. The possible types are integer, unsigned integer, list, histogram, string, tag, and clip list. Set operations are only needed for integer and string types. Legal values for a parameter either come from an enumerated type in the header file `rcp_types_public.h`, or by parsing lists of values sent from the camera.

Data returning from the camera are handled with callbacks unique to the data type. These are the application supplied callbacks `send_data_to_camera_cb()`, `cur_int_cb()`, `cur_uint_cb()`, `cur_list_cb()`, `cur_hist_cb()`, `cur_str_cb()`, `clip_list_cb()`, `cur_tag_cb()`, `cur_status_cb()`, `notification_cb()`, `cur_audio_vu_cb()`, `cur_menu_cb()`, `cur_menu_node_status_cb()`, `rftp_status_cb()`, and `state_cb()` defined in the `rcp_camera_connection_info_t`.

SETTING/GETTING CAMERA PARAMETERS

Once a camera connection has been established, parameters on the camera are changed by calling `rcp_set_int(), rcp_set_int_relative(), rcp_set_uint(), rcp_set_uint_relative(), rcp_set_str(), rcp_set_list(), rcp_set_list_relative(), or rcp_send()`. Do not assume the value sent will automatically take effect on the camera - there could be conditions that prohibit your setting from taking effect. Instead, display any new values to the user if and when the appropriate current value callback is called.

For example, to toggle record on the camera you can call

```
rcp_set_int(con, RCP_PARAM_RECORD_STATE, SET_RECORD_STATE_TOGGLE);
```

However, do not indicate recording on the application until an incoming CURRENT for `RCP_PARAM_RECORD_STATE` with the value of `RECORD_STATE_RECORDING` has been received.

`rcp_send()` is new with version 5.1 and essentially performs a SET but is used for parameters without a value field, whose mere reception causes some action to be performed. The SHUTDOWN parameter would be an example of this. The API validates the payload for parameters in an `rcp_set_`*`type`*`()` call, so its not possible to use it for a valueless parameter. Passing in even a dummy value for something like SHUTDOWN, which does not take a value, will result in no message being sent to the camera.

When prompting the user to select a new value for a parameter (say ISO), the correct process is to request the list of available values from the camera (rather than hard-coding this list - which may change - in the application). Use `rcp_get_list()` to request the list. The list is returned in the `rcp_cur_list_cb()` callback. Each item in the list is a tuple of {integer value, string}. The string should be presented to the user, and the corresponding integer value is used in a `rcp_set_int()` if that entry is selected. With this model, the application code doesn't know or care how any of the parameter data is stored and interpreted by the camera. Some lists can be edited and customized in the camera. Others may change based on other camera settings (e.g. the sensor frame rate list is dependent on the current record format). Adhering to these guidelines helps ensure your application will continue to work with future camera firmware upgrades.

The update-ability of some parameters can change as a function of camera operating mode. For example, when recording, some settings may not be changed. The `rcp_get_status()` is used to check this condition for a given parameter. If it returns false, then the application should not allow modifying the value and possibly remove it from UI, or grey it out. The API will also push updates to the `cur_status_cb()` when changes in a parameter's status occur.

RCP PARAMETER PROPERTIES

There are five functions for getting properties of a parameter: `rcp_get_label()`, `rcp_get_update_list_only_on_close()`(*deprecated as of RCP v6.x), `rcp_get_is_supported()`, `rcp_get_name()`, and `rcp_get_id()`. These properties are a function of the parameters themselves and not the state or value of the parameters in camera. The name and id properties are only intended to be used when wrapping the API in another language where the `rcp_param_t` enum cannot be used directly.

- label: Human readable name of parameter (this can be used as a label in the UI). For example, the label for `RCP_PARAM_ISO` is "Sensitivity".
- update_list_only_on_close: (deprecated as of RCP version 6.x, use the `update_list_only_on_close` flag in the `cur_list()` callback instead.) Some camera parameters take a non-trivial amount of time to take affect in-camera (e.g. record format). In these cases, it is not advisable to set the new value as the user is navigating through the list. This flag indicates whether the value should be updated while navigating the list or only once the list has been closed. *Deprecated means it is no longer supported and may be deleted in the future without warning.
- is_supported: Check if the given parameter is supported by the connected camera. This function uses the camera's RCP Parameter Set Version, camera hardware type and attached modules to determine if a parameter is available. This should be used to conditionally show newer commands to the user when connecting to older camera builds. As of RCP version 6.x, this call now also fills a structure of properties about the parameter, requiring an additional parameter. If the properties are not required, NULL can be passed in for this.
- name: Stringified version of the `rcp_param_t` enumerated value. For example the name for `RCP_PARAM_ISO` is "RCP_PARAM_ISO".
- id: Actual enumerated value looked up using the name. The id for "RCP_PARAM_ISO" is `RCP_PARAM_ISO`.

RCP PARAMETER PROPERTIES STRUCTURE

A structure of the type `rcp_param_properties_t` is filled in by the `rcp_get_is_supported()` function. This information can be used by the application to tailor how the parameter data is handled. It indicates which kind of calls can be made for the parameter. Consult the auto-generated documentation for further details.

PARAMETER PROPERTIES STRUCTURE

PARAMETER PROPERTIES STRUCTURE		
NAME	**TYPE**	**DESCRIPTION**
name	const char *	Name of parameter

PARAMETER PROPERTIES STRUCTURE

NAME	TYPE	DESCRIPTION
label	const char *	Label for parameter
has_get	int	If true, calls to rcp_get() are valid for this parameter
has_get_list	Int	If true, calls to rcp_get_list() are valid for this parameter
has_get_status	Int	If true, calls to rcp_get_status() are valid for this parameter
has_send	Int	If true, calls to rcp_send() are valid for this parameter
has_set_int	Int	If true, calls to rcp_set_int() are valid for this parameter
has_set_int_relative	Int	If true, calls to rcp_set_int_relative() are valid for this parameter
has_set_uint	Int	If true, calls to rcp_set_uint() are valid for this parameter
has_set_uint	Int	If true, calls to rcp_set_uint_relative() are valid for this parameter
has_set_str	Int	If true, calls to rcp_set_str() are valid for this parameter
has_set_list	Int	If true, calls to rcp_set_list() are valid for this parameter
has_set_list_relative	Int	If true, calls to rcp_set_list_relative() are valid for this parameter
has_edit_info	int	If true, edit info exists for this parameter and will be supplied in the cur_xxxx_callback()
update_list_only_on _close	Int	If true, only set data once list is closed and not as the user scrolls through the list (for the case where the parameter takes a long time to apply the setting)

DISPLAY STRINGS

The API provides display strings to supply what is to be displayed by an application for a textual representation in a user interface. These are provided in the structure returned by most of the cur_<type>_cb() callbacks. Depending on the parameter they may be provided directly or extra calls may be needed to get a list and then decipher the stringified list into usable strings. This feature is provided to avoid the application having to know and understand the underlying implementations such as the enumerated type values and logic in the cases of composite parameters. This also helps future proof the application from changes in the camera implementation.

Two member variables of the return data structure indicate whether the string is supplied or must be fetched with other calls.

- `display_str_valid`: if true, indicates the display string variables have valid strings. Otherwise there is either no string for this parameter, or it must be fetched in a list.
- `display_str_in_list`: if true, indicates the `rcp_get_list()` call must be made for this parameter and additional steps taken.

The display string is provided in four forms, decorated (`display_str` and `display_str_abbr`) and decoded (`display_str_decoded` and `display_str_abbr_decoded`), explained in the next section. There is also a `display_str_status`, which indicates a color to use. All of these are validated by the `display_str_valid` variable. The abbreviated versions are often the same as the full string. Use of the full string is preferred unless there are space constraints that justify using the abbreviated version. An example of an abbreviated string would be the camera's use of "RG3" in the histogram window instead of "REDgamma3".

The display strings can come from multiple callbacks depending on the parameter type. It might be tempting to put code in those call backs to handle updating the application UI based on the parameter ID, but it is not safe to assume the string will always come that way. For example, a string coming now in the `cur_int_cb()` might at a later version of the API come from `cur_str_cb()`. Application specific design should be minimized in the callbacks. It is a better design to simply always pass the string up to another layer that encapsulates the application UI specific information. The supplied example programs are structured this way, their structure should be taken as a best practices example for the application architecture.

DECORATED STRINGS

Decorated strings are a way to indicate where special fonts or symbols should be used to mimic how the camera displays some information. These use HTML style special characters in place of custom symbols. For example the superscripted “1/” used in the exposure time display is represented by a “&red1over;”. So the decorated display string for 1/48 of a second would be “&red1over;48&redsec;”. Decorator markup can occur anywhere in the decorated string (not just be a prefix or suffix).

If the application platform can render the special symbols, the mapping is shown in the table below. The application will need to provide the parsing of the decorated string and perform the symbol replacement.

DECORATED STRING MAPPING

DECORATED STRING MAPPING

DECORATOR MARKUP	DISPLAY SYMBOL	DESCRIPTION
&red1over;	1/	Small superscripted 1/
&redfover;	f/	Small superscripted f/
&redsec;	sec	Small lowercase sec
&rediso;	ISO	Small uppercase ISO
&redkelvin;	K	Subscripted uppercase K
&deg;	°	Degree symbol
&redfps;	FPS	Small uppercase FPS
&redana2;	ANA 2	White ANA above black 2 in white box
&redana13;	ANA 1.3	White ANA above black 1.3 in white box
&redformatk;	K	Subscripted uppercase K
&redae;	AE	AE icon
&redav	Av	AV icon
&amp;	&	Ampersand
&redcheck;	✓	Check mark
&copy;	©	Copyright symbol
&reg;	®	Registered symbol
&trade;	™	Trademark symbol

DECODED STRINGS

Decoded strings use standard characters. If the application cannot or chooses not to use special symbols, the decoded version can be used. The API returns both the decorated and decoded strings for the `cur_<type>_cb()` callbacks, except for the `cur_list_cb()` callback.

The API provides methods to do the decoding so the application does not need to do the parsing and replacement. For the curious, the mapping is shown in below table and can also be found in the source file /decorated_string/decorated_string.c.

DECODED STRING MAPPING

DECODED STRING MAPPING

DECORATOR CHARACTER	STANDARD CHARACTERS
&red1over;	"1/"
&redfover;	"f/"
&redsec;	" sec"
&rediso;	"ISO "
&redkelvin;	"K"
&deg;	" deg"
&redfps;	" FPS"
&redana2;	" ANA 2"
&redana13;	" ANA 1.3"
&redformatk;	"K"
&redae;	" AE"
&redav	" Av"
&amp;	"&"
&redcheck;	" Check"
&copy;	"(C)"
&reg;	"(R)"
&trade;	"(TM)"

LISTS

Lists are the way in which the camera conveys the set of valid selections for a parameter and also indicate the current target setting. The cList class provided in the RCP SDK handles the parsing of the list into a current index selection and multiple pairs of numeric value and display strings. The string portion is what should be displayed to the user, and the numeric portion is what should be sent in an `rcp_set_int()` call for the list index selected by the user. The current index points to the value/string pair that is currently selected as the target value. The "current" value in the list may not match the last display string sent in the `current_int_cb()` callback. For example in the case of RCP_PARAM_REDCODE, the last display string could be "RC 8:1 YELLOW" and the current value in the list is "RC 5:1". This indicates the target of 5:1 could not be achieved.

RCP_GET_LIST()

To get a list, the application should make an `rcp_get_list()` call for the parameter in question. The data for the `cur_list_cb()` contains a string (`list_string`) that needs to be converted and an indication whether the string is valid (`list_string_valid`). There are also the elements `min_val` and `max_val`, qualified by `min_val_valid` and `max_val_valid`. If either is marked as valid, then they represent lower and upper bounds outside which the value in the value/string pairs list entries should be shown in a different color to indicate they are not available. The user should still be able to select values outside min to max as a target setting and call `rcp_set_xxxx()` with the user selected value. With RCP version 6, the list call back data now includes three properties `send_int`, `send_uint`, and `send_str` to indicate which `rcp_set_xxxx()` function is the correct one to use. These are the same as provided by the `rcp_get_is_supported()` function and provided to avoid extra function calls. The `update_list_only_on_close` property is now also provided in the list call back data.

The current index and value/string pairs are retrieved by using the cList class methods `importStringList()` (for decorated strings) or `importStringListAndDecode()` for decoded strings. Pass these functions the string in `list_string`. The cList functions `getNum()` and `getStr()` can then be used to get the value and display string for a given index. The function `length()` can be used to get the number of elements in the list. The function `getIndex()` returns the current index.

If the data returned from `rcp_get()` for a parameter has the `display_str_in_list` element set, then `rcp_get_list()` method must be used to get the display string for that parameter. The function `getCurrentStr()` provides a single call to get the display string of the current index when the entire list does not need to be shown. Figure 6 shows the interactions to do this for autofocus mode.

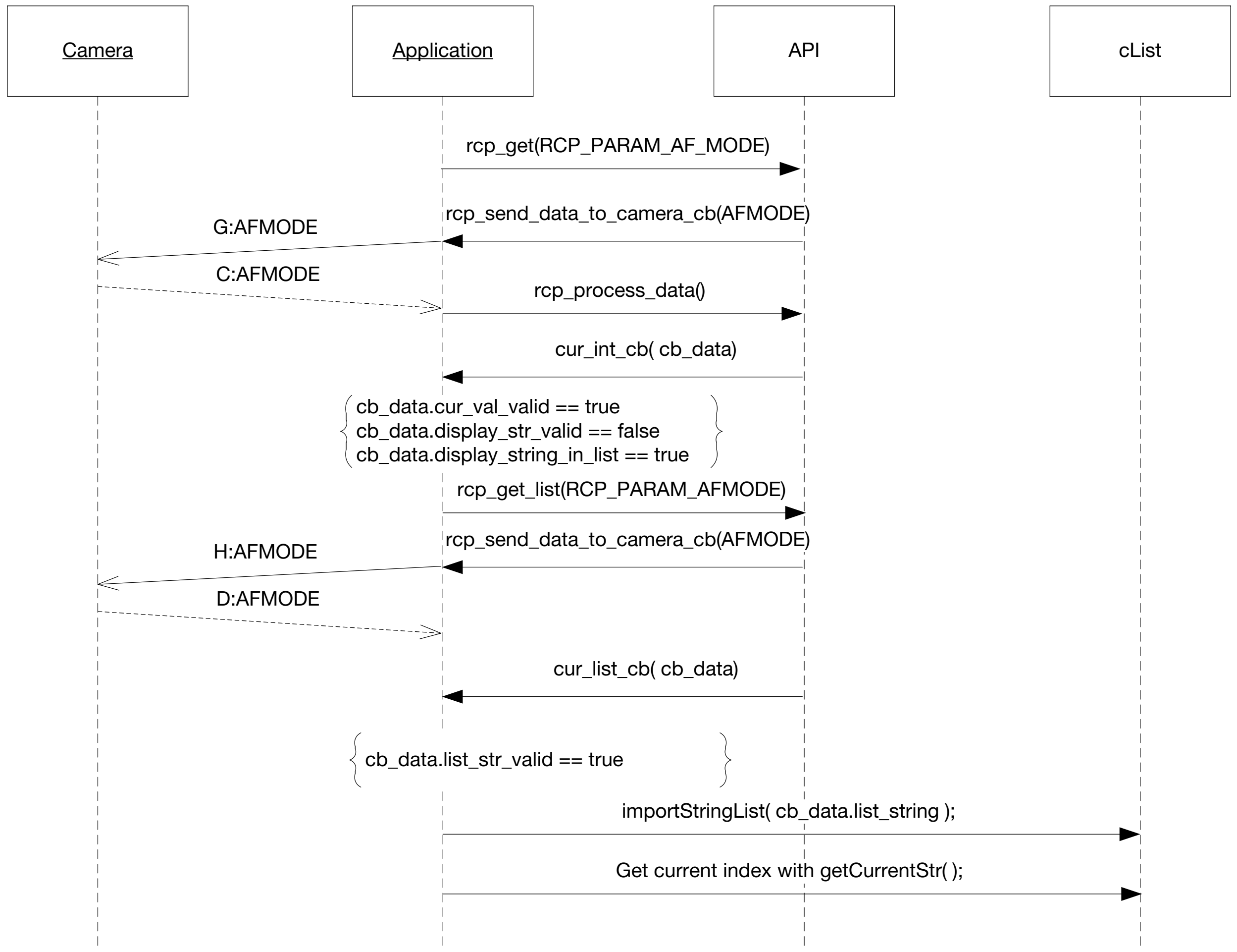

Camera
Application
API
cList
rcp_get(RCP_PARAM_AF_MODE)
rcp_send_data_to_camera_cb(AFMODE)
G:AFMODE
C:AFMODE
rcp_process_data()
cur_int_cb(cb_data)
cb_data.cur_val_valid == true
cb_data.display_str_valid == false
cb_data.display_string_in_list == true
rcp_get_list(RCP_PARAM_AFMODE)
rcp_send_data_to_camera_cb(AFMODE)
H:AFMODE
D:AFMODE
cur_list_cb(cb_data)
cb_data.list_str_valid == true
importStringList(cb_data.list_string);
Get current index with getCurrentStr( );

Figure 6. Getting Display String from List

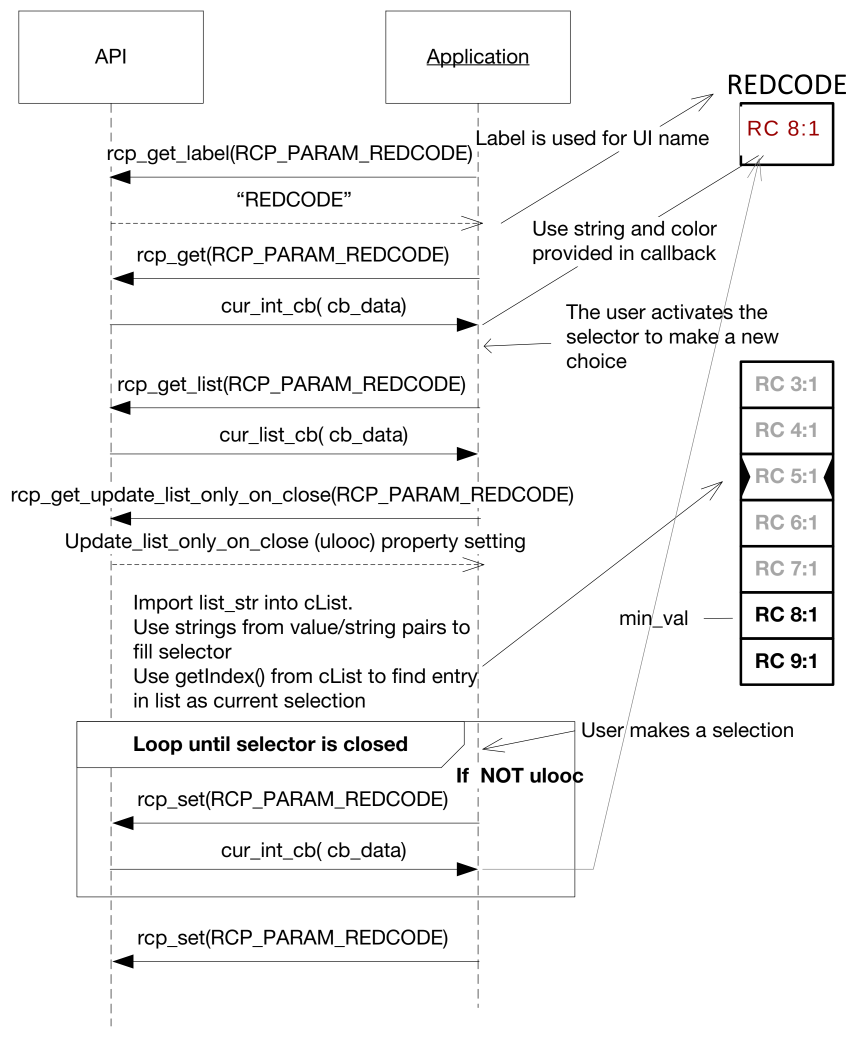
RCP_SET_LIST()

Some parameters (those with the has_set_list property true) will allow their list to be modified directly with the `rcp_set_list()` function. This function accepts a stringified c_list. Don't assume the list will be accepted by the camera. Always rely on the CURRENT messages for the camera's values.

REDCODE EXAMPLE

Figure 7 shows the proper sequence for managing a control that uses a list. In this case the REDCODE setting. Implementing this as shown in the example programs will provide robust, generalized code that can be used for many of the parameters.

This example assumes the camera connection has already been established and that periodic data is being handled at the same time. To simplify the drawing, the camera lifeline has been omitted and the `rcp_send_data_to_camera_cb()` and `rcp_process_data()` calls are not shown.


API
Application
rcp_get_label(RCP_PARAM_REDCODE)
"REDCODE"
rcp_get(RCP_PARAM_REDCODE)
cur_int_cb(cb_data)
rcp_get_list(RCP_PARAM_REDCODE)
cur_list_cb(cb_data)
rcp_get_update_list_only_on_close(RCP_PARAM_REDCODE)
Update_list_only_on_close (ulooc) property setting
Import list_str into cList.
Use strings from value/string pairs to fill selector
Use getIndex() from cList to find entry in list as current selection
Loop until selector is closed
If NOT ulooc
rcp_set(RCP_PARAM_REDCODE)
cur_int_cb(cb_data)
rcp_set(RCP_PARAM_REDCODE)
Label is used for UI name
Use string and color provided in callback
The user activates the selector to make a new choice
User makes a selection
min_val
REDCODE
RC 8:1
RC 3:1
RC 4:1
RC 5:1
RC 6:1
RC 7:1
RC 8:1
RC 9:1

- Initialize the closed selector getting the label and value from API.
- Use the string provided by API and the color specified.
- Do not create display string from numeric value.
- This should only get updated on receiving CURRENT messages, not from other UI interaction.

- Open the selector and populate it using either decorated or decoded strings provided by cList.
- Always validate the presence of min_val and max_val with corresponding _valid fields.
- Show entries outside the range of min_val to max_val in alternate color.
- Do not use value or string from last CURRENT to figure out which entry to mark as selection. It can be different. Current list selection will match the target value which may be different than value in use.
- Logic should not depend on the parameter ID. Done properly, the same code should handle any parameter with similar UI.
- Do not cache any data. The API is doing that for you.
- The API knows that REDCODE is normally set as a target value and takes care of that logic.

Figure 7. REDCODE Selection and List Display Example

COMPOSITE PARAMETERS

Some parameters such as the media capacity or input power are shown in different formats depending on various conditions. The media capacity is shown in percent or minutes. The power is shown in time remaining or volts. These require the logical combination of multiple parameter values. That logic is embedded in the camera, but the components are made available in RCP parameters. The appropriate logic is embedded in the API and the application need only deal with a single data parameter. The API generates the correct display string, getting other parameters as needed. This is another reason to always use the display strings generated by the API.

There are six composite parameters shown in the table below with their component parameters. Use the composite parameter instead of the components to get the display strings for these items.

COMPOSITE PARAMETERS

COMPOSITE PARAMETERS	
COMPOSITE PARAMETER	**USE INSTEAD OF THESE COMPONENTS**
RCP_PARAM_MEDIA_DISPLAY_VAL	RCP_PARAM_MEDIA_VAL RCP_PARAM_MEDIA_TIME_REMAINING RCP_PARAM_MEDIA_DISPLAY_MODE RCP_PARAM_RECORD_MODE
RCP_PARAM_POWER_DISPLAY_VAL	RCP_PARAM_POWER_VAL RCP_PARAM_POWER_TIME_REMAINING RCP_PARAM_POWER_DISPLAY_MODE
RCP_PARAM_MEDIA_DISPLAY_LABEL	RCP_PARAM_RECORD_MODE RCP_PARAM_MEDIA_LABEL
RCP_PARAM_HDR_MODE	RCP_PARAM_RECORD_HDR_MODE RCP_PARAM_PLAYBACK_STATE RCP_PARAM_PLAYBACK_NUM_HDR_TRACKS
RCP_PARAM_HDR_MODE_DETAILED	RCP_PARAM_RECORD_HDR_MODE RCP_PARAM_PLAYBACK_STATE RCP_PARAM_PLAYBACK_NUM_HDR_TRACKS
RCP_PARAM_QUALITY	RCP_PARAM_REDCODE RCP_PARAM_RECORD_FILE_FORMAT RCP_PARAM_RECORD_VIDEO_CODEC
RCP_PARAM_ND_DISPLAY_VAL	RCP_PARAM_ND_VAL RCP_PARAM_MM_MODE RCP_PARAM_MM_ND_MODE
RCP_PARAM_EXPOSURE_DISPLAY	RCP_PARAM_SHUTTER_DISPLAY_MODE, RCP_PARAM_EXPOSURE_INTEGRATION_TIME, RCP_PARAM_EXPOSURE_ANGLE
RCP_PARAM_PLAYBACK_CLIP_DATE_TIME	RCP_PARAM_PLAYBACK_CLIP_DATE, RCP_PARAM_PLAYBACK_CLIP_TIME
RCP_PARAM_RECORD_STATE	RCP_PARAM_RECORD_STATE_BASE, RCP_PARAM_RECORD_MODE, RCP_PARAM_TETHERED_SERVER_STATE
RCP_PARAM_PRORES_DIMENSION	RCP_PARAM_PRORES_HEIGHT, RCP_PARAM_PRORES_WIDTH
RCP_PARAM_OPEN_GATE_STATE	RCP_PARAM_OPEN_GATE_MODE_ALLLOWED, RCP_PARAM_OPEN_GATE_MODE

PERIODICALLY UPDATED PARAMETERS

The camera broadcasts various periodic data about its state at varying rates. This data starts coming as soon as the physical connection to the camera is established. Even before calling `rcp_create_camera_connection()`. Even though the connection has not been created, the application must call `rcp_process_data()` for all received messages. The API needs incoming data to complete the connection process. The API just does not make any callbacks until the connection is created.

The type of information that comes periodically includes timecode, histogram, gyro data, temperatures, fan speeds, VU meter data, and battery level. Details of the update rates are found in the REDLINK Command Protocol: Reference Guide document.

If it is desired to ignore the histogram or VU meter and not have to process that data, set the respective callback pointer to NULL. The histogram data still comes and the `rcp_process_data()` call still needs to be made, but no callback happens.

PARAMETER STATUS

The API maintains the notion of whether a given parameter is available at any given time. For example changing camera state such as record or playback can make some parameters unavailable for modification, or changing fan mode to adaptive makes the manual record and preview speeds unusable. The function `rcp_get_status()` can be used to query the status of any parameter. The application must provide a call back, `rcp_cur_status_cb()`, for handling API generated change in status states. The intent is that the application should change the presentation of affected parameter controls, by hiding them, or perhaps greying them out and disable their selection.

NOTIFICATIONS

Events generated by the camera that require user attention are handled by notifications. In the camera, these are pop-up dialogs that can either timeout on their own or require the user to make a selection or explicitly dismiss them. When a notification occurs the camera will generate an RCP CURRENT message for the parameter NOTIFY. The API supports notifications with three supplied functions: `rcp_notification_get()`, `rcp_notification_timeout()`, and `rcp_notification_response()`; and one required application supplied callback: `notification_cb()`. This feature of the API is set up to aid handling and management of notifications in a manner consistent with the camera's operation. The application must provide presentation of the notification, update the presentation and manage timeouts based on data provided by the API in the callback function.

When a NOTIFY message is received from the camera, the API will call the application supplied callback with a structure of data filled in containing an action to perform and details about the notification. The action can be one of open, update or close defined in `rcp_notification_action_t`. The details will contain a unique id used to relate response and timeout calls, a title, a message to be displayed, the type of progress bar to be shown if any, progress percent if appropriate, a response list to be provided to user, and a timeout in seconds or 0 if no timeout. Consult the auto-generated documentation for details.

It is possible for multiple notifications to be open at the same time. The camera assigns a UUID to each as they are created and the API provides this in the callback data. This id becomes a parameter to the `rcp_notification_timeout()` and `rcp_notification_response()`. The API will handle the management of stacked up notifications, sending callbacks and handling responses until closed, one notification at a time in order of arrival. The application only needs to support a single instance.

NOTIFICATION FUNCTION USAGE

NOTIFICATION FUNCTIONS

FUNCTION	USE WHEN
notification_cb()	Called by the API when a notification is initiated by the camera, when an update is needed (progress bar update), or when the notification is to be closed.
rcp_notification_response()	This function should be called when a user responds to a notification by selecting one of the response options.

NOTIFICATION FUNCTIONS

FUNCTION	USE WHEN
rcp_notification_timeout	This function should be called to notify the API that the timeout associated with the current notification has expired and that the notification should be closed. Note: this can also be called if the user dismisses a timeout based notification by tapping on it (or other application specific appropriate action) Note: the client code should wait until the API issues a CLOSE action on the current notification before actually closing it
rcp_get_notification	This function will cause the notification callback to be re-called with the current notification to be displayed (if there is one). Note: this function doesn't usually need to be called. Only call it, if for some reason, your application ignores notification callbacks until some specific time during execution. Once your application is ready to handle the notification callbacks, call this function one time.

MENUS

Starting with RCP parameter set version 6.0, the camera can supply all the information needed to navigate and display the majority of the menu system. This is done using the `rcp_menu_get_children()` function. The menu system is represented as a tree where every node is either a branch or a leaf. Branches will have child nodes, which can be branches or leafs. The information for a leaf tells what kind of data element the menu item represents and what is needed for display. How the application handles those tasks is left to the developer, but node types are defined that represent the recommended methods. The tree root ID is a special enum value of `RCP_MENU_NODE_ID_ROOT`, which equates to Menu button of the camera UI. From there, the tree can be iteratively traversed by using the child or ancestor node IDs to descend or ascend the menu system.

Unsolicited callbacks should be ignored, because they are generated by another client navigating the menus. When switching between playback and record the menu should be closed and reopened starting from the root since the tree changes as a function of camera state.

The example iPhone application has the full source code of the best practices method of using the menu feature. It is highly recommend to use, or at least mimic that code.

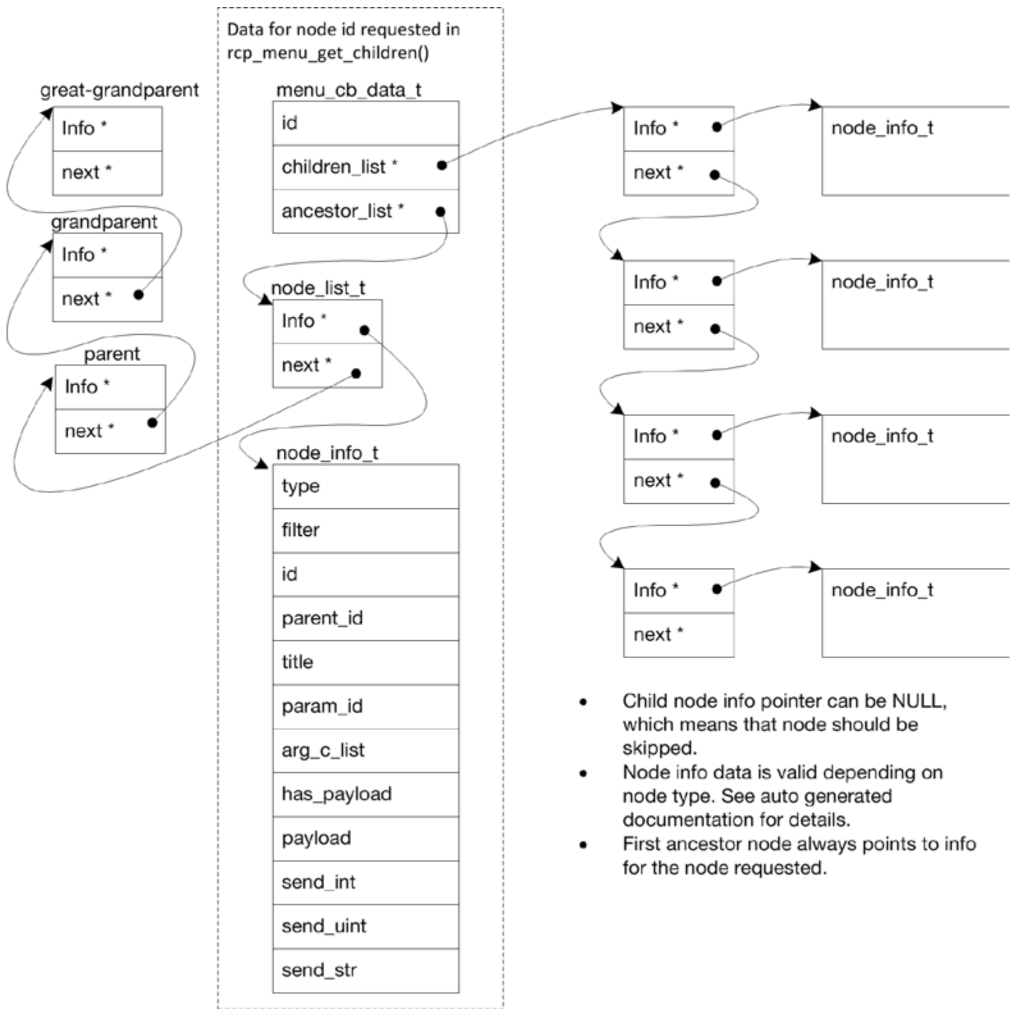

Data for node id requested in rcp_menu_get_children()
great-grandparent
grandparent
parent
Info *
next *
menu_cb_data_t
id
children_list *
ancestor_list *
node_list_t
node_info_t
type
filter
id
parent_id
title
param_id
arg_c_list
has_payload
payload
send_int
send_uint
send_str
Child node info pointer can be NULL, which means that node should be skipped.
Node info data is valid depending on node type. See auto generated documentation for details.
First ancestor node always points to info for the node requested.

Figure 8. Menu Tree Data Structures

FILE TRANSFER

Starting with camera firmware version 6.2.x, the REDLINK protocol supports transfer of files to and from the camera via RCP. This is known as ‘rftp’ for REDLINK file transfer protocol. Storage is presented as a virtual file system implemented partially in camera memory, and partially in SSD media. The ability to read files, write files, delete files and get directory listings is provided.

FILE SYSTEM

The virtual file system comprises a group of folders and one file at the root folder ‘/’. Only certain file types are permitted in each folder and the camera enforces this based on filename extension with one exception. The /luts folder allows any extension name. Extension names are case sensitive.

VIRTUAL FILE SYSTEM

FOLDER	FILES	ACCESS TYPE	DESCRIPTION
/	log	Read only	The only file available is 'log'. A camera log file for sending to customer support.
/force_preset	*.preset	Read and write	Place a preset file in this folder and the re-boot the camera to have preset automatically applied during boot.
/force_upgrade	*.bin	Read and write	Place an upgrade package file in this folder and then re-boot the camera to perform a firmware upgrade
/looks	*.RMD	Read and write	Look settings files
/luts	Any extension	Read and write	Currently supports .cube and .cns LUT files.
/media	SSD contents	Read only	This maps to the mounted SSD. Allows transfer of recorded media or other files out of the camera. Writing onto the SSD is not permitted.
/overlays	*.overlay	Read and write	Overlay settings files
/presets	*.preset	Read and write	Preset files
/thumbnails	*.rtn	Read only	This folder contains thumbnails of the clips on the mounted media. It is loaded when the SSD is mounted if there is already content, and added to at the end of each subsequent clip recording.

DIRECTORY LISTING FORMAT

The directory listing data is a list of file properties and filenames, in the format of a type 0 cList. The properties are a bit mapping defined as below.

- 1: the file is a directory.
- 2: the file has read permissions.
- 4: the file has write permissions.

A read only directory, such as /media, will have a property of 3. A file with both read and write permissions, such as a .RMD, will have a property of 6.

The file names are all relative to the directory for the listing was requested.

THUMBNAIL FORMAT

Thumbnail images are created for each recording and are available in the /thumbnail virtual folder. They are optimized for rendering on a camera monitor and therefore do not use the full 8 bit range for color. Each color (red, green, blue) uses only the most significant 4 bits. The least significant 4 bits are 0s.

THUMBNAIL DATA FORMAT

ITEM	SIZE (BYTES)	DESCRIPTION
Signature	12	Identifies the file as a RED thumbnail = "REDTHUMBNAIL"
Version	1	File format version, currently 3
Reserved 1 size	4	Size of the following reserved block
Reserved 1 data	Reserved 1 size	N/A
Reserved 2 size	4	Size of the following reserved block

THUMBNAIL DATA FORMAT

ITEM	SIZE (BYTES)	DESCRIPTION
Reserved 2 data	Reserved 2 size	N/A
Reserved 3 size	4	Size of the following reserved block
Reserved 3 data	Reserved 3 size	N/A
Image width	4	Width of image data in pixels
Image height	4	Height of image data in lines
Image data	Image width * 4 * Image height	Image data in row major order. 4 bytes per pixel, 8 bits per color, as ARGB (alpha, red, green, blue). Only the upper 4 bits of the red, green, and blue are used.

FILE TRANSFER FUNCTIONS

FILE TRANSFER FUNCTIONS

FUNCTION	USE WHEN
rcp_rftp_is_supported()	Check if rftp operations are supported by the connected camera.
rcp_rftp_directory_lilsting()	Get a directory listing for the specified virtual path. The listing is returned as a raw string by the status callback. Call `c_list_import_from_string()` to convert to a cList.
rcp_rftp_send_file()	Send a file to the specified virtual path (including filename). The file may be compressed or uncompressed. It is the application's job to compress if desired.
rcp_rftp_retrieve_file()	Retrieve the specified filename path from the camera virtual file system. The file data is returned by the status call back.
rcp_rftp_delete_file()	Delete the specified filename path from the camera virtual file system.
rcp_rftp_abort_transfer()	Halt the current file transfer (send or retrieve).

STATUS CALLBACK

Return data and status for rftp functions are all communicated via the callback function provided to the connection creation call. File contents for a read or directory listing data are provided as data to the callback. The callback is also called periodically to provide completion status of a transfer.

FILE COMPRESSION

Files being transferred to or from the camera may be compressed. When sending, the application makes this decision and must include the compressed and uncompressed data size in the function call and indicate whether it is compressed. When retrieving a file, the application can indicate whether the file may be compressed and the status callback indicates if the camera did compress. Compression is not always performed and the camera makes this choice. If the application indicates to not compress, the camera will not.

The camera uses the Zlib library to compress and uncompress files. This library is not included in the API. The application developer will need to get version 1.2.8 or later from www.zlib.net. However, since the application can disallow compression, the inclusion of Zlib can be avoided.

APPLICATION PROVIDED FUNCTIONS

To help keep the API as platform independent as possible, these helper functions must be implemented by the application code. These are declared in the file rcp_api.h. The user has the option to place the implementation c files anywhere they choose as long the linker can find them.

- **rcp_malloc**: Wrapper for memory allocations required by API.
- **rcp_free**: Wrapper for memory allocations required by API.
- **rcp_mutex_lock**: Wrapper for mutex lock required by the API. Note that the mutexes provided must be recursive mutexes. That is, the same thread must be able to lock the same mutex multiple times before unlocking it.
- **rcp_mutex_unlock**: Wrapper for mutex unlock required by the API.
- **rcp_log**: Wrapper for logging messages from the API. The application may choose to do nothing with them, but the call must be provided.
- **rcp_rand**: Wrapper for getting pseudo-random integer required by the API.
- **rcp_timestamp**: Wrapper for getting system time in milliseconds.

WRAPPER FOR JAVA

The REDLINK Java wrapper is provided to allow use of the REDLINK API by Java desktop (Windows and Linux) and Android applications. It wraps the REDLINK API with Java code utilizing the Java Native Interface (JNI).

The Java wrapper is supplied in several folders under /rcp_api/java in the REDLINK SDK sources. It has its own README.txt and auto generated html documentation. A small test function is also provided.

There is not a one to one correspondence of Java methods to API C functions and the method names are different. Be sure to review the Java wrapper documentation.

SDK FOLDER STRUCTURE

The REDLINK SDK Source and Reference Applications zip archive contains the SDK sources and several example applications. The top-level folder /redlink_sdk, contains the API and supporting function source files. The /redlink_sdk folder can be copied elsewhere for use in your own application development. The applications under the top level /examples folder reference the /redlink_sdk folder, so the archive structure should be maintained as is when uncompressing to your machine.

SDK CODE FOLDER CONTENTS

SDK FOLDER CONTENTS

FOLDER	DESCRIPTION
/redlink_sdk/types	Contains rcp_types_public.h which exposes the various enumerated values.
/redlink_sdk /rcp_api	.c and .h files for the top level API. The application will need to include rcp_api.h. This folder also contains the /doc folder with the auto generated documentation and the /java folder for the Java interface wrapper.
/redlink_sdk /rcp_api/doc	Doxygen auto generated html files for the API. Start with a browser pointing to index.html in this folder.
/redlink_sdk /rcp_parser	.c and .h files for the core parser module. Not used directly by the application.
/redlink_sdk /base64	.c and .h files for the base64 encoder and decoder module. Not used directly by the application.
/redlink_sdk /clist	.cpp and .h files for the cList. The application will need to include clist.h
/redlink_sdk /decorated_string	.c and .h files for the decorated_string module. Not used directly by the application.
/redlink_sdk /keys	.h file for #defines used to build key codes. Not required to be used by the application.
/redlink_sdk /stringl	.c and .h files for the stringl module. Not used directly by the application.
/redlink_sdk /rcp_api/java	The Java wrapper, arranged in several folders.
/redlink_sdk /rcp_api/java/doc	Javadoc auto generated html files for the Java wrapper. Start with a browser pointing to index.html in this folder.
/redlink_sdk /rcp_api/java/jni	.cpp and .h files providing a wrapper around the API that is callable from Java classes in ./rcp_api/java/src/com/red/rcp
/redlink_sdk /rcp_api/java/test	Java test application
/redlink_sdk /rcp_api/java/src/com/red/rcp	Java interfaces and classes for access from Android or Java desktop
/examples/android	Android Jelly Bean 4.2 (API level 17) app sources
/examples/ios	iOS 7 iPhone app sources
/examples/qt/api_example	Source and executable (OS X and Windows) of PC based app using API
/examples/qt/core_example	Source and executable (OS X and Windows) of PC based app using only low level core functions
/examples/qt/common	Files used by both Qt applications

API BUILD CONFIGURATION

The RCP API can be customized at build time with the macro definition options below to reduce the overall footprint. That is, both code size and memory usage can be reduced by disabling certain portions of the API or changing the size of internally used buffers. Modifications to the options are made by editing the file rcp_api_config.h

Note: The default settings enable everything and have properly sized buffers. Furthermore, the API has primarily been tested only with the default settings. Take care when changing any of the settings below.

API BUILD OPTIONS

OPTION	DEFAULT	DESCRIPTION
RCP_API_ENABLE_LABELS		If disabled, there will be no support for parameter labels. `rcp_get_label()` will always return NULL.
RCP_API_ENABLE_STR_TO_ENUM		If disabled, there will be no support for converting parameter ids (`rcp_param_t`) to strings, or vice versa. `rcp_get_name()` will always return NULL and `rcp_get_id()` will always return RCP_PARAM_COUNT
RCP_API_ENABLE_CACHEING		If disabled, memory usage will be reduced by not caching the value of every parameter in the API itself. Note: this will increase the amount of traffic between the application and camera.
RCP_API_ENABLE_NOTIFICATIONS		If disabled, notifications from the camera will not be supported. `rcp_notification_get()`, `rcp_notification_timeout()`, and `rcp_notification_response()` will all do nothing and return RCP_SUCCESS.
RCP_API_ENABLE_CLIP_LIST		If disabled, the clip list from the camera will not be accessible through the API.
RCP_API_ENABLE_LOGGING		If disabled, no log messages from the API will be generated.
RCP_API_ENABLE_DISCOVERY		If disabled, camera discovery will be disabled. `rcp_discovery_get_list()` will always return NULL and all other `rcp_discovery_*` calls will do nothing.
RCP_API_ENABLE_MENU		If disabled, menu navigation will be disabled.
RCP_API_DISPLAY_STR_SIZE	100	Maximum size of any display string generated by the API
RCP_API_LOG_LINE_SIZE	1024	Maximum size of any line sent to a logger function
RCP_API_PARSER_BUFFER_SIZE	RCP2_MAX_PACKET_LENGTH	Size of internal buffer used to parse incoming RCP packets. Note: if this is reduced in size, not all packets are guaranteed to be handled correctly.
RCP_API_OUTGOING_PACKET_BUFFER_SIZE	RCP2_MAX_PACKET_LENGTH	Maximum size of any outgoing RCP packet.

API BUILD OPTIONS

OPTION	DEFAULT	DESCRIPTION
RCP_API_SOURCE_NAME	"API"	Up to 8 characters, RCP message source name inserted by API in outgoing messages to a camera.
RCP_PARAMETER_SET_MIN_VERSION_MAJOR	5	Minimum RCP parameter set version supported
RCP_PARAMETER_SET_MIN_VERSION_MINOR	0	
RCP_PARAMETER_SET_VERSION_MAJOR	6	Maximum RCP parameter set version supported
RCP_PARAMETER_SET_VERSION_MINOR	0	

BYPASSING THE API

Since the application owns the camera connection, the application can also be written to simply not use the API. This might be useful for applications with limited memory or processing capacity, where the provided API build options are insufficient to minimize the footprint. Only the RCP Core sources would be needed in that case.

ACCESSING RCP IN EPIC/SCARLET

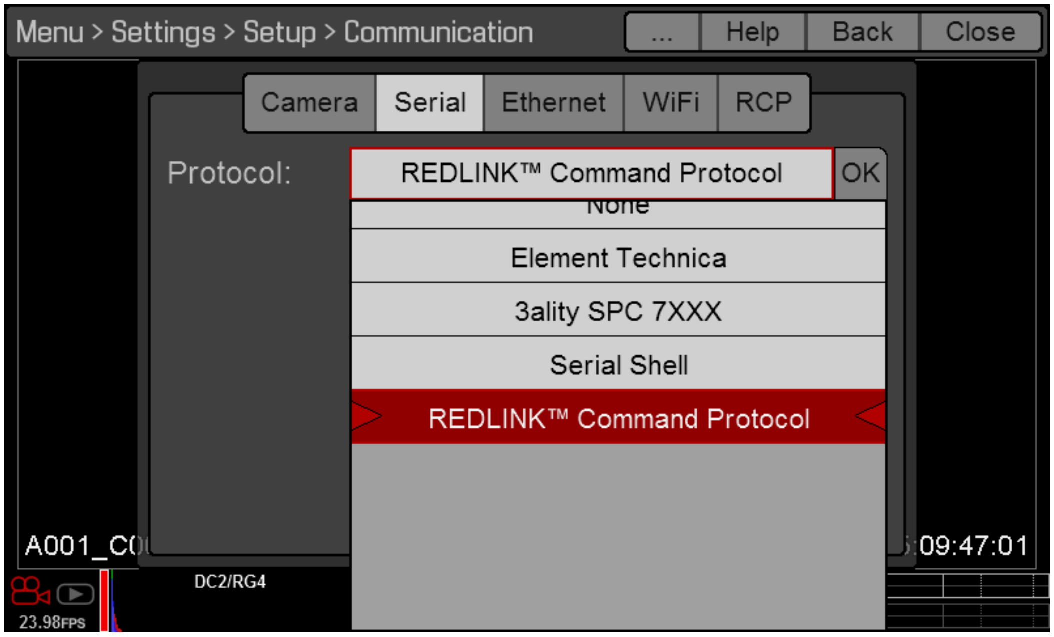
VIA SERIAL PORT

Starting with camera firmware 3.3.x, the RCP is available through the CTRL connector (RS232 port) on the rear of the camera body. The RCP is disabled by default, and must be enabled in the DSMC menu.

WEAPON and RAVEN cameras will require a module that supports the user serial port. Currently this includes the DSMC$^{2™}$ BASE EXPANDER, DSMC^2 JETPACK EXPANDER, DSMC^2 BASE I/O V-LOCK EXPANDER and the DSMC^2 REDVOLT® EXPANDER.

1. Go to **Menu** > **Settings** > **Setup** > **Communication**.
2. Select the **Serial** tab.
3. Select **REDLINK Command Protocol** in the **Serial Protocol** drop-down menu.

   Once selected, REDLINK Command Protocol is persistent across boots.



Protocol: REDLINK™ Command Protocol

Figure 9. Serial Port Configuration Menu.

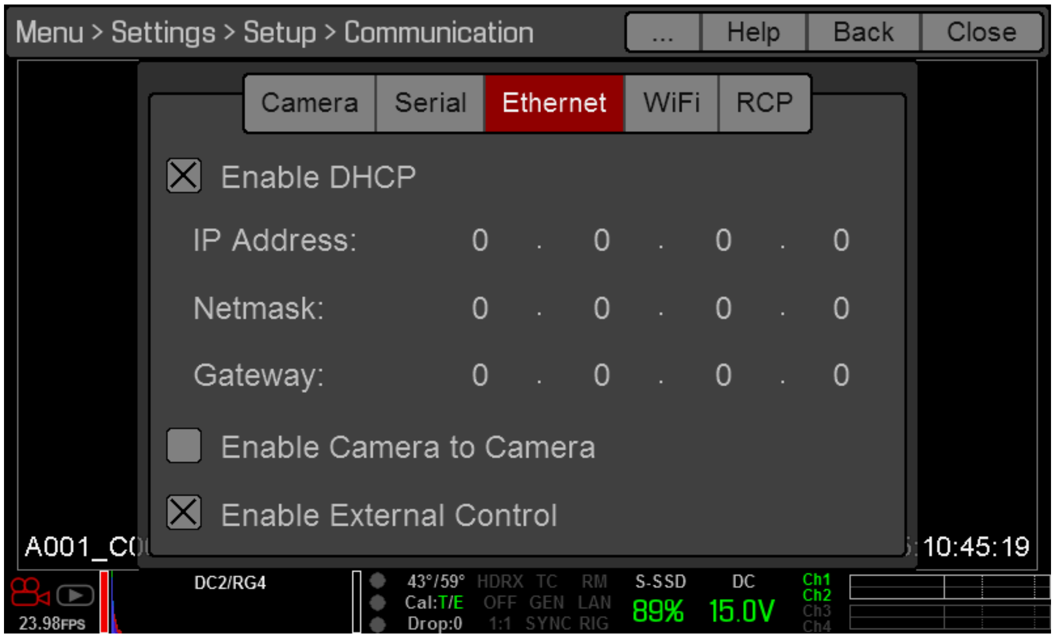
The port settings are 115200 baud, 8N1 (8 bits, no parity, 1 start bit, no flow control).

The 4-PIN 00 LEMO®-TO-FLYING LEAD (P/N 790-0187) may be used for custom connections. See the RED DSMC Operation Guide, section “CTRL (RS232 CONROL)” for details.

VIA GIGE

RCP is available through the GigE connector on the rear of camera, and must be enabled in the camera menu. For WEAPON and RAVEN, the DSMC^2 REDVOLT Expander module is required.

1. Go to **Menu** > **Settings** > **Setup** > **Communication** menu.
2. Select the **Ethernet** tab.
3. Select **Enable External Control check box**.



☒ Enable DHCP

IP Address: 0 . 0 . 0 . 0

Netmask: 0 . 0 . 0 . 0

Gateway: 0 . 0 . 0 . 0

☐ Enable Camera to Camera

☒ Enable External Control

Figure 10. Ethernet Configuration Menu.

Communication is allowed over TCP (port 1111). It is your responsibility to configure the camera, network, and application properly. The camera can be configured for a fixed or dynamic IP address. A dynamic discovery process is also supported.

The LEMO-to-CAT5E Ethernet Cable (9') (P/N 790-0159) is required. See the RED DSMC Operation Guide, section "GIG-E (Ethernet)" for details.

Check that the LAN indicator turns green when External Control is enabled and the camera is connected to a network. If the LAN indicator turns yellow or red, contact a Bomb Squad representative to determine if the camera needs updating.

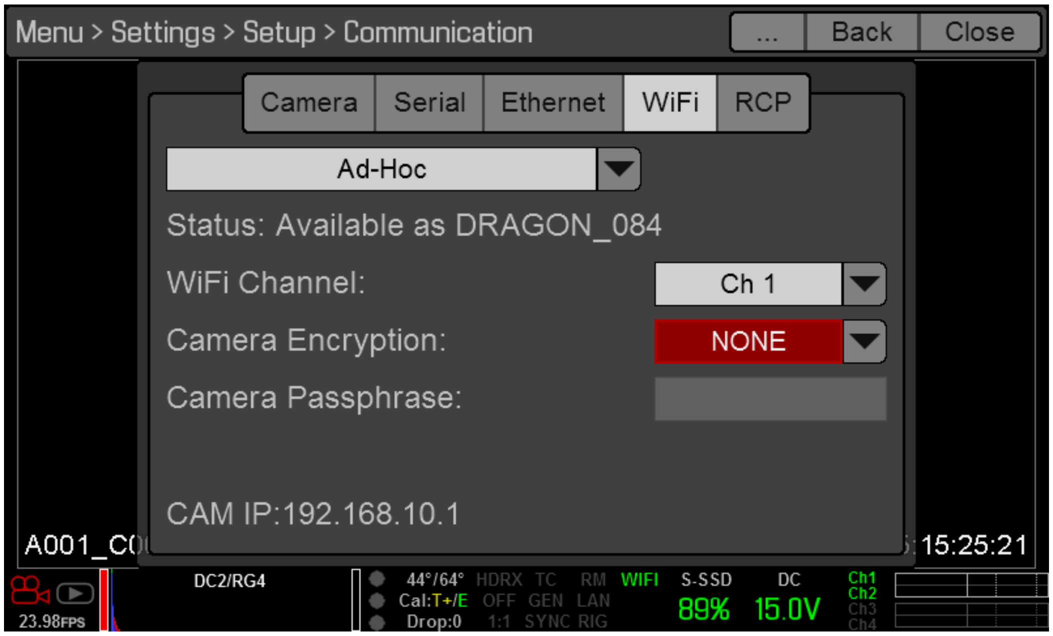
As of firmware version 5.1.33, the camera does not support more than eight (8) simultaneous connections. Your application should maintain just one. If the connection is lost, close down the connection before opening a new one. Once the maximum number of connections is reached, the camera ignores additional requests. Other than refusing the connection, the camera does not indicate why the connection is unsuccessful.

VIA WIFI

Starting with camera firmware 5.2.x, the RCP is available through the REDLINK Bridge module for EPIC and SCARLET cameras. The REDLINK Bridge is automatically enabled if connected, but you must enable WiFi and the connection method in the camera menu. The REDLINK Bridge can support one connection.

The WEAPON and RAVEN cameras have WiFi built in and also require enabling and specifying the connection method.

1. Go to **Menu** > **Settings** > **Setup** > **Communication**.
2. Select the **WiFi** tab.
3. Select the **Enable WiFi** check box**.**
4. Select **Ad-Hoc** or **Infrastructure** mode and set up the connection using the instructions from the REDLINK BRIDGE Operation Guide, available at www.red.com/downloads.



Ad-Hoc

Status: Available as DRAGON_084

WiFi Channel: Ch 1

Camera Encryption: NONE

Camera Passphrase:

CAM IP:192.168.10.1

Figure 11. WiFi Configuration Menu.