Remote Access Protocol

Introduction

The Remote Access Protocol, or RAP, allows for remote control and configuration of Ample Power products via either a computer or another Ample Power product.

Throughout this manual, the system generating a request is referred to as the *host* and the system responding to the request is referred to as the *target*. Any system in the network can act as *host*, *target* or both *host* and *target*. (Often referred to as *peer to peer*.

The term *dictionary* refers to the Object Dictionary described in more detail later. Reference is also made to UI, for *user interface*. The Dictionary contains user interface elements as well as variables and functions.

Information is communicated between the systems in packets of ascii text. All packets are terminated by a newline character (ascii value `0x0A`, commonly represented as the `'\n'` character).

If a hash symbol, `'#'`, is seen before the start of a packet, the rest of the input up to a `'\n'` is ignored. This allows one to write RAP scripts with comments that do not need to be removed before sending the script to a *target*.

Note that carriage returns are simply ignored if a terminal emulator running on the *host* sends a carriage return (`'\r'`) followed by a newline (`'\n'`) to terminate a line. The *target* will always terminate packets with a newline only.

In the packet descriptions, a sequence of characters in angle brackets is only a symbolic representation of that portion of the packet. The angle brackets are not part of the actual packet. For example, to query the value of the `'b1v'` variable, `'<name>'` is simply replaced with `'b1v'`.

All communication is initiated by a *host* system sending out a request packet. The *host* sends a request packet and waits for the *target* to respond with a response packet. If the *target* is unable to fulfill the request, an error packet is returned. If the *target* is unable to parse the request packet, due to line noise or dropped data, no response will be sent since there is no way to route back to the *host* without successfully parsing the request.

Source code for an example packet parser can be obtained from Ample Power upon request.

The serial port configuration parameters are listed in Table **??**.

Packet Handling

A RAP port can both send and receive request packets. This allows multiple RAP enabled devices to communicate with each other. What this also means is that after sending out a request packet, the next packet received may not be the response to the request. It could be a request from another device or a delayed response from a previous request.

After a packet has been sent to the *target*, the *host* should wait for replies from the *target* systems before sending the next packet. The host should expect any number of replies. If there are N *targets* connected either directly or indirectly to the *host*, the *host* may get 0 to N replies to any given request. The time for all responses to get back to the *host* is a function of the number of *targets* and the number of network hops between the *host* and each *target*. Typically, the *host* will need to timeout the request after some nominal period.

Data Packet Format

Data packets can be either requests or responses. The general format of a rap packet is:

    $<direction><data>#<crc>\n

where `$` denotes the start of the packet, `<direction>` is either `'+'` or `'-'`, `<data>` is the packet's data, # marks the end of the data and `<crc>` is an optional 16 bit Cyclic Redundancy Checksum (CRC). The newline character terminates the transmission. The `<data>` part of the packet should not contain the `$`, # or newline characters.

If the `<crc>` is present, it must be exactly 4 hexadecimal digits (case is ignored). The *target* will generate an error if the CRC is invalid. The CRC calculation will include the leading `$` and trailing # characters in addition to the `<data>` portion of the packet.

If the *host* wishes to use CRC's for packet transmission, the C function shown in the *Example CRC16 Function* section can be used to calculate the CRC value. An example of using that function is shown in the *Example CRC16 Usage* section.

The *target* will always generate the CRC when sending data packets.

The `<data>` part of a Request RAP Packet has this form:

    <cmd>:<idx>:<name>:<seq>:<args>:<stid>

while the `<data>` part of a Response RAP Packet is just an extension of the request packet (the `<request_data_fields>` part) and has this form:

    <request_data_fields>:<err>:<resp>

Where all ':' characters are required, but some fields may be empty if not used. The fields of the packets of are defined as follows:

\<cmd\> The RAP command.

> Can be multiple characters. If prefixed with a '+', the packet is a request. If prefixed with a '-', the packet is a response.

> The various generic command types are summarized in Table **??** and specialized UI element query command types are summarized in Table **??**. A detailed description of request packet types and the expected responses is given in the following sections of this manual.

\<idx\> The Dictionary entry Index. Hexadecimal format.

> May be left blank if not needed by command. May be left blank if *host* used `<name>`. If *host* used `<name>`, `<idx>` will be filled in by *target*.

\<name\> The Dictionary entry Name.

> May be left blank if not needed by command. May be left blank if *host* used `<idx>`. If *host* used `<idx>`, `<name>` will be filled in by *target*.

`<seq>` A sequence number. Hexadecimal format.

> If the request sequence number is non-zero, it tells the *target* that the request is a periodic request and the value is the period of the responses. The *host* can tell the *target* to stop the periodic responses by resending the request with a 0 sequence number.

> If the response to a request requires multiple responses to be fulfilled, the *target* will set the first response packet with a seq number of 1, while incrementing the seq number for each subsequent response packet. A termination packet with a seq number of zero will be sent after the final data packet.

`<args>` Arguments for the command.

> Leave blank if command takes no arguments. A string of comma separated arguments.

`<stid>` Source Task ID. Hexadecimal format.

> The task id of the task that generated the request. This field can be left blank by the *host* and will be filled in by the system when needed.

> The *target* to a request must copy the `<stid>` from the request into the response packet.

`<err>` Response RAP error code. Hexadecimal format.

> If code is non-zero, `<resp>` will contain a string message describing the error (NOTE: systems with limit code space, may not be able to supply the error string and will simply leave `<resp>` blank).

`<resp>` The response to the request.

> May be left blank is no response string is needed.

`<crc>` The 4 hex digit CRC.

> Calculated the same as in RAP version 1. Optional in requests, but will always be sent with responses.

RAP Routing

RAP Packets can have an optional RAP routing header that precedes the $ sign. The routing header contains *source* and *destination* information. The *host* can be connected to many *targets* directly (via multiple RAP ports), or indirectly (via a network). Without a RAP routing header, the *host* can only get a response from a single, directly connected *target*. Adding a routing header allows the *host* to get responses from multiple *targets*. The routing header also allows the *host* to send a request to a specfic *target*.

A RAP packet with the routing header looks like this:

```
&<snid>:<dnid>$+<data>#<crc>\n
```

Where the routing header fields are defined as follows:

`<snid>` The Source Node ID.

> The node id of the system transmitting the packet.

`<dnid>` The Destination Node ID.

> The node id of the system receiving the packet.

For a request packet, SNID refers to the node id of the *host* system and the DNID refers to the node id of the *target*. The DNID can be the broadcast id.

For a response packet, SNID refers to the node id of the responding *target* system. The DNID is the node id of the *host* system which originally generated the request. Both fields must be filled in by the *target*.

A node id is a 64 bit number uniquely identifying a system. If either node id is left blank by the *host* in a request packet, it refers to the directly connected system and will be filled in correctly by the *target* when responding. The *host* can also set DNID to `'*'` to send a request to all *targets* (i.e. broadcast the packet). A node id also has some internal structure which is detailed in Table **??**.

A special case involves node ids that represent phone numbers for sms text messages. The value displayed in the rap header is the negated phone number in hex format. For example:

```
printf ("%llx", -phone_num);
```

Unfortunately, this makes typing the phone number by hand at a terminal difficult if not impossible. To make this easier, an alternate format is available. Consider the following example RAP packet:

```
&:+18885551212$+<rap_data>#
```

In this case, the phone number is prefixed with a `'+'` character and interpreted as a decimal number. The response back from this RAP request would see the phone number negated in the snid field displayed in hex format as such:

```
&ffffffffb9a555b94:<localhost_id>$-<data>#
```

The Object Dictionary

The *target* system contains a database of objects which is referred to as the *dictionary*. The objects within the *dictionary* are variables, executable functions or user interface elements. All accessible variable objects are readable, while a subset of those are writable.

Changing the value of a variable or calling a function may require the *host* to login to the system at a certain privilege level. The privilege levels are:

root	Reserved for Ample Power. Can do anything.
dist	Distributor.
dlr	Dealer.
syscfg	System Configurator.
tech	Technician.
oper	Operator.
user	Default User. Minimal privileges.

Each privilege level can do anything that the level below it can do. To login at a specific privilege, use the `f_login_<priv>` function, where `<priv>` is one of the above privilege levels, and the password is given in the `<arg>` field of the request packet.

All objects are referenced by either name (`<name>`) or index (`<idx>`). Accessing an object by index is more efficient for the *target* system, but the indices are not guaranteed to be invariant from one firmware version to the next. Object names will be consistent across firmware versions.

Every reply from the *target* which accesses a *dictionary* object will include both the index and the name of the object. This is done to allow the *host* to map `idx` to `name` pairings if it needs to track that information.

The index (`<idx>`) part of a *target* response packet is a hexadecimal digits in big-endian byte order.

For brevity, only *dictionary* access requests made by name are shown in the discussions which follow. In all cases, the same access request can be made by using the `<idx>` field instead of the `<name>`.

Querying the size of the Dictionary

For some host systems, knowning the number of entries in the *dictionary* can be useful. The number of entries is obtained using the following format:

    $+?N:::::#

The *target* replies with the `<resp>` field set to:

    <entries>

where `<entries>` is a hexadecimal number representing the number of entries in the *dictionary*.

Querying an Object's Data Type

Why does the *host* need to know about data types? So it can program setpoints that match the rules for the conversion functions embedded with the variable in the dictionary. These rules are enforced to cut down on user input errors.

The type information also supplies meaning to some variables beyond their numerical value. For instance, an 8–bit variable may be a simple boolean that matches a yes/no answer to a configuration question. Here a yes is represented by a one value and a zero indicates no.

Other variables represent states within finite state machine logic, where their numerical value indicates the present state of the machine.

Finally, other variables are bit fields, or masks, where each bit in the mask indicates a particular condition.

To get data type information, the *host* sends:

    $+?t::<name>:::#

The *target* replies with the `<resp>` field set to:

    <t>,<b>,<p>,<d>

with this significance:

\<t\> The object type . . .
    r – object is a readable variable;
    w – object is a writable variable;
    f – object is an executable function;
    A – object is an anykey ui element;
    C – object is a confirm ui element;
    F – object is a form ui element;
    M – object is a menu ui element; or
    P – object is a panel ui element.

\<b\> The size in bits (decimal integer).

\<p\> A polyglot of information . . .
    s – variable is signed;
    u – variable is unsigned;
    b – variable is a boolean;
    v – variable is state vector.
    m – variable is bit field/mask; or
    t – variable is text string.

\<d\> The number of acceptable decimal points, typically 1 or 2.

Note: If an object is a function or a ui element, then the remainder of the type information is not meaningful and can be ignored.

Tip: The *host* system can build up it's own (name, index, type) table for all objects in the *dictionary* by querying object types in a loop starting with index zero and incrementing the index until the *target* returns an 'Invalid Index' error code. The *host* would need to rebuild it's table after a firmware update in the *target* to re-map names to indices.

Querying an Object's Description

Every object in the *target* dictionary has both a short and a long description string associated with it.

To request the short description string, the *host* sends:

    $+?d::<name>:::#

The *target* responds with the `<resp>` field set to:

    <len>,<str>

where `<len>` is a hexadecimal value in big-endian format which specifies the length (in bytes) of the `<str>` part of the packet.

Note that no terminating zero is sent with the string and `<len>` does not include one either. This holds for all strings transferred via the RAP protocol.

To request the long description string, the *host* sends:

    $+?D::<name>:::#

The *target* responds with the `<resp>` field set to:

    <len>,<str>

Long descriptions are intended as *help* for a given variable and are used for that purpose with the user interface via terminal emulation. In that user interface, short descriptions are used as a terse description for the variable and are usually concatenated with the value for that variable when displayed.

Querying a Variable's Value

The value of a variable is requested using the following format:

    $+?v::<name>:::#

The *target* responds with with the `<resp>` field set to:

    <val>

The format of `<val>` depends on the type of the object. For text string variables, `<val>` will be in the form `'<len>,<str>'`. For bit field or mask variables, `<val>` will be a hexadecimal value in big-endian byte order. For all other variable types, `<val>` will be a decimal number, possibly having a decimal point.

Querying a Mask Variable's Information

Bit fields or *masks* have a text description associated with each bit in the mask. The description for any bit can be obtained by sending a query with one bit set.

The *host* sends the string:

    $+?m::<name>::<mask>:#

where `<mask>` is a hexadecimal number.

The *target* responds with with the `<resp>` field set to:

    <len>,<str>

where `<len>` is a hexadecimal value in big-endian format which specifies the length (in bytes) of the `<str>` part of the packet. A request for a mask without a text description will return a zero length.

Querying a State Vectors's Information

State variable objects have a text description associated with numeric values of the variable. The description for any value can be obtained by sending a query with that value.

The *host* sends the string:

```
$+?S::<name>::<state>:#
```

where `<state>` is a decimal number.

The *target* responds with with the `<resp>` field set to:

```
<len>,<str>
```

where `<len>` is a hexadecimal value in big-endian format which specifies the length (in bytes) of the `<str>` part of the packet. A request for a state vector without a text description will return a zero length.

Setting a Variable's Value

To set a variable's value, *host* sends the string:

```
$+s::<name>::<val>:#
```

The *target* responds with the `<resp>` field empty.

The decimal point precision is enforced by the *target* according to the variable's data type specification. An error is returned if the precision does not match the specification.

Executing a Function Call

Some names belong to functions, i.e. 'f_eng_start'. The *host* can instruct the *target* system to execute a function by sending:

```
$+e::<name>:::#
```

The *target* responds with the `<resp>` being either empty or set to something appropriate for the function.

If arguments are required for the function they are passed as a comma separated list in the `<arg>` field of the request packet:

```
$+e::<name>::<arg1>,...,<argN>:#
```

Example CRC16 Function

```
uint16_t
update_crc16 (uint16_t crc, uint16_t data)
{
    int i;

    for (i = 8; i; i--) {
        if ((data ^ crc) & 0x0001)
            crc = (crc >> 1) ^ 0xA001;
        else
            crc >>= 1;
        data >>= 1;
    }
    return crc;
```

```
}
```

Example CRC16 Usage

```
#include <stdio.h>
#include <stdint.h>

struct tc_t {
    uint16_t exp; /* Expected CRC value. */
    uint8_t *buf; /* The test string. */
};

static struct tc_t tests[] = {
    {0x35c0, "M"},
    {0xff01, "T"},
    {0x23b6, "THE"},
    {0xb96e, "THE,QUICK,BROWN,FOX,"
            "0123456789"},
    {0, NULL}
};

int
main (int argc, char ** argv)
{
    uint16_t crc16;
    uint8_t *ptr;
    struct tc_t *tt = tests;

    while (tt->buf) {
        crc16 = 0;
        ptr = tt->buf;

        while (*ptr) {
            crc16 = update_crc16 (crc16,
                                  *ptr);
            ptr++;
        }
        printf ("crc of \"%s\"\n"
                "    crc16 = 0x%04X "
                ": expected = 0x%04X\n",
                tt->buf, crc16, tt->exp);

        tt++;
    }
    return 0;
}
```

CMD	Required Fields	Description	Response
`?v`	`<idx|name>`	Query object's value.	`<value>`
`?d`	`<idx|name>`	Query object's short description.	`<len>,<str>`
`?D`	`<idx|name>`	Query objects's long description.	`<len>,<str>`
`?t`	`<idx|name>`	Query object's data type.	`<t>,<b>,<p>,<d>`
`?m`	`<idx|name>,<arg>`	Query mask variable's information. Arg is hex mask for bit to query.	`<len>,<str>`
`?S`	`<idx|name>,<arg>`	Query state variable's information. Arg is decimal number of state to query.	`<len>,<str>`
`s`	`<idx|name>,<arg>`	Set variable to value. Arg is the value to use to set the variable.	
`e`	`<idx|name>`	Execute a function on the *target* system. Arg is an optional, comma seperated list of function arguments.	`<str>`
`x`		Terminate RAP terminal instance if possible.	
`h`		Display help for commands. Arg is the command to get help for. If no arg given, gives a list of all commands without descriptions.	`<str>`
`E`		Display error code descriptions. Arg is error code to lookup. If no arg is given, dump all error codes and descriptions.	`<err_code>,<err_msg>`
`?N`		Query number of dictionay entries.	`<value>`
`Rd`		Display routing table entries. Optional arg is a node id to display. If no arg is given displays all entries.	`<if_mask>,<if_name>,<node_id>`

Table 1: Summary of generic packet types

Error Code	Description
E00	Packet OK.
E01	Invalid CRC.
E02	Name is not in the dictionary.
E03	Object is not writable.
E04	Object is not a function.
E05	Programmed value is out of range.
E06	Decimal point error.
E07	Only decimal digits accepted.
E08	Malformed Packet.
E09	Invalid Query.
E0A	Invalid Operation.
E0B	Variable is not a mask type.
E0C	Variable is not a state vector.
E0D	Invalid Index.
E0E	Invalid State.
E0F	Invalid Mask.
E10	Input too long.
E11	Programmer Bug.
E12	Not Implemented.
E13	Type Mismatch.
E14	Permission denied.
E15	Invalid Offset.
E16	Bad Routing.

Table 2: Error Codes

Parameter	Value
Speed	38400
Parity	None
Data Bits	8
Stop Bits	1
Hardware Flow Control	Off
Software Flow Control	Off

Table 3: Serial Port Configuration

Field	Bits	Width
Network InterFace ID	0-3	4
Serial Number	4-23	20
Product ID	24-31	8
Manufacturer ID	32-40	8
Unused	41-62	8
Phone Number Flag	63	1

Table 4: Node ID Subfields

CMD	Required Fields	Description	Response
?UAP	`<idx|name>`	Anykey UI Element. Get prompt string.	`<str>`
?UAn	`<idx|name>`	Anykey UI Element. Get index of next ui element.	`<idx>`
?UCP	`<idx|name>`	Confirm UI Element. Get prompt string.	`<str>`
?UCa	`<idx|name>`	Confirm UI Element. Get the index of confirmation action function.	`<idx>`
?UCD	`<idx|name>`	Confirm UI Element. Get the index of what to display if user denies.	`<idx>`
?UCA	`<idx|name>`	Confirm UI Element. Get the index of what to display if user accepts.	`<idx>`
?UFt	`<idx|name>`	Form UI Element. Get title string.	`<str>`
?UFp	`<idx|name>`	Form UI Element. Get index of parent ui element.	`<idx>`
?UFi	`<idx|name>,<arg>`	Form UI Element. Get index of form item. Arg is item number (e.g. 0 - (N-1)).	`<idx>`
?UMt	`<idx|name>`	Menu UI Element. Get title string.	`<str>`
?UMp	`<idx|name>`	Menu UI Element. Get index of parent ui element.	`<idx>`
?UMP	`<idx|name>,<arg>`	Menu UI Element. Get menu item prompt string. Arg is item number (e.g. 0 - (N-1)).	`<str>`
?UMi	`<idx|name>,<arg>`	Menu UI Element. Get index of menu item. Arg is item number (e.g. 0 - (N-1)).	`<idx>`
?UPt	`<idx|name>`	Panel UI Element. Get title string.	`<str>`
?UPp	`<idx|name>`	Panel UI Element. Get index of parent ui element.	`<idx>`
?UPi	`<idx|name>,<arg>`	Panel UI Element. Get index of panel item. Arg is item number (e.g. 0 - (N-1)).	`<idx>`

Table 5: Summary of UI packet types