Sequencer RPC Protocols

Version	Date	Author	Comment
1.0	11/20/15	Wei Wang	Initial Draft
1.1	11/23/15	Wei Wang	Added the status command in the application layer
1.2	12/18/15	Wei Wang	Now the run request takes a e_traveler param.

1. Protocol Layer

Request

These are the valid fields in a request:

Field Names	Content of the Field
jsonrpc	Version of the protocol
id	A unique id that identifies this request
function	The name of the function
params	This is a positional arguments list for the function

The request will be serialized into json format before being sent over the socket.

Response

These are the valid fields in a response:

Field Names	Content of the Field
jsonrpc	Version of the protocol
id	The id from the request.
error	This field only exists if there is an error. This field and the result field can not exist at the same time. There are two sub fields of this field: code and message. (in other words, this field should be a dictionary data type before it's serialized into json strings). code: A negative integer number unique to the types of error. message: description of the error

Field Names	Content of the Field
result	The returned value of the function call. If the function call does not returned a value, the result field should be an empty string. The fact the result field exists and the error field does not exist means the function executes successfully.

The response will be serialized into json format before being sent over the socket.

2. Application Layer

Below are the functions supported by the sequencer. The parameters and the results of each function are explained. In this table, result is the value of the result field in the roc response documented in the protocol layer.

Note that we don't use the result field here to hold error message. All of these functions raise an exception if an error is encountered. The dispatch layer takes care of putting the information from the exception in an error response and sent back to the client

Request			Response
function	**params**	**effect**	**Result**
status	status string	returns the sequencer status. If a timeout error is returned, the client can assume the sequencer server is not started or has hang.	There are three possible statuses: • ***NONLOADED***: no sequence has been loaded. All commands that require sequence to be loaded will return error. • ***READY***: Sequence is loaded to be run • ***RUNNING***: A test sequence is being run
run	etraveler	Runs the sequencer from beginning to end. Stop on fail in general. However if a step's function is parse, continues to run until the next detect function. If you use the run command the sequencer will not stop at breakpoints. etraveler is a dictionary of information about the DUT being tested. Right now the only allowed key in e_traveler is 'attributes". This item must be a dictionary, and all the item in the "attributes" dictionary will be reported by the reporter as an DUT attributes. etraveler can be None	True or False. True means the test sequence has passed. False means the test sequence has failed

Request			Response
function	**params**	**effect**	**Result**
load	test_plan_path	Load a test plan in the sequencer. If a test plan is already loaded then the new test plan override the old test plan	A message: [test_plan_path] has been loaded
show	variable_name	Gets the value of a variable maintained by the sequencer	the value of the variable
jump	line_number or test_group_name or tie	if the parameter is an integer, jumps to that line (line is 1 based, not 0 based) in the test sequence. If the parameter is a TID, jumps to that test. If the parameter is a test group name, jumps to the first test of that group. After this function returns, next step in the test sequence to be executed is the destination of this function	Returns a list. The first item is the line number of the destination, the second item is a string representation of the information of the test, like this: “{‘FUNCTION’:relay, ‘PARAM1’: BATTERY_POWER, …}”
list	lines if no lines is specified, default to 10 lines	List the [lines] number of test items surrounding the next item to execute. [lines]/3 lines before the next line to execute is displayed	A list of lists. The first element of the list is a list consisting of [next_line_to_excute, line#_of_the_first_test_item_returned, line#_of_the_last_test_item_returned]. The rest of the elements represent of the test items returned. Each one is a list consisting of the line# of the test item, and a string representation of the test item
next	none	returns the line number of the next test item to run	line number of the next test item to run
step	none	execute the next item. This executes only one test item but if the next item should be skipped per KEY/VALUE, the sequencer will advance to the next item that should be executed and execute that	a list represents the actual test item just executed. The content of the list is of the same format as the list returned by the jump command. If we are already at the end of the test sequence, returns None and resets the sequencer’s internal PC to 0
abort	none	abort the running sequence. reset the pc to 1, clear all variables	true or false, indicating if the abort has been successful
wait	timeout	Now that the run is a non blocking call. The client can use the wait command to block until the current running sequence ends.	True if wait returns because of timeout. False if the sequence ends normally. If timeout=0 then no time out.

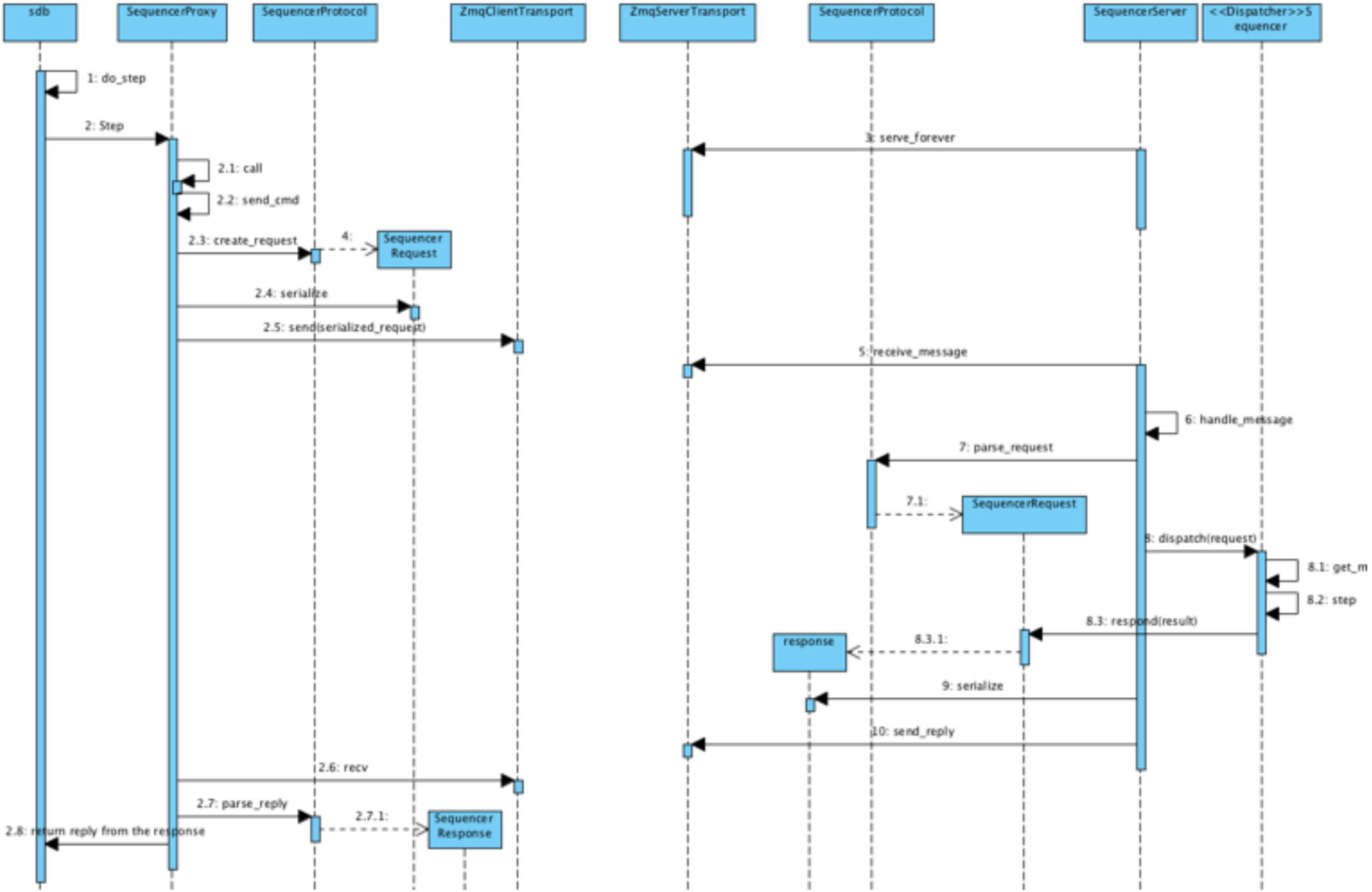
3. sdb commands

sdb is a command line sequence debugger. It's a remote client of the sequencer server. It adds some debug features on top of the function supported by the sequencer rpc server. These added functions are:

Command	Params	Effect
continue	none	runs the sequence from the next item to be executed until a breakpoint of the end of the sequence is encountered. The sequencer has no knowledge of breakpoints. All breakpoints are maintained by sdb. Internally *continue* does not call the run function, because run function always starts from the beginning of the sequence and does not stop at breakpoints. *continue* instead calls the step function repeatedly.
break	line no	Add line no to the break point list
all	none	show all break points
timeout	timeout value in ms	change the timeout value for all subsequent calls to sequencer. If timeout = 0, then no timeout

Below is the output from a sample run of sdb.

```
sdb>list
-> 1: INTELLIGENT  | INTEL_HOG_100_STAT_UNITSTAGE | station | Get Station Type |
   2: INTELLIGENT  | INTEL_HOG_110_CHAN_CHANNELID | channel | Get Channel ID |
   3: BOOT THE UNIT  | BOOT_BATT_100_RELA | relay | Connect the Battery | BATTERY_POWER |
   4: BOOT THE UNIT  | BOOT_BATT_110_SUPP | supply | Supply 3.85V to PP_BATT_VCC | PP_BATT_VCC |
3.85
   5: BOOT THE UNIT  | BOOT_BATT_120_DELA | delay | Delay | 2000 |
   6: BOOT THE UNIT  | BOOT_BATT_130_BUTT | button | Press BUTTON_TO_PMU_BTN_L |
BUTTON_TO_PMU_BTN_L |
   7: BOOT THE UNIT  | BOOT_DIAGS_100_DETE | detect | Get into :-) | :-) |
   8: SYSCFG  | SYSCFG_OFF_100_DIAG | diags | Write Tristar Host ID | tristar --wr_hostid 0x04 |
   9: SYSCFG  | SYSCFG_OFF_110_DIAG | diags | Initialize SysCfg | syscfg init |
   10: SYSCFG  | SYSCFG_MLB_100_DIAG | diags | Print MLB SN | syscfg print MLB# |
sdb>break 5
sdb>break 10
sdb>all
  5
  10
sdb>step
Just executed:
   1: INTELLIGENT  | INTEL_HOG_100_STAT_UNITSTAGE | station | Get Station Type |
sdb>step
Just executed:
   2: INTELLIGENT  | INTEL_HOG_110_CHAN_CHANNELID | channel | Get Channel ID |
sdb>continue
   3: BOOT THE UNIT  | BOOT_BATT_100_RELA | relay | Connect the Battery | BATTERY_POWER |
   4: BOOT THE UNIT  | BOOT_BATT_110_SUPP | supply | Supply 3.85V to PP_BATT_VCC | PP_BATT_VCC |
3.85
BREAK: -> 5: BOOT THE UNIT  | BOOT_BATT_120_DELA | delay | Delay | 2000 |
sdb>continue
   5: BOOT THE UNIT  | BOOT_BATT_120_DELA | delay | Delay | 2000 |
   6: BOOT THE UNIT  | BOOT_BATT_130_BUTT | button | Press BUTTON_TO_PMU_BTN_L |
BUTTON_TO_PMU_BTN_L |
   7: BOOT THE UNIT  | BOOT_DIAGS_100_DETE | detect | Get into :-) | :-) |
```


sdb
SequencerProxy
SequencerProtocol
ZmqClientTransport
ZmqServerTransport
SequencerProtocol
SequencerServer
<<Dispatcher>>Sequencer
1: do_step
2: Step
2.1: call
2.2: send_cmd
2.3: create_request
4:
Sequencer Request
2.4: serialize
2.5: send(serialized_request)
3: serve_forever
5: receive_message
6: handle_message
7: parse_request
7.1:
SequencerRequest
8: dispatch(request)
8.1: get_m
8.2: step
8.3: respond(result)
8.3.1:
response
9: serialize
10: send_reply
2.6: recv
2.7: parse_reply
2.7.1:
Sequencer Response
2.8: return reply from the response

```
   8: SYSCFG  | SYSCFG_OFF_100_DIAG | diags | Write Tristar Host ID | tristar --wr_hostid 0x04 |
   9: SYSCFG  | SYSCFG_OFF_110_DIAG | diags | Initialize SysCfg | syscfg init |
BREAK: -> 10: SYSCFG  | SYSCFG_MLB_100_DIAG | diags | Print MLB SN | syscfg print MLB# |
sdb>continue
   10: SYSCFG  | SYSCFG_MLB_100_DIAG | diags | Print MLB SN | syscfg print MLB# |
   11: SYSCFG  | SYSCFG_MLB_110_PARS_MLBSN_VERIFY | parse | Parse MLB SN | {{mlbsn}}
   12: SYSCFG  | SYSCFG_WMAC_100_DIAG | diags | Print Address | syscfg print WMac |
   13: SYSCFG  | SYSCFG_WMAC_110_PARS_WMAC_VERIFY | parse | Parse Address | {{wmac}}
   14: INFANCY  | INF_ACT_NTC3_PARS_ACTIVE_ADC_TEMP3 | parse | Measure NTC3 (10K) | {{temp3}}
   15: INFANCY  | INF_ACT_NTC4_PARS_ACTIVE_ADC_TEMP4 | parse | Measure NTC4 (R2195) | {{temp4}}
   16: CAL  | CAL_BUCK0_B0_MEAS_SLEEP1_BUCK0_CAL_VALUE | measure | Measure PP0V6_SLEEP1_BUCK0 |
PP0V6_SLEEP1_BUCK0
   17: CAL  | CAL_BUCK0_140_CALC_SLEEP1_BUCK0_CAL_VALUE_MV | calculate | Convert V to mV |
smokeyshell -e 'print([[sleep1_buck0_cal_value]]*1000)'
sdb>
```

4. Sequence diagram for a sample interaction:

This is the sequence diagram of a step call. It shows how the application layer, dispatch layer, protocol layer and transport layer work together on both sides of the rpc call.