Application Note

74-0050-180915

Performing Real Time Data Acquisition With ThinkRF R5500 Real-Time Spectrum Analyzer

This application note explains the different methods of real time data acquisition with ThinkRF R5500 Real-Time Spectrum Analyzer, includes providing the SCPI commands to use with each method. These methods are single block capture, stream capture with limited capacity, and complex sweep setup to capture one or more blocks of data.

thinkRF

monitor. detect. analyze

Contents

Overview

The R5500 real time data acquisition can be done in three ways:

a) a single block of data capture per user's request,
b) stream capture but with limited memory, or
c) through a complex sweep setup of one or many captures.

The capture is handled by a controller engine, which provides to users a mean of defining and performing the capture methods using commands created basing on Standard Commands for Programmable Instruments (SCPI) protocol. The methods could be conditioned with decimation, and/or triggering.

This paper explain in details the capture methods and the SCPI commands to use for each method and focuses on the data acquisition aspect only. It assumes that the users already have the knowledge of connecting and communicating to a R5500. It is important to refer to the *R5500 Programmer's Guide* for details on VRT protocol, SCPI commands, and usage.

Note: More examples, beside those provided in this document, are available in each API folder provided in a R5500 Release Package (see http://www.thinkrf.com/firmware-updates/).

Data Acquisition System and Process

1 illustrates the overall R5500 functional hierarchy including data acquisition and control, starting from the user's application end to the functional layers within the R5500.

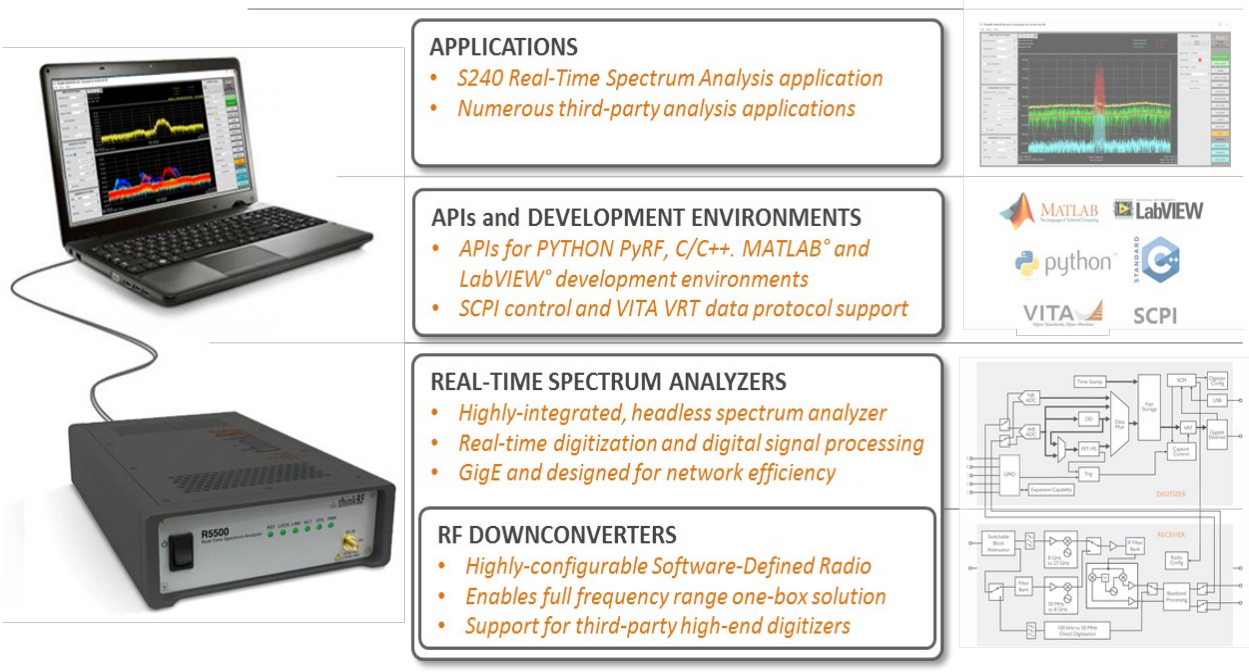

Figure 1: R5500 functional hierarchy
and users' application end interface methods to the device

The R5500 is a network ready device, communicating control interface and data through a network using ThinkRF provided APIs (some are listed in the figure) or through users' own or a third-party application. All applications, regardless of the programming language, would be using the SCPI commands to interface with the device and following VRT[1] protocol to decode the data, including its related context information.

R5500 has an on-board, fast data storage memory of 128 MBytes. Each capture method will be constrained by that memory size, as explained in the following sections. To minimize unnecessary storage, a data acquisition could further be

- conditioned using a trigger event, whether through a complex synchronized sweep setup using external pulse or synchronized word input or through a simple yet powerful internal frequency level detection trigger engine (see AppNote 74-0046 Triggering Features of the R5500); and/or
- down-sampled using decimation, with supported rates of 4 to 1024 (in power of 2).

When a trigger method is used, the capture starts after the trigger event has occurred.

Definition and Process

The “block” of data to be captured is continuous and contiguous in nature, and it is termed as a “trace capture”. The data from one block to another, however, would not be continuous as it is a “different” block. The block size is ranging from 256 samples to a maximum determined by the device's memory size (mentioned above) and the RFE data format.

A typical capture setup includes

- determining the capture method that best fit the application;
- sending the appropriate device and data configuration SCPI commands (or using API functions), directly in standalone mode or through a network; then
- at the end, issuing the appropriate capture start command depending on the method used.

The R5500 capture controller initiates the data capture and data storage to an on-board memory, with proper VRT headers and trailer inserted. In other words, data captured is sectioned into VRT packet(s), with the packet size specified through samples-per-packet (SPP) parameter. As soon as a complete data packet is available, the embedded firmware starts processing and sends VRT data packets to the end user. The firmware also creates and sends any associated VRT context packets back to the user. A notable note, included in the VRT headers are the timestamp of when the data packet is captured, basing on the system clock, in nanoseconds.

See the R5500 Programmer's Guide for more information.

1 VRT stands for the VITA-49 Radio Transport protocol, used for digitized data and its associated context information. See R5500 “Programmer's Guide” for further information.

Capture Setup Requirement

The following setup is required after successfully connected to an RTSA and before starting the capture:

- Reset the system by issuing `*RST` command
- Request a system lock for data acquisition using `:SYSTem:LOCK:REQuest? ACQuisition` command
- Flush the RTSA's internal buffer through `:SYSTem:FLUSh` command. Issue this command only after having the acquisition lock request.

See the R5500 Programmer's Guide for more information on the commands.

Block Capture

To do a single block capture of continuous data, the total number of samples captured (a single block) is determined by SPP (`:TRACe:SPPacket`) and number of packets-per-block or PPB (`:TRACe:BLOCk:PACKets`). The SPP size is limited by the VRT protocol, and the (SPP * PPB) block is limited by the device's memory. When the block data capture command (`:TRACe:BLOCk:DATA?`) is issued, the R5500 will start the capture and store the total number of samples into a buffer. The data, as mentioned earlier, are sectioned into VRT packets, each of size SPP with 5 headers word and a trailer word inserted. The `:TRACe:BLOCk:DATA?` command must be issued again, with or without changing any configuration, to start another capture block. Hence, the samples within a single block capture is continuous from one packet to the other, but not necessary between successive block capture commands issued.

The following examples illustrate how to do a single block capture and another with triggering. Note, the examples are written in python language. SCPI commands are used in the example and thinkRF's pyRF API functions are commented right beside some `scpiset()` to indicate the equivalent functions availability.

Example 1 – A single block capture

```
#####
# Capture a block of 131072 ZIF samples (I14Q14) with the VRT packet
# size set to 4096 and 32 packets requested
#####

# import required libraries
import sys
from pyrf.devices.thinkrf import WSA
from pyrf.util import collect_data_and_context

# Parameters for configuring the RTSA for a block capture
SPP = 4096
PPB = 32
CENTER_FREQ = 2450 * 1e6
RFE_MODE = 'ZIF'
DEC_RATE = 0 # for no decimation
```

```
# Define the RTSA device
dut = WSA()

# connect to RTSA device with a given IP address
dut.connect(sys.argv[1])

# reset device to default settings
dut.scpiset(":SYSTEM:LOCK:REQ? ACQUISITION")  # dut.request_read_perm()
dut.scpiset(":*RST")                          # dut.reset()
dut.scpiset(":SYSTEM:FLUSH")                  # dut.flush()

# set RFE mode to ZIF, which yields I14Q14 data
dut.scpiset(":INPUT:MODE " + RFE_MODE)        # dut.rfe_mode(RFE_MODE)

# does some device configuration, such as set frequency
dut.scpiset(":FREQ:CENTER " + CENTER_FREQ)    # dut.freq(CENTER_FREQ)

# uncomment to set the desired decimation rate, default is off
#dut.scpiset(":SENSE:DEC " + str(DEC_RATE))   # dut.decimation(DEC_RATE)

# configure and capture the required block of data
dut.scpiset(":TRACE:SPP " + str(SPP))         # dut.capture(SPP, PPB)
dut.scpiset(":TRACE:BLOCK:PACKETS " + str(PPB)
dut.scpiset(":TRACE:BLOCK:DATA?")

# read the block of data and any context packets from the R5500
for i in range(PPB):
    data, context = collect_data_and_context(dut)
```

Example 2 – A large block capture with frequency level trigger

```
#####
# Perform a large capture of 32,768,000 SH samples (16384 * 2000),
# conditional to a level trigger setting of range 2400 MHz – 2500 MHz
# with an amplitude of -70 dBm
#####

# import required libraries
import sys
from pyrf.devices.thinkrf import WSA
from pyrf.util import collect_data_and_context

# set a large block size 16384 * 2000 or 32 Msamples
SPP = 16384
PPB = 2000
RFE_MODE = 'SH'
CENTER_FREQ = 2450 * 1e6
#TRIGGER_SET = {'type': 'LEVEL','fstart': 2400 * 1e6,'fstop': 2500 *
1e6, 'amplitude': -70}

# define the RTSA device
dut = WSA()

# connect to RTSA device with a given IP address
dut.connect(sys.argv[1])

# reset device to default settings
```

```
dut.scpiset(":SYSTEM:LOCK:REQ? ACQ")          # dut.request_read_perm()
dut.scpiset(":*RST")                          # dut.reset()
dut.scpiset(":SYSTEM:FLUSH")                  # dut.flush()

# set RFE mode to SH, which yields I14 data
dut.scpiset(":INPUT:MODE " + RFE_MODE)        #dut.rfe_mode(RFE_MODE)

# does some device configuration, such as set frequency
dut.scpiset(":FREQ:CENTER " + CENTER_FREQ)    # dut.freq(CENTER_FREQ)

# configure the trigger setting and enable it
dut.scpiset(":TRIGGER:LEVEL 2400 MHz, 2500 MHz, -70 dBm")
dut.scpiset(":TRIGGER:TYPE LEVEL")            # dut.trigger(TRIGGER_SET)

# configure and capture the required data
dut.scpiset(":TRACE:SPP %s" % SPP)            # dut.capture(SPP, PPB)
dut.scpiset(":TRACE:BLOCK:PACKETS %s" % PPB)
dut.scpiset(":TRACE:BLOCK:DATA?")

# read the data and any context packets from the R5500
for i in range(PPB):
    data, context = collect_data_and_context(dut)
```

Stream Capture

With stream capture, data packets will be 'pushed' from the R5500 whenever data is available (as opposed to the block capture mode, which data is being “pulled” per user's request). Since it is streaming, only SPP needs to be specified, not PPB. Once all the device and data capture configuration commands are sent, issue `:TRACE:STREAM:START` (not `:TRACE:BLOCK:DATA?`) command to start the streaming, and `:TRACE:STREAM:STOP` to stop.

The data samples are continuous and contiguous until memory overflow occurs, which would happens very fast. Since the R5500 has a fast system clock rate of 125 MHz for data capture with a limited memory storage and the network interface transfer rate is much slower than that of the system clock, data overflow would bound to happen. After which, the capture system will sustain the capture at a best effort as the transfer rate to the host could not match up to the capture rate. With this limitation, it is recommended that stream capture **should only** be used with a high decimation rate (such as 16 or higher) to slow down the sample rate.

Example – A stream capture

```
#####
# Perform a stream capture of ZIF data with a decimation rate of 16
#####

# import required libraries
import sys
import msvcrt
from pyrf.devices.thinkrf import WSA
from pyrf.util import collect_data_and_context

# set the VRT packet size
SPP = 4096
```

```
CENTER_FREQ = 2450 * 1e6
RFE_MODE = 'ZIF'
DEC_RATE = 16

# define the RTSA device
dut = WSA()

# connect to RTSA device with a given IP address
dut.connect(sys.argv[1])

# reset device to default settings
dut.scpiset(":SYSTEM:LOCK:REQ? ACQ")         # dut.request_read_perm()
dut.scpiset(":*RST")                         # dut.reset()
dut.scpiset(":SYSTEM:FLUSH")                 # dut.flush()

# set RFE mode to ZIF, which yields I14Q14 data
dut.scpiset(":INPUT:MODE %s" % RFE_MODE))    # dut.rfe_mode(RFE_MODE)

# does some device configuration, such as set frequency
dut.scpiset(":FREQ:CENTER " + CENTER_FREQ)   # dut.freq(CENTER_FREQ)

# configure the VRT packet size
dut.scpiset(":TRACE:SPP %s" % SPP)           # dut.spp(SPP)

# set the decimation rate to slow down the capture rate
dut.scpiset(":SENSE:DEC %d" % DEC_RATE))     # dut.decimation(DEC_RATE)

# Start the stream capture
dut.scpiset(":TRACE:STREAM:START")

# read the stream data and any context packets from the R5500
total_pkts = 0
while True:
      data, context = collect_data_and_context(dut)

      # optional, just to indicate the stream capture is still running
      total_pkts = total_pkts + 1
      if total_pkts % 100 == 0:
            print('.')

      # Add your conditional code here so to stop the stream
      # capture or just Ctrl+C to exit the program. For example:
      # when detect a key stroke, exit
      if msvcrt.kbhit():
            dut.scpiset(":TRACE:STREAM:STOP")   # dut.stream_stop()
            print
            break

print "Total packets captured: %d" % total_pkts
```

Sweep Capture

The sweep capture control provides the ability to define and execute simple or complex sweeps, in which each sweep entry defined a capture and device configuration as with the block capture. To capture a block of data in an entry, use `:SWEEP:ENTRY:SPP` and `:SWEEP:ENTRY:PPB` as

with the block method. Once all sweep entries are created, issue `:SWEEP:LIST:START` to start the sweep as well as the capture. The engine will stop when the iterations (`:SWEEP:LIST:ITERATION`) have been reached or either a `:SYSTEM:ABORT` or `:SWEEP:LIST:STOP` command has been issued. The iteration is defaulted to 0, which means infinite looping.

If a trigger is defined for an entry, captured data is returned only when the trigger event occurred. Otherwise, when the trigger `:DWELL` time is reached, the trigger is aborted and the next sweep entry will be executed. During sweeping, the R5500 internal buffer might be overflown, at which point the sweep engine will pause. The engine will resume sweeping once there are enough space for the next "block" of data or more.

Example – A sweep capture with multiple entries and a trigger setup

```
#####
# Create multiple sweep entries with different configuration per
# entries as an example of the sweep capabilities.
#
# This example makes use of pyRF functions as well as direct SCPI
# commands
#####

# import required libraries
import sys
import time
from pyrf.devices.thinkrf import WSA
from pyrf.numpy_util import compute_fft
from pyrf.util import collect_data_and_context

# setup RTSA and connect
dut = WSA()
dut.connect(sys.argv[1])

# use thinkRF's pyRF functions to initialize the unit, instead of
# direct SCPI here
dut.abort()
dut.flush()
dut.reset()
dut.request_read_perm()

# clear any potential existing list
dut.scpiset("SWEEP:ENTRY:DELETE ALL")        # dut.sweep_clear()

# create first entry
dut.scpiset("SWEEP:ENTRY:MODE DD")
dut.scpiset("SWEEP:ENTRY:FREQ:SPP 2048")
dut.scpiset("SWEEP:ENTRY:SAVE 0")

# create second entry
dut.scpiset("SWEEP:ENTRY:MODE ZIF")
dut.scpiset("SWEEP:ENTRY:FREQ:CENTER 62.5 MHZ, 8000 MHZ")
dut.scpiset("SWEEP:ENTRY:FREQ:STEP 100 MHZ")
dut.scpiset("SWEEP:ENTRY:FREQ:SPP 2048")
```

```
dut.scpiset("SWEEP:ENTRY:FREQ:PPB 10")
dut.scpiset("SWEEP:ENTRY:DEC 8")
dut.scpiset("SWEEP:ENTRY:SAVE 0")

# create third entry, but using the pyRF function
s = SweepEntry(
            fstart=62.5 * M,
            fstop=8000 * M,
            fstep=25 * M,
            fshift=0,
            decimation=1,
            spp=4096,
            ppb=100,
            rfe_mode='SH')
dut.sweep_add(s)

# set iteration to infinite
dut.sweep_iterations(0)

# start sweeping
dut.scpiset("SWEEP:LIST:START")             #dut.sweep_start()

while True:
      data, context = collect_data_and_context(dut)
      print hex(data.stream_id), context['rffreq']

      # add some conditional code here to stop the sweep & exit the
      # loop or set iteration to a non-zero value
      # dut.scpiset("SWEEP:LIST:STOP")
```

Document Revision History

This section summarizes document revision history.

Document Version	Release Date	Revisions and Notes
v1.0	Sept 01, 2017	First release
v1.1	Feb 15, 2018	Added Capture Setup Requirement section

Contact us for more information

ThinkRF Support website provides online technical documents for ThinkRF products at http://www.thinkrf.com/resources.

For all customers who hold a valid end-user license, ThinkRF provides technical assistance 9 AM to 5 PM Eastern Time, Monday to Friday. Contact us at https://www.thinkrf.com/support/ or by calling **+1.613.369.5104**.