Design Process for Smart, Distributed RF Sensors

Contents

Application Note Number: AN-5501

Authors: John Smith and Jane Smith

Last Modified Date: 2016/04/15

Reference: https://www.ettus.com/content/files/kb/application_note_distributed_rf_sensors.pdf

The Ettus ResearchTM USRPTM (Universal Software Radio Peripheral) is a flexible, low-cost software defined radio (SDR) used to create many capable and unique RF systems, including intelligent RF sensors. As use of wireless communications continues to grow rapidly, limited RF spectrum becomes more crowded. This creates an increased demand for RF sensors that can monitor the spectrum in an effort to measure wireless traffic conditions, detect illegal interferes, or perform other signals intelligence tasks. For example, the evolution of whitespace radios depends on RF monitoring capability to reduce co-channel interference between competing devices. Organizations in the civilian, defense, and homeland security segments all rely on RF sensors in greater numbers to identify, characterize, and locate various targets.

The USRP N210 and USRP E110 are a complimentary set of devices enabling rapid development of intelligent RF sensors. The USRP N210 is intended to operate as a host-based software radio platform. A Gigabit Ethernet (GigE) interface provides a digital, baseband interface for a receive and transmit chain. GNU Radio, along with other software frameworks and tools such as GNU Radio Companion (GRC), accelerate custom radio development.

The USRP E110 is an embedded platform. Rather than offloading application-specific DSP operations to an external host, the ARM processor of the USRP E110 performs these operations internally. This processor runs a custom built Angstrom Linux distribution. The factory-image supplied on an SD card also includes GNU Radio and GRC.

This allows developers to implement intelligent RF sensors with a USRP N210 in a lab setting where various algorithms can be easily tested. After validation, these algorithms can then be deployed on a USRP E110. This process can be as simple as transferring the associated files and executing the applications through the network interface provided by the USRP E110.

This application note assumes you have some familiarity with the USRP products, GNU Radio, and GRC. If this is not the case, you can find suggested reading at the end of the document that will clarify many things discussed in this document. This application note is also applicable to the USRP N200 and USRP E100.

As the need for distributed RF sensors increases, a number of issues arise. In some cases, RF sensors may only relay raw data back to a central processing location. While this has some advantages, systems relying on the transfer of raw RF samples require significant back-haul infrastructure to support large bandwidths. This data might be transferred to some central processing location, requiring large, sophisticated, and expensive processing platforms. In many cases, signals collected are recorded for post processing, thus requiring significant storage capability.

Rather than supplying data without regard for events in the detection environment, an intelligent RF sensor applies various techniques to reduce the data. In turn, this reduces requirements for back-haul infrastructure, data storage, and centralized processing. Deploying smart RF sensors requires some knowledge about the signals of interest. This can present some disadvantages if the distributed sensors are not flexible and easily reconfigured for new conditions.

The availability of tools such as the USRP product family and GNU Radio, provides a path to develop intelligent RF sensors with ease and agility. Software radios, such as the USRP, and flexible development tools are key pieces of a strategy of deploying smart, distributed RF sensors.

Error creating thumbnail: File missing

The USRP Hardware Driver (UHD) provided by Ettus Research allows users to develop software defined radios in a GNU Radio, LabVIEW, Matlab, Simulink, or custom SDR frameworks. Operating in Windows, Linux, or MacOS, UHD provides an abstraction layer between the software radio and the USRP device. This allows applications developed with UHD to maintain compatibility with all USRP models.

UHD provides an easy method to move SDR applications across USRP platforms. In this example, applications are developed on a host-PC with a USRP N10, and then seamlessly migrated to a USRP E110. This enables the flexibility required for changing RF environments and signal collection missions.

The USRP N210 is a member of the Network Series of products. It is generally connected to a host computer via a Gigabit Ethernet interface that can provide up to 25 MS/s of 16-bit samples, and 50 MS/s of 8-bit samples. It contains a Spartan 3A-DSP 3400 FPGA, a 100 MS/s, 14-bit ADC, a 400 MS/s, 16-bit DAC, and other components to support operation. The USRP N210 also provides various mechanisms to synchronize multiple units for phase coherent operation. This can be accomplished with an Ettus Research MIMO cable, or with externally distributed 10 MHz and 1 PPS signals. The USRP N210 also includes an optional GPS disciplined oscillator for synchronization over wide areas.

The USRP E110 is a unique device that incorporates an ARM-based computer-on-a-module (COM). This embedded computer is shipped with a fully functional Linux distribution with GNU Radio and other useful software installed. This makes the USRP E110 ideal for standalone operation as a smart, distributed RF sensor. The USRP E110 provides several interfaces to achieve application requirements: HDMI output, audio in/out, USB, 10/100 Ethernet ports, and a USB based debug port. Multiple USRP E110s can be time synchronized with external 10 MHz and 1 PPS inputs. An optional GPSDO module can also be used to time synchronize multiple USRP E110s over a wide geographic area. The USRP E110 uses the same FPGA as

the USRP N210, a 12-bit, 64 MS/s ADC, and 14-bit 128 MS/s DAC. Both models are supported by UHD.

Error creating thumbnail: File missing

The portability offered by UHD allows the user to exploit a wide array of development tools. These include graphical development environments, text-based programming languages such as C++ or Python, and in some cases, third-party products with application-specific functions. In this example, GNU Radio and Python are used to generate a smart RF detection algorithm. This algorithm is verified in a lab setting with an RF signal generator. Finally, the application is deployed for operation on a ?remote? USRP E110. An illustration of this development concept can be seen in Figure 1.

Error creating thumbnail: File missing
Several other tools are at your disposal when development occurs on a host-based PC. Sophisticated development environments, such as LabVIEW or MATLAB, can be used for various pre-and-post-processing functions to verify these algorithms. In some cases you may also modify FPGA designs to support the algorithm and transfer these designs to the USRP E110 with the other software components.

In the end, this approach combines the best of both worlds ? accessibility and convenience on a host-based platform, and the deployability of the USRP E110.

To illustrate the power of this development concept, Ettus Research assembled a simple demonstration. The objective of this example is to implement a straight forward power detection algorithm on the USRP E110. The goal is to simultaneously listen on five consecutive 25 kHz channels, and issue a report to the back-haul network when transmission starts or subsides in each of those channels. For clarity, the channel arrangement for the detection algorithm is shown in Figure 3. Power detection and threshold operations are performed on each channel independently. The output of the threshold operation will be used to generate reports of transmitters, starting and stopping operation in the particular channel. These transitions will be reported to another entity on the back-haul network with a specific IP address and port number.

Simple algorithms like this are often used to detect the presence of interferers within shared spectrum. More sophisticated algorithms may also be helpful in calculating the density of RF communications on a particular channel, such as those used for whitespace communications.

Error creating thumbnail: File missing
In order to meet the detection requirements for this example, you have several tools heart your disposal. Graphical tools like GRC, are useful for DSP development, but text- based programming methods may also be used. Figure 4 shows several components used to implement this example. The USRP device provides the hardware required to sample the RF spectrum. UHD provides a common API for all USRP devices. GNU Radio provides stream-based DSP capability, and the Python script provides the developer an easy way to implement reporting logic. Finally, the detection reports re sent to the network with a socket interface from the Python script.

Error creating thumbnail: File missing

The implementation used to meet these objectives is fairly straight forward. GNU Radio is used to perform stream based processing tasks, such as channelization, power detection, and thresholding. GRC was used to synthesize code for these functions. The general architecture for this DSP chain is shown in Figure 5, and screenshot of the flowgraph as drawn in GRC can be seen in Figure 6 and Figure 7. This flowgraph outputs a stream of bytes to a file. The five LSBs of the byte represent the power detection state of each channel. A Unix Pipe (FIFO) is used as an inter-application interface to another program, which performs additional processing.

Error creating thumbnail: File missing
A complex multiplication and sine signal source set to the channel offset frequency is used to perform frequency translation of the baseband signal. Each channel is low pass filtered with a bandwidth of 18 kHz. Other components such as the Frequency Translating FIR Filter could be used to combine these to functions in a single block. GNU Radio also provides a Polyphase Channelizer block. This more discrete implementation was chosen for the sake of illustration. An RMS function with parameterized time constant is used to estimate the power of each channel. The user-defined threshold is subtracted from the result and a binary slicer is used to provide a binary indication of RF power. The bit for each channel is effectively shifted to the corresponding channel position within a single byte using multiplication and addition. This is output to an external file sink ? in this case, set to a pipe named ?detect_fifo.?

The flowgraph includes several parameters, such as detection threshold, and center channel frequency, set through a command line interface. This allows you to adjust for changing targets and environments without changing hard-coded values.

Error creating thumbnail: File missing
Figure 6 - GNU Radio Flow Graph - Channelization w/ Complex Multiplication and Complex LPF

Error creating thumbnail: File missing
While GNU Radio is ideal for stream based DSP operations, sometimes other tools such as text-based programming languages are more appropriate for asynchronous or logical programming. A Python script was used to implement a simple state machine to produce reports for transmit start (?Tx On?) and transmit stop (?Tx Off?) events. Accepting bytes from the pipe, which is fed by the GNU Radio flowgraph, the Python application interprets bit transitions, and generates textual reports. These textual reports are output to a UDP socket. This UDP socket shares this information with other devices on the back-haul network. The code for this example can be found in the appendix.

Plugging in a host-based USRP device, such as the USRP N210, this program can be executed and the algorithms can be validated with real RF signals before remote deployment.

During and after algorithm development, it is important to validate the performance of the application with reference signals. The simple nature of this examples, allows the algorithms to be tested with a run-of-the-mill RF signal generator. A block diagram of the apparatus used for verification can be seen in Figure 8.

Error creating thumbnail: File missing
The signal generator was used to produce RF signals of various amplitudes and frequencies while the applications were running. Turning RF on, and sweeping the frequency from below the lowest channel to a value above the highest channel produces a report, such as the one shown in Figure 10. This makes intuitive sense if you refer to the spectral illustration of Figure 3. The default threshold and gain values set in the application correspond to a signal power of approximately -100 dBm with an SBX daughterboard. The application generates a report when a transition across this boundary occurs in each channel. As the frequency is swept, the power may exceed this threshold in an adjacent channel before falling below the threshold in the current channel. This is due to a course frequency step moderate overlap in the passband of the channel filters.

Error creating thumbnail: File missing
Figure 9 - Host Execution of Flow Graph (900 MHz, threshold -70 dBfs, 250 kS/s)

Error creating thumbnail: File missing
Figure 10 - Host Execution of smart_rf_sensor_demo.py - RF Sweep Across Channelizer Centered at 900 MHz

After developing and testing a new RF detection algorithm, this algorithm can be deployed on the USRP E110 platform. In many cases this is as simple as transferring the files to the USRP E110 SD card and executing the applications. The USRP E110 can be accessed through an SSH connection, available through the 10/100 Ethernet port. You can transfer these files by mounting the USRP file system with gvfs-mount. For more instructions on this items see the USRP E100/110 FAQ:

http://code.ettus.com/redmine/ettus/projects/usrpe1xx/wiki/FAQ

After transferring the files, you can execute the application. If both programs have been copied to the home folder of the root account of the USRP E110, the commands to run the programs will conform to those shown in Figure 11 and Figure 12Error! Reference source not found.. Figure 13 shows the output of the application displayed on a remote machine listening on the appropriate UDP socket. In this case, the USRP E110 was located near and connected to an externally mounted antenna to allow observation of real-world signals. The USRP E110 is accessed remotely from an office computer. The flowgraph is tuned to the Family Radio Service (FRS) band, and the reports show bursting radio traffic. These reports are verified visually with a USRP N210- based spectrum analyzer connected to an adjacent antenna.

Error creating thumbnail: File missing

Figure 11 - SSH Execution of power_event_frontend_demo.py (464.4 MHz, threshold -70 dBFS, 125 kS/s)

Error creating thumbnail: File missing

Figure 12 - SSH Execution of smart_rf_sensor_demo.py (receive from PIPE named detect_fifo, output reports to 192.168.1.159:8002, verbose not enabled)

Error creating thumbnail: File missing

Figure 13 - Distributed Sensor Output - Received by Remote Host on Port 8002

The GNU Radio flowgraph and Python script were both implemented with command line arguments allowing you to adjust various parameters. For example, the noise floor and total gain of the SBX varies across frequency, so a log-threshold argument is used to accommodate for these variations. A complete list of these arguments can be provided by including ??help? in the command line. Successful deployment of this application may require adjustment of the log-threshold parameter. Different sample rates must also be specified because the USRP N210 output rate cannot be reduced down to 125 kS/s as the USRP E110 can.

When designing a portable application for USRP devices, it?s important to note the sample reference clock on the USRP N210 is fixed at 100 MHz. The USRP E100/E110 provides a flexible-frequency clocking solution, with a maximum and default frequency of 64 MHz. In both cases, the sample rate used by a portable application must be set to a factor of these frequencies. You must consider these differences when porting code from a USRP N210-based system to a USRP E110. This can be accomplished by choosing sample rates that are common factors of 100 MHz and 64 MHz. It is also possible to use a rational re-sampler configured to work with different data rates. If any of this is unclear, you should reference additional resources in the knowledge base for a better understanding of the USRP clocking systems.

You should also consider the difference in ADC and DAC resolution. The USRP E110 ADC and DAC provide 12 and 14 bits of resolution, respectively. The USRP N210 ADC and DAC provide 14 and 16 bit of resolution, respectively. This will impact spurious free dynamic range, and to a lesser extent, noise figure.

It is important to proceed with an understanding of the USRP E110?s processing capabilities. The OMAP processor used in the device is a low-power, small-form-factor computer. It does not provide the processing performance required to implement complicated algorithms at the same rates as a host-based system. This example runs at approximately 125 kS/s. 1 MS/s is the maximum rate typically achieved when GNU Radio is used on the USRP E110. However, this figure is highly dependent on the complexity of the flowgraph. Optimized routines can improve this figure. These can be implemented with custom frameworks, or with Vector Optimized Library of Kernels (VOLK), which is a component within GNU Radio facilitating deployment of efficient, vectorized math routines.

The power detection example used in this application note is intended to be fairly simple and easy to understand. This application may be useful for users looking to evaluate traffic in particular bands. However, it is more interesting to note you can deploy more complex algorithms than this simple power detector. For example, decoders for various digital standards can be integrated into the application to identify and tag RF targets. Also, some USRP users have made external provisions for direction finding. There is enormous potential to deploy smart, distributed RF sensors using the USRP E110 and the principles discussed in this document. The USRP E110 can also transmit signals to provide stimulus for various events.

Sophisticated users with experience in FPGA development can take advantage of the un-utilized resources of the Xilinx Spartan-3A DSP 3400 FPGA. This can drastically increase the processing performance of the system, and reduce the amount of data that must be handled by the slower OMAP processor. The DSP chains in the USRP E110 and USRP N210 are identical on the ?host-side? of the rate conversion blocks (decimators, interpolators). As with processor-based implementations, FPGA modifications can be verified in a lab with the USRP N210, and ported to the USRP E110. This requires compilation and timing verification for each model, which provides for a less streamlined deployment compared to this example implemented with GNU Radio.

This document presents a flexible method to deploy and reconfigure a system of smart, distributed RF sensors. The application presented shows a clear path to develop and test various DSP algorithms in the lab, and how to easily deploy them to remote sensors. This is appropriate for a variety of applications requiring agile use of distributed RF systems. If you have any questions about this document or the USRP devices, please refer to the Knowledge Base on the Ettus Research website or contact support@ettus.com. If you would like to make a sales inquiry, please contact sales at sales@ettus.com. You can access the GRC files and code for this example at:

http://files.ettus.com/app_notes/distributed_rf_sensors/smart_rf_sensor_demo.zip

http://gnuradio.org/redmine/projects/gnuradio/wiki

http://www.ettus.com

```
import time
import socket
import thread
import string
import struct
from optparse import OptionParser

CHANNEL_COUNT = 5

state_array = [0,0,0,0,0]    #channel power state array


def report_event(channel_num, event_string, network_socket, options):

     #assemble report string and output to udp socket
     report_string = time.asctime(time.gmtime(time.time())) + ',' + event_string + ',' + 'Chan. ' + str(channel_num)

     network_socket.sendto(report_string + "\n\r",(options.ip, options.port) )

     if(options.verbose):
             print report_string
```

```
def apply_detection_logic(file_object,network_socket,options):

     #get byte from fifo
     byte = int(struct.unpack('B', file_object.read(1))[0])

     #look at CHANNEL_COUNT LSBs to determine if power is detected
     for j in range(0,CHANNEL_COUNT):

             result = ( byte >> j ) & 1

             #apply simple logic to determine TX on/off transission
             if state_array[j] == 0:
                     if result == 1:
                             report_event(j,"Tx On",network_socket,options)
             if state_array[j] == 1:
                     if result == 0:
                             report_event(j,"Tx Off",network_socket,options)

             state_array[j] = result


def main():

     #command parser
     parser = OptionParser()
     parser.add_option("-p", "--port", dest="port",action="store",type="int",
             help="Port for outging UDP Socket.", metavar="PORT")
     parser.add_option("-a", "--address", dest="ip",action="store",type="string",
             help="IP Address for outgoing UDP Socket.", metavar="ADDR")
     parser.add_option("-f", "--file", dest="filename",action="store",type="string",
             help="File w/ incoming bytes", metavar="FILE")
     parser.add_option("-v", action="store_true", dest="verbose",help="Print reports to stdout.",
             default=False)
     (options, args) = parser.parse_args()

     print "The program is starting"

     #open file pipe for itnerface to flow graph
     print "Opening Pipe to GNU Radio Flow Graph"
     file = open(options.filename,"rb")

     #bind UDP socket
     print "Opening Outgoing UDP Socket"
     sock_network = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

     #process output from gnuradio flowgraph
     while(1):
             apply_detection_logic(file,sock_network,options)

if __name__ == "__main__":
    main()
```