DEBUGGING STRATEGIES FOR SCA COMPONENTS AND WAVEFORMS

Drew Cormier (Wireless@Virginia Tech: Blacksburg, VA, USA; acormier@vt.edu)
Carl Dietrich (Wireless@Virginia Tech: Blacksburg, VA, USA; cdietric@vt.edu)

ABSTRACT

In order to diagnose errors in a communications system, an SDR developer must have a debugging environment that is able to send and receive known signals to various ports in a waveform, as well as assess the output response exhibited by components of interest. The developer must also be able to understand the effects of latency in his/her system

These methods of diagnosis are illustrated using ALF, a free, open source graphical debugging environment that takes advantage of the SCA's use of CORBA to connect to component ports within a running waveform.

ALF is written in python for ease of development and maintenance. It includes several tools that can be used to monitor waveform performance and compare it with expectation. Users can develop and add their own tools to ALF as needed. Current tools include a software arbitrary waveform generator, a real-time spectrum and constellation plotter, and a signal sink. Provisions also exist for monitoring timing and determining latency throughout a waveform, enabling identification of components requiring buffering or optimization for real-time operation.

1. INTRODUCTION

The rapidly evolving nature of communications technology demands a debugging environment that is able to change based on the needs of the SDR designer; therefore the best debugging environments are those that are flexible. Certain fundamental tools are commonly used in debugging many communication systems, such as sources, sinks, and basic plotters. The use of sources, sinks, and plotters allow the developer to quickly identify the location of errors in the system. Even though each individual component in the system creates the correct input/output response, it is also important to know if the component is responding in a timely fashion relative to the other components in the system; therefore tools for analyzing timing are also needed.

These basic capabilities are realized through the use of ALF, an open source debugging and running environment that runs on the Open Source SCA Implementation :: Embedded (OSSIE), which is also open source. ALF was originally donated to Virginia Tech by SAIC in January 2007; since then members of the OSSIE development team have contributed various upgrades. The open source natures of the tool as well as the framework allow the developer to utilize the tool in whatever way is necessary for his/her system. Since the tool is developed in Python, it can be easily modified, in a timely manner, to meet new debugging requirements that arise with emerging technologies.

2. DEBUGGING METHODS

There are many criteria for determining if a software defined radio is functioning properly. First of all, the algorithm within each component must have the proper input/output response. Once the correct independent functionalities of the components in a waveform are verified, the overall algorithm of the waveform must be tested. In order to diagnose problems with the waveform, it is often advantageous to be able to diagnose sub-groups of connected components within the waveform. There are two schools of thought for the diagnosis of subgroups of components: one can either develop smaller waveforms and test them individually, or one can develop a set of tool sets that allows the developer to debug subsets of the waveform while the entire waveform is running. The use of standards, such as CORBA (the middleware used in the SCA), make this runtime debugging possible.

Having spec-compliant algorithms is not the only criteria for having a working radio. In the case of an SCA software defined radio, the SCA compliance of the radio must also be diagnosed. For example, a developer may wish to test the correct functionality of the Ports in a Resource.

The developer must often test his/her software under extreme and unusual conditions. Certain operating conditions occur so infrequently that it becomes necessary for the debugger to have the ability to feed artificial data to the system in order to verify the response of the system in "what if" scenarios. In many cases, this involves sending invalid data to the system in order to test the system's robustness or in order to confirm that the proper error messages are generated. In order to asses the functionality of a portion of the system, the time spent debugging can be dramatically reduced if the programmer has the ability to send data to multiple points in the system as well as the ability to monitor data at multiple points in the system. The debugger's ability to asses a system as a collection of smaller systems reduces time to market; without this ability the developer must waste time developing separate smaller systems that must be later reconstructed.

2.1 Timing

Correct timing is essential for any communication system. The throughput of larger and more complex communication systems is often limited by the slowest component in the system. In many situations, excessive latency in one signal processing area can lead to eventual, if not immediate, system failure. Because of the importance of timing, it is essential that the debugger has the ability to asses the timing performance of his/her system. By being able to immediately identify latency in the system the developer is able to know where in his/her code it is necessary to implement more efficient algorithms or buffers. Assessing timing is an inherently difficult problem to solve since any assessment of timing will have at least some impact on the timing of the system itself. In order to accurately analyze the timing of a system the runtime performance of the system should be impacted as little as possible.

2.2 Plotting

Plotting data is a convenient method for determining the approximate response of an algorithm. By plotting the output of a component or a waveform the developer is able to quickly identify severe problems in his system. On the component level, for example, the developer can see if his/her low-pass filter is indeed attenuating high frequencies. On a waveform level, a developer can plot the output of each component in a system in order to determine where in the waveform the signal may be becoming corrupt.

2.3 Signal Sources and Sinks

One of the most intuitive means for debugging is the use of signal sources and sinks. Signal sources are used to send known data to a system, and signal sinks allow the developer to compare the resulting output against expected data. There are multiple approaches to implementing sources and sinks. One approach is to include the sources and/or sinks in a waveform; this approach is advantageous when the sources and/or sinks are permanent (e.g., a waveform that always writes data to file). In the event that the signal sources and/or sink are temporary (i.e., they are only present for debugging), they must be added and removed from the waveform during the debugging process. Alternatively, it is possible to use a development environment that can add and remove signal sources and sinks during runtime. Using this approach, the developer does not have to reconstruct his/her waveform every time a different signal source/sink configuration is desired.

2.4 Use of Multiple Debugging Environments

The use of multiple debugging software packages can allow a developer to take advantage of the strengths of each package. For example, MATLAB contains many toolboxes for signal processing. Ideally, any software-debugging tool should have the ability to interact with existing software packages. This allows the debugger to take advantage of the most appropriate tools at appropriate times, and helps prevent the debugging package from becoming obsolete. Allowing software packages to interact with each other all keeps the tool developer from having to "reinvent the wheel."

One of the most common methods of facilitating interaction between software packages is reading and writing to file. By giving a debugging tool the ability to write data to a file, it becomes compatible with any other software package that is able to read that data file.

3. EXAMPLE DEBUGGING ENVIRONMENT: ALF

OSSIE is an SCA-based framework for developing, debugging, and running software defined radios. The software is open source, and free to the public [1]. ALF is an open-source graphical debugging environment for SCA-based waveforms running on the OSSIE framework. Because ALF is open-source and it utilizes an open-source framework (OSSIE), there is unlimited potential for tool add-ons as well as other modifications. This expandability allows the developer to utilize the tool for unique and proprietary systems. Simply put, if ALF or the ALF tools do not fit the needs of the developer, they can be changed by the developer.

Once a developer has created his/her component waveform code, he/she can use the OSSIE framework along with ALF to run and debug the waveform. Debugging can be performed using various tools (generally written in Python) to monitor data, send data to the system, as well as monitor latency in the waveform. The ALF tools are written in, but are not limited to, Python. In ALF's current version, available tools include Plot, Arbitrary Waveform Generator (AWG), and Write to File.

Through the use of CORBA, ALF is able to connect to any available *uses* or *provides* port. ALF connects to the framework in order to get a list of all installed applications, through which it can obtain a list of all components and ports in each application. When a user wishes to establish a connection between an ALF tool and a port, a pointer to the resource is obtained, which allows ALF to get a reference to the port using the getPort() method through CORBA. Once the reference to the port is obtained, ALF is able to narrow to the Port. In the case of a *provides* port, the handle to the port allows ALF to call port methods such as pushPacket(). In the case of a *uses* port, the connectPort() method is called. Once this method is called, all data sent to the *uses* port will in turn be sent to the appropriate ALF tool.

Prior to the development of ALF, in order to debug a single component, a waveform containing the component must be generated in order to run the component in the OSSIE framework. ALF utilizes the OSSIE Waveform Developer to automatically create a temporary waveform environment for any existing component, allowing the ALF tools to be utilized on stand-alone components (not just waveforms).

Since ALF is able to communicate with ports using the OSSIE file system, ALF can also be used in debugging applications running on multiple hardware platforms (e.g., an embedded platform).

Figure 1 displays the ALF main window. The upper left portion of the display contains a list of available waveforms on the file system. To install and start a waveform, the user simply double-clicks on the waveform. Once installed, the running waveform is added to the list of installed waveforms in the bottom left portion of the display. A block diagram display of the waveform of interest inhabits the right portion of the display.

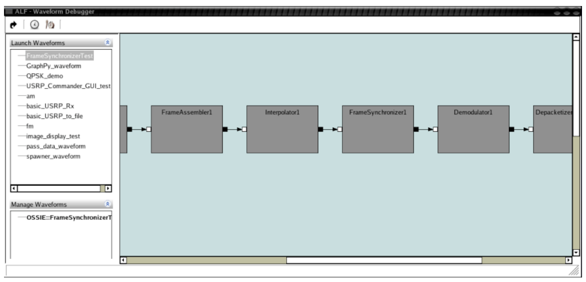

Figure 1: The ALF Main Display

3.1 ALF Timing

The ALF timing tool allows the developer to view the approximate throughput (in samples per second) of *provides* ports in a waveform. Using this feature allows the developer to quickly realize which components in a system have unsatisfactory throughput relative to other components in a waveform. In order to use the timing capabilities of ALF on a given *provides* port, there must exist a corresponding timing *uses* port in the component.

When timing is enabled in ALF, the timing ports in each component of a selected waveform are activated and a timestamp is made for every packet being sent to a *provides* port in a waveform. After a *provides* port is finished processing an incoming packet, a message is sent to the component's timing port. The timing port then creates a second timestamp that is retrieved by ALF. ALF then calculates the approximate throughput in samples per second by simply dividing the difference between the two timestamps by the known number of samples per packet. The programmer is then able to see the throughput of the port in real time in the ALF graphical user interface (GUI).

In the event that a developer wishes to approximate the throughput of a component without adding a timing port, it is also possible to approximate the throughput of a *uses* port by connecting the *uses* port to a component with a *provides* port with timing support that belongs to a different component. This connection can either be established in the software assembly descriptor file of the waveform prior to runtime or during runtime using ALF's connect tool (not detailed in this paper). In order to obtain a more accurate approximation of the throughput of the uses port using this method, the timing component should have as little overhead as possible.

It is possible to run SCA waveforms on multiple hardware platforms within a single domain; therefore, it is possible to monitor the throughput of provides ports running on embedded platforms using ALF. This ability is extremely useful since the throughput of a component running on, for example, and digital signal processor varies significantly from the throughput of the same component running on a general purpose processor.

3.2 Plot Tool

The plot tool allows a developer to plot any data coming from a component *uses* port in real time. The tool currently supports plotting frequency domain (utilizing an FFT) as well as plotting constellation diagrams.

Prior to the development of this tool, a developer could visualize his/her data by either adding plot components to the system prior to runtime or by writing data to file using whatever means and then plotting the recorded data using a second software package (e.g., Microsoft Excel). Using the plotting tool is advantageous over using plotting components since the system does not have to be modified prior to runtime. Adding multiple plot components is unwieldy, and often leads to runtime issues as the overhead of graphical displays is often excessive. As mentioned in Section 2.1, introducing latency can arrest the entire system. By having a plotting tool available, the waveform does not need to be reconstructed, and graphical processing only occurs when needed.

The plot tool is based on a commonly used open source package, but it has been modified to be compatible with the OSSIE framework. During initialization the Plot tool initializes its own ORB. Once initialized, the ORB allows the Plot tool to connect to the naming service used by OSSIE. Since ALF has already retrieved the naming service names associated with the port (application name, component name, and port name), the tool can narrow to the port and connect to the port. Once the port connection is established, all data sent to the uses port by the component will in turn be sent to the Plot tool.

As an example, the Plot tool is used to diagnose a problem in a 16 QAM system. It is known that the output signal is exhibiting extremely bad bit error rate performance. Without a visual representation of the signal flow of the system one can only speculate the cause of error.

In this example, the plot tool is used in order to observe the frequency domain content of the signal in order to determine if the poor BER performance may be a result of interference. The FFT of the output of the interpolator component (see Figure 2) exhibits a good signal to noise ratio without any apparent interference.

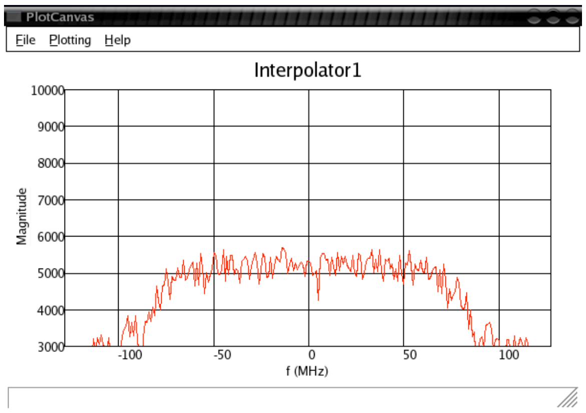

Figure 2: Frequency domain plot

Since the spectrum at the output of the interpolator is consistent with expectation, the output response of the next component in the system, the frame synchronizer, is analyzed. In this situation, the spectrum of the output signal does not provide any insight, but the constellation diagram of the signal does. As seen in Figure 3, the signal constellation is exhibiting a phase rotation. Once this phase rotation is detected, the developer is able to narrow his/her search to areas in the code within the frame synchronizer that could potentially cause a phase rotation.

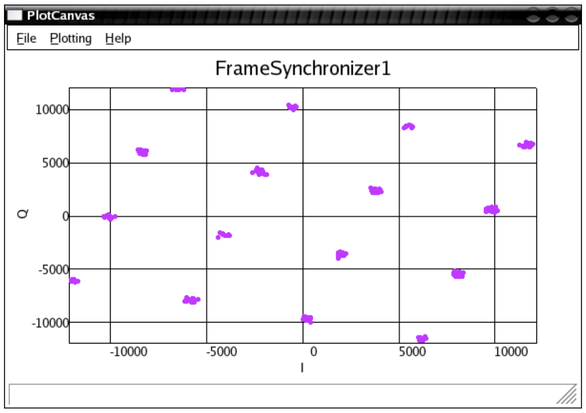

Figure 3: Constellation plot with phase rotation.

An appropriate correction is made, and the resulting constellation plot can be seen in Figure 4. With the phase rotation removed, the system now exhibits a satisfactory BER.

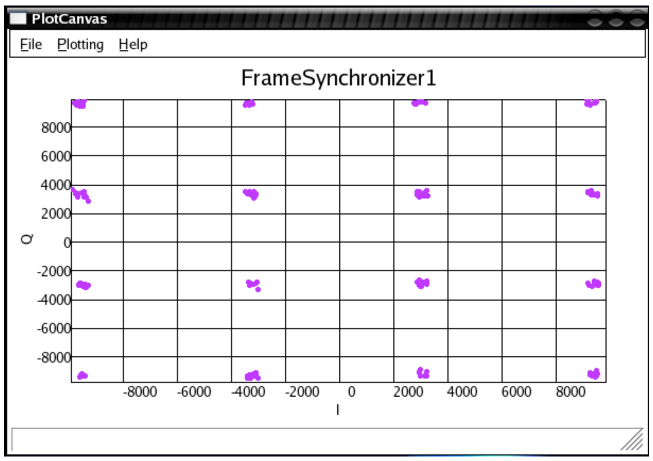

Figure 4: Constellation plot without phase rotation.

3.3 Write to File Tool

The “write to file” tool allows the developer to record packets that are being sent from a component *uses* port. This tool gives the developer the ability to observe packets being sent anywhere throughout the waveform without having to neither add any code to the component nor add any “write to file” type components to the waveform. In the case of a data radio, the “write to file” tool can serve as the system’s application layer.

This use of data logging gives the developer the ability to post process his/her data using any tool with the ability to read data from a file. For example, a developer is able to analyze the effects of multiple types of filters on his/her data by post-processing in MATLAB or Octave.

Due to the Pythonic nature of the tool, formatting data becomes trivial; for example, the developer can use Python's built in XML parsers to write the data in XML format. By having the flexibility to write data to whatever format the developer desires, exchanging data with other software packages does not require manual editing or a third software package.

The method that the Write to File tool uses to obtain data through the use of CORBA is identical to that of the Plot tool (detailed in section 3.2). Built-in Python methods are used to write the data to file at the push of a button.

3.4 The Arbitrary Waveform Generator Tool

Within the available ALF tool library there exists an arbitrary waveform generator (AWG) tool. The tool allows the developer to connect to an existing *provides* (input) port in a running waveform. Once the connection is established, the tool is able to send any arbitrary signal (as defined by the developer) to the component. As long as the component is capable of processing enough data in real time (either through the use of efficient algorithms or buffering), the AWG can send data to the component simultaneously with data being sent through a previously existing connection.

Multiple options exist for the type of signal the developer wishes to generate (or read from file) and "push". In the file "sources.py" there exists a signal class "sources". This class contains the method "get_sources_list" as well as a method for each signal type the user wishes to define. The default tool currently has 6 available signal types [1]. The available signal types are defined in the "sources" __init__ attribute:

```
def __init__(self,parent):
    self.parent = parent
    self.available_sources={
                'file': 'read_file()',
                'sine': 'gen_sine()',
                'cosine': 'gen_cosine()',
                'random': 'gen_random_data()',
                'zeros': 'gen_zeros()',
                'ones': 'gen_ones()'}
```

The "get_sources_list" method simply returns a list of the available sources.

```
def get_sources_list(self):
    return self.available_sources
```

This list of sources serves two purposes in the AWG.py module. First, the list is used in the initialization of the AWG GUI so that each source in sources.py can be selected from in the GUI menu. When the AWG tool is running, the developer is able to select the desired source, and the index of the selected source then dictates which signal method is called.

By having a Pythonic list of signals available, a developer is able to easily add his/her own signal generation code without having to edit any of the code in AWG.py; therefore, knowledge of wxPython is not necessary for adding signal sources to the wxPython tool. However, due to the open-source nature of AWG.py a more experienced wxPython user is still able to add his/her own graphical switches/buttons/inputs to the GUI as he/she feels necessary. The SCA nature of the tool as well as the connections made also remains transparent to the developer, but is still editable if necessary for more in-depth debugging.

The "gen_sine" method is analyzed as an example of the source methods available.

```
def gen_sine(self):

    #initializations
    count = 0
    sine = []

    #recursively generate the sinusoid
    while count < self.parent.len:
        sine.append(math.sin(
            self.parent.freq * 2 *
            math.pi * count / self.parent.len))
        count = count + 1
    return sine
```

A variable "count" is initialized for the *while* loop. The variable "sine" is declared as a Python list so that the "append" method can be utilized. A Python list has been selected as the standard return type for the source methods. The list returned (in this case "sine") will be sent directly to the component *provides* port. The type of data within the list (e.g., short, float, or char) can either be set in the signal source method or type-casted in the AWG.py file (in this case, the latter option is used). The data type within the list must be consistent with the data type of the *provides* port being connected to (otherwise CORBA will throw an error).

In this example, the signal is recursively generated using the Python "math" module. This approach for generating the list can be replaced with more efficient/appropriate approaches as desired by the developer. The variables parent.freq (sinusoidal frequency) and parent.len (desired length of the list) are inherited from the GUI class. The use of inheritance allows the developer to add switches/buttons/input to the GUI which set any desired variable(s). For example, the developer can add an option for phase noise variance in the GUI, which can then be set as a class field that can be inherited by the "sources" class. In the case that the developer wishes to generate signals that are not easily or efficiently constructed by calling methods from the Python math module, custom modules can be written in other programming languages (commonly C).

Using the math library, or any other available library, a developer can easily customize this method or generate new method based on this method (e.g., Gaussian phase noise could be added to the sinusoid if appropriate).

The "read_file" method exists for developers who wish to generate their signals using other methods for whatever reason. For example, a developer may desire to generate a signal in MATLAB or use data that has been taken from over the air.

```
def read_file(self, parent):
  try:
    # attempt to open the file:
    my_file = open(parent.file_name, 'r')

  except IOError:
    print "error opening, or no file
          named " + parent.file_name
    return []  #return an empty packet

  # read the file as a single string
  data_from_file = my_file.read()
  my_file.close()

  # reformat the string by
  # removing any unwanted characters
  data_from_file.strip('\n')
  data_from_file.strip('[')
  data_from_file.strip(']')

  data_from_file =
    #break string into a list
    data_from_file.split(parent.delimiter)
  return data_from_file
```

Currently, the developer is able to specify two file names in the GUI: one for the I channel and one for the Q channel. The data delimiter can also be specified (though in the current version the delimiter is hard coded to be a comma). The "read_file" method attempts to open the file, and if successful it reads data in the file as a string. The data can then be reformatted as desired (in this case the square-brackets generated by MATLAB are removed). The tool's ability to reformat data facilitates interoperability with other software packages that may parse data in different formats. Once the data is reformatted, the Python method "split" (available in the string library) breaks the data string into a Python list that can then be returned to the parent.

The AWG frame contains a button labeled "Push Packet". Once the desired options are set in the AWG GUI and the Push Packet button is pressed, the I and Q signals are generated by calling the appropriate method in the sources class. The method "pushPacket" is called on the port handle created when the connection was established:

```
self.PortHandle.pushPacket(self.I,self.Q)
```

The data is now in the hands of the component that the tool connected to. The pushPacket method can be called as many times as the user desires. Due to the Pythonic nature of the tool, it is trivial for the developer to add a loop that recursively sends signals at the single push of a button.

4. CONCLUSION

In order to debug a system under development, as well as validate the system's correct functionality, many aspects of the system must often be analyzed: timing, output responses, standard compliance, etc. To effectively perform this analysis it helps to have a tool that is capable of performing several operations while allowing for expandability for unforeseen debugging requirements. By providing interoperability between existing tools, the developer is able to take advantage of the strengths of each tool he/she has available. The open source nature of ALF allows for a variety of debugging abilities through tools, such as timing, plotting, reading from file, and writing to file. The Pythonic nature of the tools allows a developer to adapt the tools to suit his/her needs.

[1] "Welcome to the OSSIE development site for software-defined radio," Accessed September 14, 2007 at http://ossie.wireless.vt.edu/trac

This material is based in part upon work supported by SAIC.
This material is based in part upon work supported by the National Science Foundation under Grant No. 0520418. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.