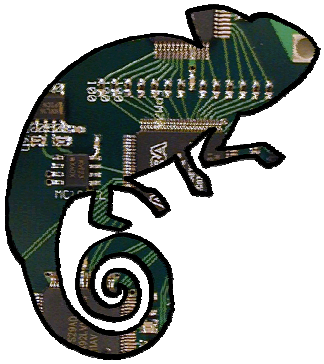

Chameleonic Radio

Technical Memo No. 14

Report for the NIJ SCA Based Radio

P. Balister, T. Tsou, and J.H. Reed

September 30, 2006

Bradley Dept. of Electrical & Computer Engineering
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061

Report for the NIJ SCA Based Radio

Philip Balister, Tom Tsou and Jeffrey H. Reed
MPRG/Wireless@Virginia Tech
balister@vt.edu ttsou@vt.edu reedjh@vt.edu

September 30, 2006

1 Introduction

This report describes efforts to develop a Software Communication Architecture (SCA) baseband processor for a multi-band, multi-channel radio for public safety. This report describes the baseband processor software for FM transmission and reception, the hardware used to test the software, installation instructions for people who wish to study this work, and some some discussion of issues discovered over the course of the project.

2 Radio Description

2.1 SCA

2.1.1 The SCA Standard

The Software Component Architecture (SCA) defines how elements of hardware and software operate and interact within a waveform application. In doing so, the SCA provides developers with specific guidelines in designing software and interfacing with associated hardware. The SCA, however, does not provide information regarding the actual implementation of software radio devices. Developers create software implementations in a manner similar to an architect designing a structure in accordance with local building codes. Compliance with the SCA ensures that software components and devices will meet necesary interoperability standards and interact with other compliant components.

2.1.2 OSSIE

OSSIE stands for Open Source SCA Implementation::Embedded and is a C++ implementation of the SCA standard developed at Virginia Tech. OSSIE is an open source effort designed primarily to support academic reseach and education into software defined radio (SDR) and wireless communications. OSSIE provides a Core Framework (CF) that serves as the central functionality for loading waveform applications and deploying and connecting individual software signal processing components. Furthermore, OSSIE provides a set of standard interfaces for use by developers to support portability and interoperability in component designs. Additional information pertaining to OSSIE functionality and project applications is available from the OSSIE website [1].

To support its functionality, OSSIE leverages the capabilities of two other open source projects. The first, the Xerces C++ parser, adds XML manipulation to the framework. This capability is required in order to read and process the SCA XML descriptor files that describe the waveforms and underlying components. The second open source program used by OSSIE is a CORBA implementation. As a requirement of the SCA, OSSIE utilizes the Common Object Request Broker Architecture (CORBA) for intercomponent communication. To meet this requirement, OSSIE uses omniORB, a high performance CORBA implementation initially designed for use on small embedded devices that has since spread to a wide variety of platforms and applications.

2.2 Platform

The FM Transceiver waveform operates on a SCA platform shown in Figure 1 consisting of four interface and proxy devices. These devices provide abstractions for the underlying hardware on the platform. Specifically for our implementation on standard PC's, the devices provide abstractions for the central processor unit (CPU), sound card or audio hardware, and the Universal Software Radio Peripheral (USRP). The four SCA devices are called the GPP, USRP, SoundIn, and SoundOut. The General Purpose Processor (GPP) device is an *ExecutableDevice* upon which individual software components are deployed. The USRP provides RF input and output to the waveform. Finally, the SoundIn and SoundOut components are interface devices that bring audio capture and playback capabilities to the waveform.

2.2.1 GPP

GPP is a proxy component for a general purpose processor. The device provides a device abstraction upon which multiple software components can operate. The motivation for using a processor abstraction such as the GPP is waveform portability. A certain component may

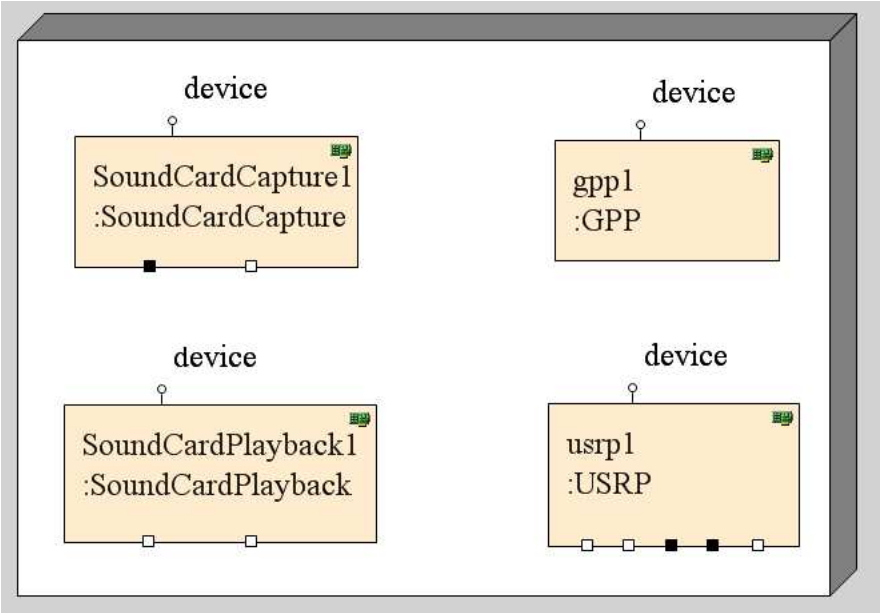

Figure 1: Platform Devices

have multiple implementations for operating on different types of hardware. For example, a component may have one implementation for a GPP and another implementation for a digital signal processor (DSP). The processor abstractions provide standard methods of deployment onto different types of devices of various possible hardware platforms.

On the FM Transceiver waveform, all software components currently operate on the GPP proxy device. Therefore, all software components are co-located on a single processor. In the future, however, other devices will be available. A digital signal processor (DSP) proxy device will serve a similar purpose for deploying components to run DSP chips.

2.2.2 USRP

The Universal Software Radio Peripheral (USRP) is a low-cost, high speed hardware board for implementing software radio applications. Design of the board is by a team led by Matt Ettus of Ettus Research LLC. The device primarily consists of an USB2 interface for connecting to a PC or other device, high speed A/D and D/A converters, an Altera "Cyclone" FPGA, and daughterboard slots for attaching a variety of RF front ends capable of operating at different frequency ranges. The USRP provides a simple way for developers to utilize RF hardware for transmitting and receiving radio signals. The flexibility of swappable daughterboards allows for development of a large variety of communication applications across a wide range of frequencies. Furthermore, the USB interface makes the USRP portable and well suited for mobile use.

For interfacing with the software drivers and physical hardware of the USRP, we employ a SCA device proxy component. The USRP device proxy provides control and data interfaces to the USRP hardware. Control information includes data such decimation and interpolation rates for the receiver and transmitter respectively as well as NCO frequency tuning. This SCA device also provides the transport mechanism for moving data between the USB interface and the CORBA interfaces in the waveform.

2.2.3 Sound In

The SoundIn provides an audio input abstractions for the waveform. Although audio hardware may differ across Linux systems, the SoundIn device provides a standard means of obtaining audio from a microphone and sound card and making that data available to the waveform components. In order to provide this abstraction, the SoundIn device uses the Advanced Linux Sound Architecture (ALSA) to receive a stereo audio stream from the soundcard. ALSA is an open source set of audio drivers and library package that primarily provides efficient support for different types of audio hardware and a user library for simple application programming and functionality.

The device component uses the ALSA library to access sound card drivers and retrieve a data stream from a microphone or line input. The interleaved stereo signal is split into two channels and output through the complexShort interface with L and R replacing the real and imaginary portions respectively.

2.2.4 Sound Out

The SoundOut component operates the soundcard in capture modes and is otherwise similar in structure to the SoundIn component. SoundOut uses ALSA for interfacing with the PC audio hardware and providing output to connected speakers or headphones. The device also differs, however, in that the incoming audio stream is buffered to smooth out the possible irregularity of the incoming packet data stream. This buffering is required because the incoming data stream is subject to processing and scheduling delays that may occur as data packets move through waveform components and the CORBA connections between them.

2.3 Waveform

The FM Transceiver waveform design is shown in Figure 2. The waveform consists of seven *Resource* components. Five components are used for signal processing, while two components perform control operations. The Interpolator has two separate instantiations of the same functional component. This setup provides a two stage process for interpolation. Another

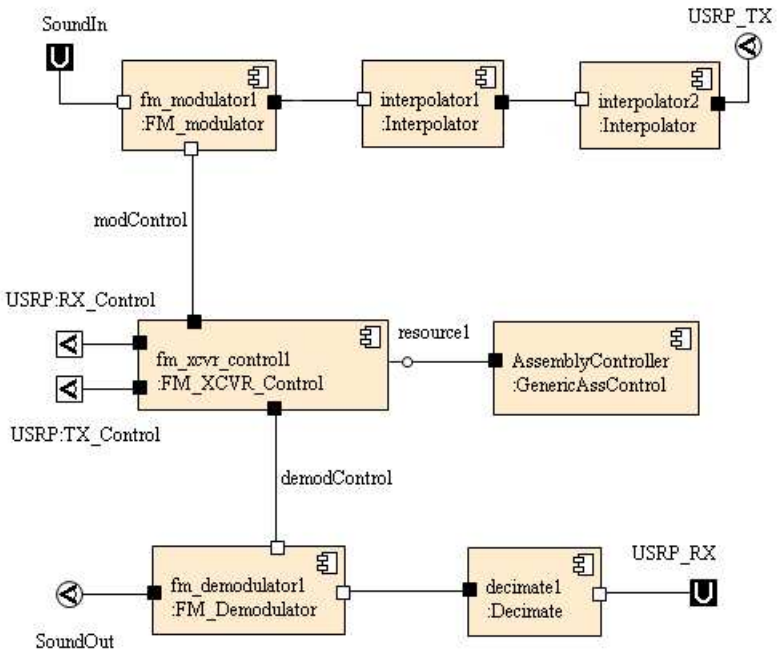

Figure 2: FM Transceiver Waveform

implementation, however, could easily be created with a single interpolator component for the same effect.

2.3.1 Decimator

The Decimator component reduces the sample rate of a signal to a lower sample rate suitable for a desired task. Specifically in the FM Transceiver waveform, the decimator component reduces the high USRP output rate of 250ksps to a rate suitable for audio processing and output. A decimation factor of 10 reduces the sample rate to 25ksps which is easily handled by the SoundIn device and underlying audio hardware.

The internal operation of the Decimator consists of a downsampler and lowpass filter. Reducing the sample rate, or downsampling, simply involves throwing away samples. For example, with a decimation factor 2, the downsampler removes every other sample. Typically, decimation also involves lowpass filtering the data to match the Nyquist criteria of the new output sample rate. In the FM Transceiver waveform, however, the incoming signal is already heavily oversampled; thus, only downsampling is necessary because the higher frequency content is limited. Consequently, there is no lowpass filtering required.

In other waveforms where the input signal is not oversampled, however, implementing the lowpass filter may be required. The decimator accomadates filtering tasks by reading

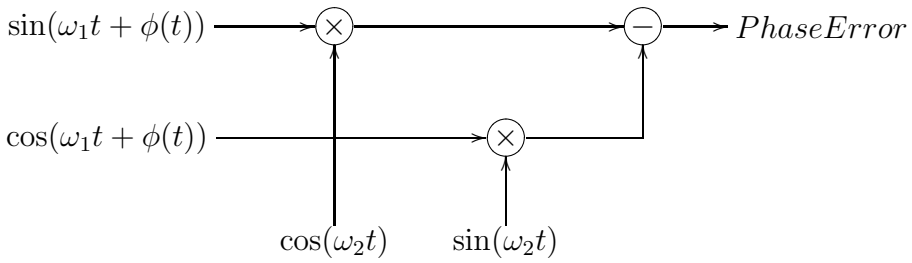

Figure 3: Phase Detector

filter coefficients from an XML property file. The developer can generate these coefficients from Matlab or any other filter design methods, and insert them into the property file. The Decimator configures itself with these values and uses them to filter the data stream.

2.3.2 FM Demodulator

The heart of the FM Demodulator is the phase detector that operates on the complex baseband signal from the USRP. Figure 3 shows a block diagram of the phase detector. The output of the phase detector is:

$$
\begin{aligned}
PD_{out} &= \sin(\omega_1 t + \phi(t)) \cos \omega_2 t - \cos(\omega_1 t \phi(t)) \sin \omega_2 t && (1)\\
&= \cos \phi(t)(\sin \omega_1 t \cos \omega_2 t - \cos \omega_1 t \sin \omega_2 t) + \sin \phi(t)(\cos \omega_1 t \cos \omega_2 t + \sin \omega_1 t \sin \omega_2 t) && (2)\\
&= \cos \phi(t) \sin(\omega_1 t - \omega_2 t) + \sin \phi(t) \cos(\omega_1 t - \omega_2 t). && (3)
\end{aligned}
$$

In equation 3, the term $\sin(\omega_1 t - \omega_2 t)$ becomes small when ω_1 is close to ω_2, in addition the $\cos(\omega_1 t - \omega_2 t)$ term is close to one under this condition. Since the case where ω_1 is close to ω_2 is the lock condition for a PLL, the output of the phase detector is $\sin \phi(t)$, where ϕ is the phase difference between the received signal and the loop NCO. Since $\sin \phi(t) = \phi(t)$ for small $\phi(t)$ linear system analysis maybe used to calculate loop stability [2].

2.3.3 Interpolator

The Interpolator upsamples and filters the data to preserve only the original sample data. Figure 4 shows the diagram for an interpolator. The waveform utilizes two identical interpolator components with an interpolation factor of 4. This yields a two-stage interpolation factor of 16 for the waveform and a final sample rate of 250ksps from approximately 15600sps out of the soundcard.

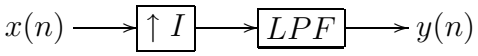

Figure 4: Interpolator Diagram

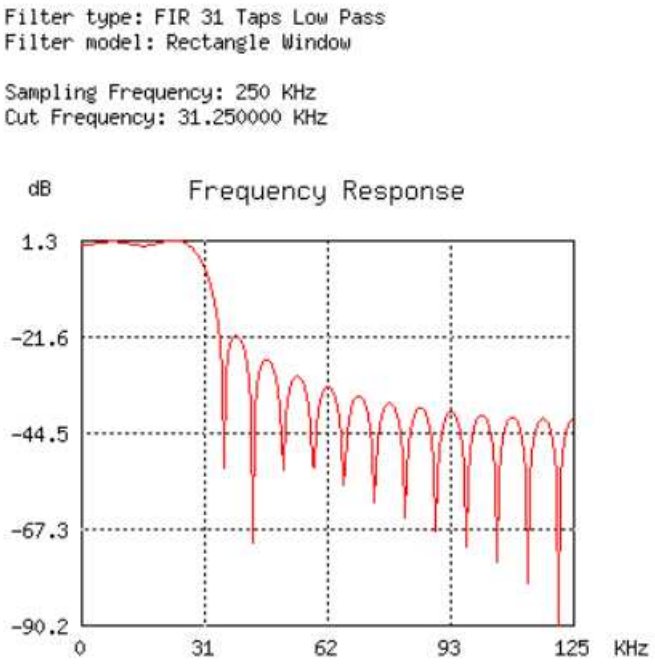

Figure 5: Filter

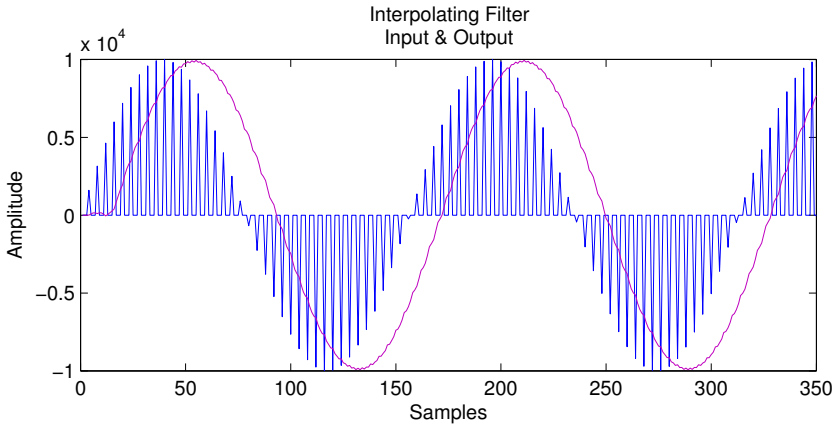

Figure 6: Interpolation

After upsampling, the the signal must be filtered to remove unwanted images created as result of the zero-padding process. The output of the ideal filter has the original spectral content below the original Nyquist frequency and zero spectral content above. Thus, for a interpolator with a factor of 4, the zero padding is followed by quarter band filter with $f_s/8$ cutoff with f_s being the sample rate after upsampling. The filter designed for the waveform interpolator consists of a 31 tap rectangular window FIR filter. Figure 5 shows the frequency response of the filter We used WinFilter [3], a freely available digital filter design tool, to generate the filter coefficients. The effect of the filter on the output samples is shown in Figure 6.

2.3.4 FM Modulator

The FM Modulator operates on a real valued input and outputs complex baseband (I and Q) signals. With two channel audio as the input, the modulator operates on the Left signal only; the Right signal is ignored. A simple analog FM implementation operates with a modulating signal fed as the tuning voltage to a voltage controlled oscillator (VCO). The output frequency of the VCO is variable to an amount proportional to the magnitude of the input voltage. Thus, the VCO outpus a frequency modulated signal. Analogous to the anolog version, the digital FM implementation uses a numerically controlled oscilator (NCO) for FM signal generation. The NCO consists primarily of a phase accumulator and sine operation and is shown in Figure 7.

FM modulation and the NCO design in Figure 7 can be described by the following equations in continuous phasor,

$$Ae^{2\pi jk\int m(t)dt} \tag{4}$$

Figure 7: Numerically Controlled Oscillator

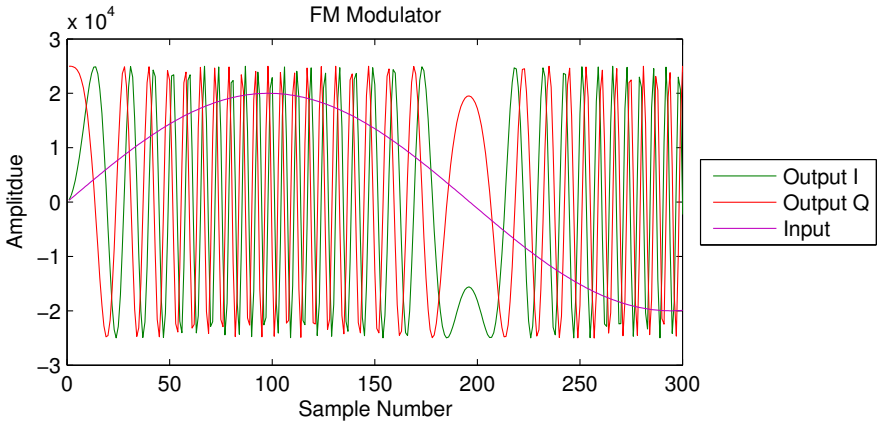

Figure 8: FM Modulator

complex,

$$A\cos(2\pi k\int m(t)dt) + jA\sin(2\pi k\int m(t)dt) \tag{5}$$

and discrete forms.

$$R(n) = \cos(2\pi\sum_{k=1}^{n} km(n)) \tag{6}$$

$$I(n) = \sin(2\pi\sum_{k=1}^{n} km(n)) \tag{7}$$

In complex baseband FM generation, the IF of the FM signal is set to 0 Hz. Thus, at any given moment, the instantaneous output frequency shifts between positive and negative values, but never both at the same time. Consequently, the time domain signal is complex and a phase shift occurs in the output when the amplitude of the input signal crosses zero. Figure 8 shows this behavior and the resulting complex time domain output along with the modulating input signal.

The frequency deviation, Δf, represents the maximum frequency difference between the instantaneous frequency and carrier. With the carrier at baseband, Δf is the frequency output by the NCO at the maximum amplitude of the modulating input signal. We can control the amplitude and consequently the deviation with automatic gain control (AGC).

Controlling the deviation allows the waveform to meet the varying operating requirements for different frequency ranges.

$$\Delta f = A_{max} k f_s \tag{8}$$

2.4 Radio Control

From a SCA architecture standpoint, the control components are identical in structure to the signal processing *Resource* components. Instead of streaming communication data, however, the control components pass information that control operations such as starting or stopping the waveform, and configuring parameters on components. The FM Transceiver waveform uses two dedicated control components. *AssemblyController* controls basic start and stop operations; the FM_XCVR_Control component controls FM specific operations and controls operations pertaining to the USRP, modulator, and demodulator components.

2.4.1 Assembly Controller

In a SCA waveform, the *AssemblyController* is that first component that is invoked when a start or stop command is issued by the radio operator. In the FM_XCVR waveform, the AssemblyController issues further start and stop signals to the signal source, sound card, and the FM_XCVR_Control, which provides more sophisticated control operations for the USRP, modulator, and demodulator components. This level of basic control is required to ensure that components operate in a predictable and controlable manner. For example, upon waveform startup, a source component such as the SoundIn device, cannot send packets until the other components in the signal chain are configured and ready receive the data. Thus, start and stop capabilites for that component are delegated to the *AssemblyController*.

2.4.2 FM_XCVR Control

The FM_XCVR_Control component, provides control operations specific to the FM waveform. It sends relevant data to the USRP such as external port information, decimation and interpolation rates, and transmit and receive frequencies. Before the USRP can transmit or receive data, it must be properly configured. The FM_XCVR_Control component issues commands to perform these operations and ensures that the waveform is sending and receiving data on the desired ports of the USRP. Furthermore, the FM_XCVR_Control component issues these commands when the user requires that these parameters should change. For example, modifying the transmit frequency of the waveform to operate on a different FRS channel would be a task performed by the FM_XCVR_Control component.

3 Radio Installation

3.1 Required Hardware

The FM Transceiver waveform requries a USRP and a PC running the Linux operating system.

3.2 Required Software

3.2.1 Linux OS

OSSIE and the waveform run on the Linux operating system. OSSIE does not require a specific Linux distribution, however, we recommend the use of Fedora Core 4 or Core 5 because those distributions are used for the bulk of our development and testing.

3.2.2 Xerces C++

Xerces is an XML parser produced by the Apache project. Xerces is used by the OSSIE framework for parsing SCA descriptor files. Xerces C++ is available for free download from their website. Download the currentt source release. Instructions are included with the download, however, the following commands will suffice.

http://xml.apache.org/xerces-c/download.gci

3.2.3 omniORB and omniORBpy

omniORB is a high performace implementation of the CORBA ORB written in C++ and Python. omniORB and omniORBpy are available for download from their website. Download the most recent stable release which is omniORB 4.0.7 and omniORBpy 2.7 at the time of writing. Follow the included installation instructions.

http://omniorb.sourceforge.net/download.html

3.2.4 Amara

Amara is an XML toolkit that provides Python tools for XML parsing, creation, and editing. Download and installation information is available from the website.

http://uche.ogbuji.net/tech/4suite/amara/

3.2.5 Subversion

Subversion is the open-source version control software used for the OSSIE project. Download and installation information is availble from the website. Subversion is also distributed with most recent Linux distributions.

http://subversion.tigris.org/

3.3 ALSA Configuration

3.3.1 ALSA

The sound components rely on the Advanced Linux Sound Architecture (ALSA) for providing audio functionality. ALSA provides Linux support for multiple audio interfaces and sound cards as well as a common library for simplified application programming. In the current 2.6 versions of the Linux kernel, ALSA is the default audio support architecture.

3.3.2 Dmix

Dmix is a PCM plugin for ALSA that extends features and functionality of supported devices. Dmix allows low-level sample conversions between channels and soundcard devices as well as mixing of multiple audio streams. The waveform utilizes Dmix to provide coexistence of multiple audio playback and capture components with a single sound card.

3.3.3 Configuration

Depending on the audio hardware and version of ALSA installed on the system, a configuration file may or may not be required. ALSA and Dmix are configured through the .asoundrc file in the user home directory or the /etc/asound.conf file for global settings. ALSA versions prior to 1.0.11 generally require the configuration file to allow multiple applications access to the sound card at the same time. More recent versions of ALSA, however, should not require the file. In either case, following basic configuration file will work for most installations.

```
#~/.asoundrc

pcm.dsp0 {
    type plug
    slave.pcm "dmix"
}
```

```
ctl.mixer0 {
    type hw
    card 0
}

pcm.!default {
    type plug
    slave.pcm "dmix"
}
```

3.4 Building OSSIE

The primary means for obtaining the OSSIE Core Framework and associated component and waveform packages is through the Subversion repository. Downloading from the repository requires an account and acknowledgement of the licensing terms (GPL/LGPL). Downloading from the repository is performed with the `svn` command followed by the repository name and package path.

```
svn co https://ossie-dev.mprg.org/repos/ossie/PACKAGE_PATH
```

The .tar.gz file extension signifies a compressed archive file. After downloading the source from the repository, the basic process for building OSSIE packages is the same for the framework, devices, resource components, and waveforms; The steps that follow will unpack the archive file, build the source, and install.

```
tar xvfz PACKAGE_NAME.tar.gz
cd PACKAGE_NAME
./reconf
./configure
make
su -c "make install"
```

3.4.1 OSSIE Core Framework

The OSSIE Core Framework (CF) is our implementation of the SCA framework standard. The CF is found in the following respository location.

```
/NIJ-sca/ossie.tar.gz
```

Thus, to build the framework using the procedure listed above, use the following steps.

```
svn co https://ossie-dev.mprg.org/repos/ossie/NIJ-sca/ossie.tar.gz

tar xvfz ossie.tar.gz
cd ossie
./reconf
./configure
make
su -c "make install"
```

3.4.2 Standard Interfaces

OSSIE components use a standard set of idl interfaces for communicating between components. Standard interfaces provides basic support for transfering data in typical forms such as complex, real, floating point, and integer. Basic radio control interfaces are also provided. Standard Interfaces are built and installed using the same process as the CF.

```
/NIJ-sca/standardInterfaces/standardInterfaces.tar.gz
```

3.4.3 Nodebooter

The Nodebooter, along with the wavLoader, are the programs required to load and execute a waveform application. Nodebooter prepares the platform environment such that waveforms can be installed on to the appropriate devices. It is built and installed using the same process as the CF.

```
/NIJ-sca/platform/nodebooter/nodebooter.tar.gz
```

3.4.4 Wavloader

wavLoader is a Python sript that enables loading, unloading, start, and stop operations for the waveform. Prior to use, the script requires creation and configuring Core Framework Python bindings. These bindings are files that allow the Python language based wavLoader to communicate through CORBA with the C++ based OSSIE framework. They are created with the following commands.

```
mkdir idl_py
cd idl_py
cd <OSSIE_IDL_PATH>/*.idl .
omniidl -p<PATH_TO_OMNIORBPY> -bpython *
```

OSSIE_IDL_PATH is typically */usr/local/include/ossie/*. PATH_TO_OMNIORBPY is typically */usr/local/lib/python¡version¿/site-packages/*. Next, create a file `ossie.pth` in the directory*/usr/lib/python¡version¿/site-packages/* with the following contents. This links the newly created Python files to the primary Linux Python package directory.

```
<PATH_TO_IDL_PY>
/usr/local/lib/python<version>/site-packages/
```

Finally, download wavLoader from the repository.

```
/NIJ-sca/platform/wavLoader
```

And copy the script into the desired waveform directory.

```
$cp wavLoader/wavLoader.py /home/sca/waveforms/<WAVEFORM_NAME>
```

3.5 Building the Waveform

The FM Transceiver waveform consists of individual Device and Resource components and the waveform descriptor files. As with the Core Framework, the build process is identical. First, check out the source files from subversion.

```
svn co https://ossie-dev.mprg.org/repos/ossie/PACKAGE_PATH
```

Build the source with standard build procedure.

```
tar xvfz PACKAGE_NAME.tar.gz
cd PACKAGE_NAME
./reconf
./configure
make
su -c "make install"
```

3.5.1 Devices

```
/NIJ-sca/platform/GPP.tar.gz
                 /SoundIn.tar.gz
                 /SoundOut.tar.gz
                 /USRP.tar.gz
```

3.5.2 Resource Components

```
/NIJ-sca/components/Decimator.tar.gz
                   /FM_Demodulator.tar.gz
                   /FM_Modulator.tar.gz
                   /GenericAssControl.tar.gz
                   /Interpolator.tar.gz
```

3.5.3 Waveform

```
/NIJ-sca/waveforms/FM_XCVR.tar.gz
                  /FM_XCVR_Control.tar.gz
```

4 Results

4.1 Operation in the FRS band

4.1.1 Family Radio Service

The Family Radio Service (FRS) is a no-license frequency band in the United States that operates in the range of 462.5625MHz to 467.7125MHz. The FRS is used for serving general use walkie talkie communications within familiy, friends and associates. FRS devices utilize FM modulation, are restricted to fixed antennas and 500 milliwatts by the FCC. FRS has 14 channels with channels 1 through 7 being shared with the General Mobile Radio Service (GMRS). The GMRS is a licensed band with more generous operational restrictions. Consequently, FRS operation with greater than 500 miliwatts of output is possible in the overlapping channels with a GMRS license and within GMRS license limits.

Channel	*Frequency* (MHz)
1	462.5625
2	462.5875
3	462.6125
4	462.6375
5	462.6625
6	462.6875
7	462.7125
8	467.5625
9	467.5875
10	467.6125
11	467.6375
12	467.6625
13	467.6875
14	467.7125

4.1.2 FRS testing

Operating in the FRS band, the FM Transceiver waveform is able to transmit and receive audio with other FRS devices as well as other similarly USRP equipped Linux machines running the same waveform. We tested communications using a Motorola TalkAbout 250. The inexpensive and readily available handheld FRS radio device can operate on all 14 FRS frequencies. Figure 9 shows the Motorola FRS radio device. Our test scenario consists of a computer equipped with a USRP device and RFX400 (400-500 MHz transceiver) board, and two Motorola FRS radios. The setup is shown in Figure 10.

The USRP serves as the local test device while the TalkAbout radios serve as remote devices operating on different FRS channels. The USRP functions with reconfigurable transmit and receive capabilities between both remote devices. Thus, with the waveform running, the local user is able to switch between remote devices and speak or listen to audio from either of the configured endpoints. Currently, push-to-talk capabilities are partially implemented. Changing between transmit and receive modes, as well as different frequencies, requires configuration that results in significant delay. This delay limits real-time push-to-talk capabilities, however, furher development of the components as well as control interfaces will allow for much faster and more efficient operation.

Figure 9: Motorola FRS Radio

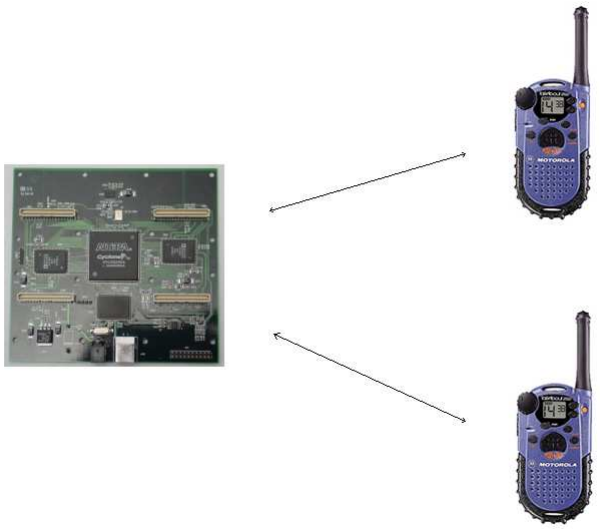

Figure 10: Waveform test setup with USRP and FRS radios

4.2 Component Development Simplification

One key development made for this project was the creation of a library to simplify creation of SCA ports in components. A future technical report will contain a detailed write up of this work since currently it has only been tested for a small number of port types and some further design refinement may be required after the structure becomes more widespread.

A SCA port is a logical construct that provides data communication paths between components. The software that implements a port uses code from two packages; CORBA provides inter-component data transfer, the SCA provides port management functions. Both of these packages require a great deal of knowledge to use them properly and effectively.

Prior to this work, a component writer chooses an interface from the standard interfaces package, a supplemental package for OSSIE, that defined several standard data transmission and control interfaces. The interfaces are defined in CORBA Interface Definition Language (IDL). To generate a port implementation, the component had to complete several sections of software. First, the component writer needed to compile the IDL into C++ header files, derive new classes for the port from the appropriate classes in the header files based on the specific port type. Once the basic structure was in place, a component threading model must be created to meet the data transmission requirements of the component. Once these tasks are complete to component writer could create the actual signal processing code.

The problem with this method is that, even with skeleton code created by the OSSIE Waveform Developer (OWD), experience showed that people new to the OSSIE project required a great deal of support from the more knowledgeable OSSIE developers.

The solution to this problem developed as part of this project is to create a new C++ class that encapsulated the CORBA code required to create the port and service requests made on the port. This C++ class requires only knowledge of CORBA sequences to use. The CORBA sequence is a variable length data type that provides bounds checking on random access operations very similar to a standard C++ vector class. For a provides port, the class defines two basic operations, a get buffer operation that returns pointers to buffers containing data and an operation that informs the class when the buffer is no longer required. For the uses port, there is one operation that takes care of sending the data to ports that have been connected to the port by the framework.

This structure greatly simplifies the the interface the component writer needs to understand in order to interface with the SCA ports defined by the component. This has improved component writer productivty and lowered the support demands on more knowledgable personell.

4.3 Impact of CORBA on Radio Performance

One question regarding use of the SCA for portable radios is the SCA's dependence on CORBA for inter-component communication. CORBA provides a vendor independent, platform neutral that allows applications to communicate with each other [4]. These applications do not need to reside in the some processing unit, they may be distributed across processors connected by some form of communication network. Typically, this is some form of network, but may be a special hardware inter-connects for platforms such as DSP's. There is an extension to CORBA under development by the Object Management Group [5] to provide a structure for specialized transports for use by CORBA. This is called the Extensible Transport Framework (ETF).

Since CORBA contains capabilities beyond what is required for providing inter-component communication for software defined radios, there was some concern that CORBA would require excessive system resources such as processor cycles and memory for the radio built for the project. This section presents information about the impact of CORBA on the final waveform. This report does not look at the latency or data transfer rate limitation, rather it looks at the processor overhead added internally to a component by using CORBA for inter-component communication. A future report will investigate memory usage.

The software performance is measured with a tool called oprofile [6]. Oprofile is a statistical, kernel based profiler that runs on Linux. A profiler records execution data from the running system that includes instruction pointer and stack pointer information. From the this information a post processor produces reports showing how frequently the program executes specific functions and lines of code. By examining call addresses on the stack, it is possible to divide subroutines time amongst the callers of that function.

A FM receiver was built from the components described earlier in this report. The RF squelch was adjusted so that signal always arrived at the demodulator. After loading and starting the waveform, the profiler was started. This prevented the profiler from counting waveform start up events. After running the waveform for over thirty minutes, the profiler was stopped and reports generated.

Three reports were produced for each executable that was profiled. The first report showed overall cpu usage by function, the second showed the call graph and how CPU usage was attributed to different parent functions, and the final report was disassembled source with execution information. Three components were profiled, the USRP interface device, the decimator and the FM demodulator.

Here are the results by routine name for the USRP device:

```
samples  %        image name               symbol name
```

```
2180     51.6343  USRP                      rx_data_process(void*)
626      14.8271  libc-2.4.so               memcpy
53        1.2553  libpthread-2.4.so         pthread_mutex_lock
50        1.1843  libc-2.4.so               free
50        1.1843  libomniORB4.so.0.7        omni::giopImpl12::
                                             marshalRequestHeader
                                             (omni::giopStream*)
44        1.0422  libc-2.4.so               malloc
41        0.9711  anon (tgid:17312 range:0x43040000-0x43041000) (no symbols)
38        0.9000  libomniORB4.so.0.7        .plt
36        0.8527  libomniORB4.so.0.7        omniObjRef::_invoke
                                             (omniCallDescriptor&, bool)
36        0.8527  libstdc++.so.6.0.8        (no symbols)
33        0.7816  libstandardInterfaces.so.0.0.6 standardInterfaces_i::
                                             complexShort_u::pushPacket
                                             (PortTypes::ShortSequence const&,
                                             PortTypes::ShortSequence const&)
```

Half of the CPU time is spent in the rx_data_process routine. This is the routine that read data from the USRP and writes it to the decimator via a SCA port. Fourteen percent of the time the component is calling the memcpy functions. Memcpy is part of the standard C library. Using the call graph report, the routines calling memcpy will be identified. In addition to these two routines, the remainder of the CPU time is scattered across many different routines. It is important to note that 25% of the execution is accounted for by routines that use less than 1% of the total time used by the USRP process.

Here is the call graph output for the memcpy function:

```
samples  %        image name               symbol name
  9         1.4377  libomniORB4.so.0.7       omni::giopStream::put_octet_array
                                               (unsigned char const*, int,
                                               omni::alignment_t)
  20        3.1949  libomniORB4.so.0.7        omni::tcpEndpoint::
                                              AcceptAndMonitor(void (*)
                                               (void*, omni::giopConnection*),
                                               void*)
  597      95.3674  libusrp.so.0.0.0          usrp_basic_rx::read(void*, int,
                                               bool*)
```

```
626      14.8271  libc-2.4.so              memcpy
  626      100.000  libc-2.4.so                memcpy [self]
```

The call graph shows the relationship between a particular function, in this case memcpy; the functions memcpy calls and and the functions that call mempcy. The function of interest, memcpy, is on the line where the function name is not indented. The functions called by memcpy are below it, and the functions that call memcpy are above it. In this case, memcpy does not call any functions, so all the execution time is attributed to memcpy. 96% of the calls to memcpy came from the usrp_basic_rx::read function. This function reads data received over the USB interface from the USRP. These number suggest reviewing the code in this routine to look for unnecessary memory copy operations. [1]

After the USRP device collects the data from the USRP hardware, the data is sent to the decimator to reduce the sample to 25 kHz. The decimator component receives data from an SCA port implemented with CORBA, reduces the sample and performs an FIR filter, then send the data to the FM Modulator via another SCA port implemented with CORBA. This should produce clearer results for CORBA versus component processing. Since the decimator only contains signal processing code and the CORBA code implementing the SCA ports, function usage should be clearer.

Here are the profile results by function name for the decimator component:

```
samples  %        image name               symbol name
44511    71.6659  Decimator                fir_filter::do_work(bool, short, short&)
14548    23.4233  Decimator                run_decimation(void*)
905       1.4571  libstandardInterfaces.so.0.0.6 complexShort::providesPort::
                                             pushPacket(
                                             PortTypes::ShortSequence const&,
                                             PortTypes::ShortSequence const&)
304       0.4895  libc-2.4.so              memcpy
```

For this component 96.5% of the time, the component is executing in one of three routines, the fir_filter (fir_filter::do_work), the main component thread (run_decimation), or the routine that receives data from the USRP (complexShort::providesPort::pushPacket).

From these results we can see that at first glance the CORBA related overhead is very low. We expect to see most of the processing time used by the filter. The 23% used by the run_decimation routine deserves a closer look.

[1] In the overall picture, the USRP device does not require a significant amount of processing time, so the relative amount of memcpy calls is not a serious system problem.

```
14548    23.4233  Decimator                 run_decimation(void*)
  43530    74.2035  Decimator                 fir_filter::do_work
                                              (bool, short, short&)
  14548    24.7993  Decimator                 run_decimation(void*) [self]
  491       0.8370  libstandardInterfaces.so.0.0.6 standardInterfaces_i::
                                              complexShort_u::pushPacket(
                                              PortTypes::ShortSequence const&,
                                              PortTypes::ShortSequence const&)
  69        0.1176  libstandardInterfaces.so.0.0.6 standardInterfaces_i::
                                             complexShort_p::getData(
                                              PortTypes::ShortSequence*&,
                                              PortTypes::ShortSequence*&)
```

Inspecting the call graph shows that besides the time allocated to the fir_filter, the remaining time used by run_decimation is in the routine itself. The fir_filter routine shows up since run_decimation calls it with data it receives from the USRP.

Examination of the oprofile report showing interleaved C++ source code lines with the assembly code suggests that much of the time spent in run_decimation is taken up copying the filter output into the CORBA sequences used to send data to the next component. The analysis is complicated by the fact the to compiler optimization process creates assembly code that no longer has a clear relationship with the C++ source. For this case, a simple routine was written that performs a similar function. The output for this case was compared with the output from the run_decimation routine to verify how assembler sections mapped to C++ source code.

Finally, here is the function profile for the FM Demodulator component:

```
samples  %        image name               symbol name
3156     20.5804  libm-2.4.so              sin
2803     18.2784  libm-2.4.so              cos
1885     12.2921  FM_Demod                 phase_detect::do_work(short, short,
                                           short, short, short&)
1732     11.2944  FM_Demod                 dc_block::do_work(short, short&)
1675     10.9227  FM_Demod                 run_demod(void*)
1322      8.6208  FM_Demod                 nco::do_work(short, short&, short&)
1010      6.5862  FM_Demod                 gain::do_work(float, short, short&)
193       1.2586  libc-2.4.so              memcpy
92        0.5999  libpthread-2.4.so        pthread_mutex_lock
```

```
81        0.5282  libstandardInterfaces.so.0.0.6 complexShort::providesPort::
                                                  pushPacket(
                                                  PortTypes::ShortSequence const&,
                                                  PortTypes::ShortSequence const&)
```

Once again, these routines use just over 90% of the processor time used by the component. The run_demod method uses more time than is expected, but this is again due to copying data into the CORBA sequence used to send the data to the next component.

While the CORBA sequence may appear to create a certain amount of overhead, it is not certain that changing to a different data structure would lead to much improvement. It would be informative investigating the use of the C++ vector class, or traditional C arrays. However, in the overall picture, the load due to using the CORBA sequence, while measurable, is not out of line with the overall system performance. Furthermore, there are benefits obtained by using the CORBA Sequence. The CORBA Sequence easily integrates with the mechanism used to send and receive data using CORBA, the CORBA sequence provides bounds checking and memory management. These features provide better performance in a CORBA environment and help increase radio security by providing a defined error path should a malicious user inject bad data into the radio that causes the software to write data outside the sequence.

4.4 OMAP status

The OpenEmbedded [7] (OE) build system has been updated to build a bootloader, Linux kernel and a filesystem images specifically for the Texas Instruments OMAP Starter Kit [8] (OSK). Previous work in this area contained steps that required assistance from an expert, work has been done to simplify the process for people using OE to build software for the OSK. Currently, the OSSIE specific additions to OE are being added to the OE package metadata. Once this is complete, complete software defined radio systems, based on OSSIE, can be built using OE.

OE is a general purpose system for building Linux distributions for small form factor and embedded devices. Once OSSIE is completely integrated into OE, it will be simple to build SDR's for a variety of platforms. Furthermore, as new platforms are added to OE, OS.

One concern with the OSK and the USRP approach, is that the FPGA image installed on the USRP can only reduce the sample rate to 250 ksps and these samples are transmitted to the OSK via a USB interface. When new hardware arrives on the market that may have a better architecture for software radio, only the hardware definition files in OE need

changing. Once these files are complete, the existing work on OSSIE can be reused with the new hardware.

Once the basic OSSIE system is integrated into OE, work will proceed moving the components developed for the PC test system to the OMAP processor. The FPGA bitstream in the USRP should also be modified so that the sample rate coming into the OSK is reduced from 250 ksps to 25 ksps. By reducing the sample rate, the load on the ARM processor is lowered increasing the chances for successfully running the waveform on the OSK. A single channel FM receiver should be running on the OSK by January 2007. The performance of this system will drive future work, whether to add the extra receiver and transmitter, or do further research looking for newer more capable hardware.

5 Summary

A SCA baseband FM signal processor has been developed that supports monitoring two frequencies simultaneously and allows the user to reply to transmissions on either channel. The software currently runs on a PC running the Linux operating system using the OSSIE framework. A USRP provides a RF interface to the FRS band for over the air testing. The software for this radio is available on the OSSIE project's website.

The work done for this project created and improved several software components for the OSSIE program. Improvements were made to the USRP and sound input/output components. Several new components were developed for the project; a decimator, an interpolator, a FM demodulator, a FM Modulator and a FM transceiver controller. All of these components are available from the OSSIE project web site.

The report discusses some work done to streamline the software component development for software radios based on OSSIE. This work reduces the the need for the component writer to have an in-depth knowledge of the SCA and CORBA.

Finally, some performance measurements, generated with the oprofile program, are presented. These show that the use of CORBA for inter-process communication does not contribute significantly to the amount of CPU time required by the baseband processor.

References

[1] "OSSIE." `http://ossie.mprg.org/`.

[2] R. E. Best, *Phase-locked loops, Design, Simulation, and Applications*. McGraw-Hill, 5 ed., 2003.

[3] "WinFilter." http://www.winfilter.20m.com/.

[4] "The OMG's CORBA Website." http://www.corba.org/.

[5] "Object Management Group." http://www.omg.org/.

[6] "OProfile - A System Profiler for Linux (News)." http://oprofile.sourceforge.net/news/.

[7] "OpenEmbedded — Metadata for building Linux Distributions - preferably Embedded target platforms." http://www.openwmbedded.org/.

[8] "OMAP 5912 Starter Kit (OSK) - TMDSOSK5912 - TI Tool Folder." http://focus.ti.com/docs/toolsw/folders/print/tmdsosk5912.html.