An Architecture for the Development of Software Radios on General Purpose Processors

Philip Mackenzie, Linda Doyle, Keith Nolan, Donal O’Mahony

Networks and Telecommunications Research Group
Department of Electronic Engineering / Department of Computer Science
Trinity College Dublin
IRELAND
E-mail: mackenp@tcd.ie

Abstract – Software radio is a growing technology that allows dedicated RF hardware to be replaced with more flexible software defined systems. Many different views exist on how to implement such a system. Most designs are concentrated on developing applications using dedicated DSP devices, which although providing good efficiency and power consumption, do not provide the flexibility required to meet the demands of future networks. The approach presented in this paper uses general-purpose processors and operating systems to achieve a flexible software radio design. A component-based architecture is presented which allows rapid development and testing of software radio systems. An implementation of the system is also presented.

Keywords – Software Radio, Software Components, DSP

I. INTRODUCTION

The increasing power of silicon based devices means that it is now possible to develop flexible radio terminals using software defined techniques rather than once-off dedicated hardware designs. Several commercial and open-source software projects are underway to develop software radio systems that run on both dedicated DSP and general-purpose processors (GPPs) [1]. The ability to switch between multiple wireless access schemes without changing hardware is the driving factor behind this work. Software radio will bring about rapid design, flexibility and future proofing to wireless technologies.

Historically there has been an evolution of wireless devices towards systems that perform digitisation of RF signals as close as possible to the antenna. Processing base band and IF signals digitally greatly simplifies the design of many RF systems, reduces cost and offers flexibility. Up until now, dedicated DSP processors have been the core technology associated with the development of software radio systems. Typical applications involve the digitisation of base band or IF signals, thus allowing software defined hardware to perform the remainder of RF processing tasks. This technique gives RF designers the opportunity to define their radio using software languages. However, the software languages used for DSP development are often cumbersome to use as many are based on assembly language. Most modern DSPs require intricate knowledge of a particular DSP to write effective code [2]. This greatly increases development and testing time, and ultimately increases the cost of products. DSP languages are in their infancy, as they do not provide a suitable abstraction layer to the underlying hardware. In contrast, the rapid growth of GPPs has been coupled with advances in the development of software languages. The abstraction layers to hardware have long since been removed by operating systems and many widespread programming languages exist. Using commonplace languages such as C/C++ and familiar development tools, means that RF design is opened up to a new community of software developers. Software radio development requires designs containing a high degree of flexibility. While the DSP has many advantages including power consumption and unit cost, but many believe that the flexibility and development tools of GPPs greatly outweigh these advantages.
Many challenges exist in the design of reliable software radio systems that run on GPPs. Choice of operating system, programming language and software architecture are crucial to its success. This paper concentrates on the latter and presents an architecture and subsequent implementation of a software radio system. Our aim was to develop a flexible software radio architecture that offers code reuse and scalability while maintaining good performance.

II. SOFTWARE ARCHITECTURE

This section describes a software architecture for developing software radio on general purpose processors. We start by presenting a discussion of multithreaded systems and how they affect signal processing. We present an analysis of signal processing algorithms which were used to develop a component based structure and a core processing engine.

a) Multithreaded systems

GPP systems execute programs in an entirely different way to DSPs. DSPs are devices dedicated to a particular task and clock cycles, timings, etc are fully deterministic, i.e. their execution times can be fully determined. Operating systems on the other hand use abstraction layers that cloud the ability to be fully deterministic about code execution. Operating systems use the concept of threads to allow programmers to specify concurrent tasks for the processor to execute which are known as threads. In reality, the operating system switches rapidly between these threads thus giving the appearance that they are executed simultaneously [3].

Primarily threads are used to increase user responsiveness in graphical user interfaces and to increase application performance for I/O operations. A performance boost can be achieved because multithreading allows useful work to take place while waiting for I/O or user interaction to take place. It is a common misconception that multithreaded programming can increase the performance of data intensive applications, also known as CPU bound operations [4],[5]. It can however greatly increase the performance of I/O bound operations. In this case, the operating system can recognise when a thread is waiting on input or output to complete and puts the thread into a wait state. This allows other threads to use the processor to do useful work. These factors have implications on the design of an architecture for executing DSP algorithms, thus we decided to analyse the traits of DSP algorithms that are affected by their execution on GPPs.

b) Analysis of signal processing algorithms

We examined a large set of signal processing processes. We found that the following traits of DSP algorithms were of particular importance to our system design:

1. DSP algorithms require a fixed sized block of data to perform processing
2. DSP algorithms produce a fixed size block of data determined by the size of the input and associated variables
3. DSP algorithms require all input variables to be known before processing begins

These facts seem trivial in the context of DSP programming, but offer a significant insight when considering a GPP implementation. A GPP operating system is designed to deal with many millions of events and operations per second, most occurring randomly according to the needs of the user. On the other hand, DSP algorithms are very predictable and have a straightforward execution behaviour. Having prior knowledge of the input and output parameters of a piece of logic offers the opportunity to improve the performance of a DSP operation running on a GPP. The third point suggests that DSP operations do not rely on external events such as I/O to complete. This is important when considering that the DSP algorithm will be running in a multithreaded environment where external events can change the assigned processor time slice for an operation.

Using this information it was possible to start categorising signal-processing operations into either CPU bound or I/O bound types. Further analysis of signal processing functions, in particular software radio systems found that most software radio systems only require asynchronous I/O at particular stages of processing, e.g. when acquiring digitised data from an A/D converter. The bulk of processing operations are performed by synchronous CPU intensive operations. If we had chosen a single threaded approach to the design we would have been left with inflexible code. This method would have required complicated scheduling algorithms to accomplish all the simultaneous tasks required in a radio. If we had used a single thread for each individual task then we would have increased the number of context switches required to switch between multiple threads. Instead we decided to build a component model that makes use of threads only where needed, thus taking advantage of multithreading but not taking it so far as to diminish system resources and performance.

c) Component model

It was desired to create a component structure to break down software radio functions into reusable blocks of code, e.g. components for FM
Demodulation, FIR filtering, FFT, etc. Using our knowledge of multithreading and DSP algorithms it was decided to implement a component model that would categorise components by their processing type, either CPU Bound (synchronous) or I/O bound (asynchronous). Figure 1 shows the object-orientated structure for the component types.

By separating out radio functions via these criteria, the system design can be greatly simplified. Synchronous components describe the majority of radio functions but are also the most lightweight to implement. Asynchronous types offer more functionality and are intended for more complex operations. Using this information we were able to define a simple interface to both component types.

Both component types share a common interface to change operating variables at runtime. These variables are represented by name/value properties associated with a component. This is defined by a generic component that forms the base type for all components.

```c
void SetValue(char *name, char *value);
char* GetValue(char *name);
```

Synchronous components have a simple interface consisting of 1 entry point:

```c
void ProcessData(float* input, float* output);
```

The interface to an asynchronous component is slightly more complicated as it may have to maintain state or produce data asynchronously. Thus, an asynchronous component may have to be initialised, started and stopped. Here is the interface to an asynchronous component:

```c
void Init();
void Start();
void Stop();
void ProcessData(float* input);
void ProcessNeighbours();
```

The `ProcessNeighbours()` primitive was introduced to facilitate the thread allocation discussed above and is described in the next section.

d) Software radio engine

The structure of the architecture can be seen in Figure 2.
The core element of the processing system is a processing engine. The engine is responsible for the instantiation of processing components and subsequent thread allocations. Different component configurations can be assembled to create different radios. Component configurations are described using an XML configuration file. A user interface can be used to display data to the user. This can be linked directly to the processing engine or can display information provided by particular components.

The engine dynamically assembles software radios at runtime. This happens in 3 stages:
1. The processing engine is started and reads the XML configuration file. It creates instances of all the component types required by the software radio.
2. The engine performs a thread optimization algorithm to group together asynchronous components with multiple synchronous components. This is achieved using the following two simple rules.
   - If the output of an asynchronous component is followed by 1 or more synchronous components, then they can all be grouped by thread. This means that the output of the asynchronous component will trigger a sequential execution of all the processing required by subsequent synchronous components.
   - The structure of components must always start with an A component.
   The user is notified of an error if the structure specified in the XML file does not conform to these rules.
3. The engine calls the Start method on all asynchronous components to start the transfer of data through the system.

To demonstrate this algorithm an implementation of an FM receiver is presented. Figure 3 shows a simple design for an FM receiver.

An implementation of this software radio can be realised by our system by breaking down the receiver into both synchronous and asynchronous components. In this example, asynchronous components are used to implement data acquisition and audio output. All other functions are implemented using synchronous components.

Figure 4 shows the diagrams used to represent asynchronous and synchronous components. Figure 5 shows how the FM receiver specified in Figure 3 is represented by components of the software radio system. The dotted line in Figure 5 shows how the asynchronous A/D component executes all the processing of the synchronous components using the same thread. In this example, when the asynchronous data acquisition component completes its own processing it calls the ProcessNeighbours() primitive. This causes the engine to process all the synchronous components that the engine previously assigned by thread. This causes all these components to be executed by the same thread. This greatly decreases the number of threads required to realise the system. This example also demonstrates code-reuse as two instances of the 'Filtering & Decimation' component are used in the receiver.

III. IMPLEMENTATION

a) Hardware Used

Our software radio hardware consists of a wideband receiver, custom made IF amplifier and high speed PCI A/D card (see figure 6). An asynchronous component was written to acquire the data from the A/D card. Currently a 10.7MHz IF signal is being sampled using bandpass sampling at 2MHz. Future designs will sample the signal at 20MHz.

b) Implementation on Windows 2000

An implementation of the software architecture was developed on Windows 2000 using C++. This has been used to develop many different software radio applications. The unique architecture of the system allows rapid development of software radio systems. The component structure gives a unique opportunity for the analysis of signals in the system. For example, a power spectrum display component was written that can display the power spectrum of a signal anywhere in the radio by placing it between adjacent stages. Figure 7 shows how the spectrum display component was used to display the spectrum of a 10.7 MHz signal. Figure 8 shows how the same component was used to display the spectrum of a demodulated FM signal.

The processing CPU time required by the processor varies per application. As an example, the FM application presented requires ~25% CPU time on a 2GHz Pentium 4.

Figure 3 – A simple design for an FM receiver
**Figure 4** – Diagrams representing both component types

**Figure 5** – Synchronous components are grouped to use the same thread

**Figure 6** – Software radio system hardware

**Figure 7** – Spectrum display of 10.7 MHz carrier

**Figure 8** – Spectrum display of demodulated FM
IV. CONCLUDING COMMENTS

Software radio will undoubtedly pay a role in future wireless networks and communications systems. This paper has demonstrated how software radio systems can provide a more flexible approach to the design of such systems. The use of GPPs offers the ultimate in flexibility, allowing developers to concentrate on software design rather than intricate details of RF engineering. This opens up many new avenues for experimentation and design. Currently GPP systems are not practical for the power sensitive requirements of mobile devices, but can represent a challenge to traditional base station design. The ability to upgrade via software only will open up many new opportunities for wireless communications.

V. REFERENCES

[1] Software radio projects:
   a. Vanu Inc. (www.vanu.com) are developing a commercial software radio system using general purpose processors.
   b. An open source software radio project has been started to develop software radio on a linux platform, see http://www.gnu.org/software/gnuradio


