LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH
Ollscoil Átha Cliath

TRINITY COLLEGE LIBRARY DUBLIN
The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

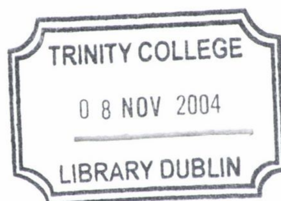
Reconfigurable Software Radio Systems

A thesis submitted for the degree of
Doctor of Philosophy

Philip Mackenzie

Department of Electronic and Electrical Engineering
University of Dublin, Trinity College

October 2004

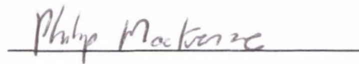


THOSIS
7449

DECLARATION

I declare that the work described in this thesis has not been submitted as an exercise for a degree at this or any other University and that, unless otherwise stated, the work is entirely my own.

I agree that Trinity College Library may lend or copy this thesis upon request.



Philip Mackenzie

October 2004

SUMMARY

Software radio has been heralded as a significant evolutionary step for wireless technology as it allows dedicated analogue radio hardware to be replaced with flexible digital signal processing. Due to current technological limitations, today's software-based radios only scratch the surface in fully exploiting the potential of this new technology. Current approaches are limited by the available hardware with software underutilised to its full capacity in most designs. The concept of reconfigurability goes a step further, placing new demands on the canonical software radio and requiring more sophisticated software designs.

This thesis explores the concept of reconfigurability in the context of software radio systems and proposes that a software-oriented component-based approach to software radio design can yield highly reconfigurable radio devices. To substantiate this claim, reconfigurability is broken down into three categories; application, structural and parametric. These categories can be used to assess the reconfigurability of a radio system and provide guidelines for their design. The design, implementation and analysis of a component-based reconfigurable radio system for general-purpose processors is presented. This system called IRIS (Implementing Radio In Software) demonstrates the concepts of reconfigurability in practice and provides insight into developing software for reconfigurable radio systems. A series of case studies are presented that demonstrate how the IRIS system and the concept of reconfigurability developed as part of this work are applicable to relevant problems in wireless communications.

The following publications directly relate to this thesis:

[Mackenzie2003] Mackenzie, P., Doyle, L. E., Nolan, K. E., Flood, D., "IRIS – A System for Developing Reconfigurable Radios", In Proceedings of the IEE Colloquium on DSP-enabled Radio, September 2003

[Doyle2003] Doyle, Linda., Mackenzie, Philip., "Exploring Reconfigurability: Towards a Future of Spectrum Rental", In Proceedings of 2003 Software Defined Radio Forum Technical Conference (SDR'03), November 2003

[Mackenzie2002b] Mackenzie P., Doyle L., Nolan K.E., O'Mahony D., "An Architecture for the Development of Software Radios on General Purpose Processors", In Proceedings of the Irish Signals and Systems Conference, pp275-280, 2002

[Mackenzie2002a] Mackenzie, P., Doyle, L., Nolan, Keith., O'Mahony, D., "Selecting Appropriate Hardware for Software Radio Systems", In Proceedings of 2002 Software Defined Radio Forum Technical Conference (SDR '02), November 2002

[Mackenzie2001] Mackenzie P., Doyle L., O'Mahony D., Nolan K., "Software Radio on General-Purpose Processors", In Proceedings of the First Joint IEI/IEE Symposium on Telecommunications Systems Research, November 2001

ACKNOWLEDGEMENTS

Firstly, my deepest gratitude goes to my supervisor Linda Doyle who not only gave me the opportunity to work on such interesting projects, but whose relentless encouragement, enthusiasm and positive attitude made my time in Trinity so enjoyable.

I would like to thank Donal O'Mahony and the members of the NTRG who all made working in this group both inspiring and enjoyable.

I really enjoyed working in the Electronics department and I would like to thank all the staff and students for making it such a pleasant place to work.

I would like to thank my fellow postgraduate students, in particular Keith and Declan for their help in all matters software radio related. A special thanks goes to Tim for all his help.

I thank all my family and friends in particular Mary, Bryan, Roz, Barbara, Gran, Ada, Frank, Aidan, Gavin and Keith for all their support.

I would like to dedicate this thesis to my parents Colin and Patricia and sister Jennifer who have always provided endless support and encouragement.

Finally, my deepest thanks go to Audrey for her constant support and patience and who without this would not have been possible.

TABLE OF CONTENTS

1 INTRODUCTION 1

1.1 Overview 1

1.2 The Basic Concept of Software Radio 2

1.3 Beyond Software Radio 5

1.4 The General Purpose Processor 8

1.5 Contribution Summary 10

1.6 Dissertation Overview 12

2 SOFTWARE RADIO 13

2.1 Introduction 13

2.2 History 13

2.3 Terms and Definitions 18

2.4 Hardware 23

2.4.1 The Ideal Software Radio 23
2.4.2 Practical Limitations 24
2.4.3 Front-End Technologies 27
2.4.4 Digital Conversion 29
2.4.5 Digital Signal Processing Devices 30

2.5 Related Work 33

2.5.1 SPECtRA and Variants 33
2.5.2 Software Communications Architecture 35
2.5.3 DSP Design Tools 37
2.5.4 Other Approaches 38

2.6 Summary 40

3 SOFTWARE ENGINEERING 41

3.1 Introduction 41

3.2 Object-Oriented Software 42

3.2.1 Overview 42
3.2.2 Object-Orientation for Software Reuse 44

3.3 The Principles of Software Components 46

3.3.1 Defining the Software Component 46
3.3.2 Objects vs. Components 48
3.3.3 Constructing Components 49

3.4 Component Composition 51

3.4.1 The Component Framework 51
3.4.2 The Component Architecture 52

3.5 Existing Component Technologies 53

3.5.1 Java Based Components 53
3.5.2 CORBA Based Component Technologies 56
3.5.3 Microsoft Component Standards 58

3.6 Summary 60

4 RECONFIGURABLE RADIO 63

4.1 Introduction 63

4.2 Reconfigurability 63

4.2.1 Reconfigurability From Hardware to Software 63
4.2.2 Reconfigurability Defined 66
4.2.3 The Benefits of a Reconfigurable Radio 67

4.3 Software for Software Radio 69

4.3.1 Reuse 69
4.3.2 Abstractions 73
4.3.3 Adaptability and Flexibility 73
4.3.4 Complexity 74
4.3.5 Security 75
4.3.6 Portability 76
4.3.7 Real-Time Behaviour 77
4.3.8 Upgrading and Versioning 77

4.4 Developing a Reconfigurable Radio 78

4.4.1 System Design Considerations 78
4.4.2 Enabling Reconfiguration 80

4.5 Summary 82

5 THE IRIS RECONFIGURABLE RADIO 83

5.1 Introduction 83

5.2 IRIS Overview 83

5.3 IRIS Architecture 85

5.4 Radio Components 86

5.4.1 Component Granularity and Component Types 87
5.4.2 Component Interfaces 88
5.4.3 Component Lifecycle 91
5.4.4 Discussion 95

5.5 Component Framework 98

5.5.1 Radio Engine 101
5.5.2 Basic XML Configurations 102
5.5.3 More Complex Radio Configurations 103
5.5.4 Internal Radio Representation 109
5.5.5 IRIS API 110

5.6 Control Logic 111

5.7 Worked Example 114

5.7.1 An FSK Transceiver 114
5.7.2 Partitioning the System 115
5.7.3 Structures 117
5.7.4 Control Logic 120
5.7.5 Reconfiguration 122

5.8 Summary 124

6 IMPLEMENTATION AND ANALYSIS 125

6.1 Introduction 125

6.2 Implementing Radio Components 125

6.2.1 Choice of Operating System 125
6.2.2 Radio Components on Windows 126
6.2.3 Programming Radio Components 127
6.2.4 Dealing with Signals 129

6.2.5 Block Size and Sample Rate 131
6.2.6 Implementing Process() 134
6.3 Radio Component Examples 135
6.3.1 Worked Example 135
6.3.2 Signal Processing Components 139
6.3.3 IO Components 140
6.3.4 Testing Components 141
6.3.5 Visualisation Components 141
6.4 Using the Component Framework 142
6.4.1 IRIS API 142
6.4.2 Tools 146
6.5 External Hardware 148
6.6 Analysis 150
6.6.1 Scalability 150
6.6.2 Memory Consumption 153
6.7 Summary 158
7 CASE STUDIES 159
7.1 Introduction 159
7.2 Over the Air Reconfiguration 159
7.2.1 Overview of Over the Air Reconfiguration 159
7.2.2 Applying IRIS to OTAR 160
7.2.3 Conclusions 164
7.3 Wireless Networking 164
7.3.1 Overview of Wireless Networking using DAWN 164
7.3.2 Applying IRIS to Wireless Networking 166
7.3.3 Conclusions 169
7.4 Spectrum Management 169
7.4.1 Overview of Spectrum Management 169
7.4.2 Applying IRIS to Spectrum Management 170
7.4.3 Conclusions 173
7.5 Summary 173
8 CONCLUSIONS 174
8.1 Introduction 174
8.2 Summary of Contributions 174
8.3 Future Work 175
8.3.1 Hardware 176
8.3.2 Software 176
8.3.3 Security 177
8.3.4 Spectrum Management 178
8.4 Conclusion 179
9 BIBLIOGRAPHY 180
10 APPENDIX 190
10.1 Methods Exposed by a Radio Component 190
10.2 Code Generator Commands 192

TABLE OF FIGURES

Figure 1.1 – Typical Receiver Using Traditional Analogue Hardware 3
Figure 1.2 – Typical Receiver Using the Software Radio Approach 3
Figure 1.3 – Software Radio Phase Space Diagram [Mitola99a] 8
Figure 2.1 – European Funded Projects Relating to Software Radio 17
Figure 2.2 – The Ideal Software Radio 24
Figure 2.3 – A More Practical Software Radio Solution 25
Figure 2.4 – Research and Development in Software Radio 27
Figure 2.5 – SPECtRA In-Band and Out-of-Band Paths [Bose99b] 34
Figure 2.6 – Software Structure of the SCA [JTRS2002] 36
Figure 3.1 – Summary of Software Engineering Principles 60
Figure 4.1 – Level of Reconfigurability for Various Signal Processing Devices 64
Figure 4.2 – A Tightly Coupled Software Component 71
Figure 4.3 – Different Approaches to Reconfigurable Radio System Design 79
Figure 5.1 – Receiver and Transmitter Example 84
Figure 5.2 – A Reconfigurable Radio System 85
Figure 5.3 – A Reconfigurable Radio with User Interaction 85
Figure 5.4 – The IRIS Radio Architecture 86
Figure 5.5 – Relationship of Component Types 88
Figure 5.6 – External View of a Radio Component 88
Figure 5.7 – Lifecycle Interface 89
Figure 5.8 – Parameter Interface 89
Figure 5.9 – Event Interface 89
Figure 5.10 – Port Interface 90
Figure 5.11 – Command Interface 90
Figure 5.12 – Reflection Interfaces 90
Figure 5.13 – Component Information Interface 91
Figure 5.14 – Abstract RadioComponent class 91
Figure 5.15 – Sequence Diagrams of DSP and IO Component Lifecycles 94
Figure 5.16 – Sequence Diagram of Standalone Component Lifecycle 95
Figure 5.17 – Radio Component Showing Interfaces Implemented by Code Generation 96
Figure 5.18 – RadioComponent 97
Figure 5.19 – The IRIS Radio Architecture 99
Figure 5.20 – Flow Diagram for Creating a Reconfigurable Radio 100
Figure 5.21 – Interaction of Radio Engine, Radio Components and Control Logic 101
Figure 5.22 – Basic Series of Components 102

Figure 5.23 – A Duplicated Signal Path 104
Figure 5.24 – Synchronisation in IRIS 105
Figure 5.25 – Multiple Synchronous Signal Paths 106
Figure 5.26 – Multiple Asynchronous Structures 106
Figure 5.27 – An Embedded Structure 107
Figure 5.28 – Signal Routing 108
Figure 5.29 – A Component with 2 Input Channels 108
Figure 5.30 – Synchronisation of Processing 109
Figure 5.31 – Internal Representation of Radio System 110
Figure 5.32 – Interface of the IRIS API 110
Figure 5.33 – Component Dependency 111
Figure 5.34 – Using Control Logic to Eliminate Component Dependencies 111
Figure 5.35 – Interface Control Logic uses to Control Radio 112
Figure 5.36 – Controller Interface 112
Figure 5.37 – Lifecycle of Control Logic 113
Figure 5.38 – FSK Transceiver Design 115
Figure 5.39 – FSK Waveform 116
Figure 5.40 – Partitioning of FSK Transceiver into Software Components 117
Figure 5.41 – Multithreading Approaches 118
Figure 5.42 – XML Configuration for FSK Transceiver 119
Figure 5.43 – FSK Transceiver Using Control Logic 121
Figure 5.44 – Sample Control Logic Source Code 122
Figure 5.45 – Code for Replacing a Component at Runtime 123
Figure 6.1 – Exporting a Component from a DLL 127
Figure 6.2 – Header File of a Signal Strength Component 128
Figure 6.3 – Data Types Supported by IRIS 129
Figure 6.4 – Sequential Layout of Samples and Channels in Memory 131
Figure 6.5 – Automatic Calculations Performed by the Framework 133
Figure 6.6 – Struct Definition used by Process() 134
Figure 6.7 – Example Process() Method 134
Figure 6.8 – Signal Format Struct 134
Figure 6.9 – Properties of FSK Component 135
Figure 6.10 – C++ Header File Definition of FSK Modulator Component 136
Figure 6.11 – XML Generated to Describe the FSK Modulator Component 137
Figure 6.12 – Code to Implement Data Received Port 138
Figure 6.13 – FSK Waveform 138
Figure 6.14 – XML for Configuring an FSK Modulator Component 139
Figure 6.15 – IRIS Screenshot of Received FM Signal 142

Figure 6.16 – Code to Create a Reconfigurable Radio 143
Figure 6.17 – Application Specified Control Logic 144
Figure 6.18 – Sample Code for Creating Application-Defined Control Logic 145
Figure 6.19 – Screenshot of Parameter Controller 146
Figure 6.20 – Radio Designer User Interface Screenshot 147
Figure 6.21 – Receiver Hardware Setup 148
Figure 6.22 – IRIS Test Hardware 149
Figure 6.23 – Scalability Test Scenario 151
Figure 6.24 – IRIS Scalability Test Results 152
Figure 6.25 – Difference between IRIS and Native Implementation 153
Figure 6.26 – Memory Allocation Technique 155
Figure 6.27 – Memory Allocation for Multiple FIR Filters 155
Figure 6.28 – Memory Test for Down Sampler Scenario 156
Figure 6.29 – Memory Consumption for Multiple Down Samplers 156
Figure 6.30 – Memory Test for Up Sampler Scenario 157
Figure 6.31 – Memory Consumption for Multiple Up Samplers 157
Figure 7.1 – Items for Download with OTAR 161
Figure 7.2 – Software Download with Control Logic 162
Figure 7.3 – Software Download Using the IRIS API 163
Figure 7.4 – Typical DAWN Topology 165
Figure 7.5 – A Typical DAWN Stack 166
Figure 7.6 – IRIS Incorporated into DAWN 167
Figure 7.7 – FSK Transceiver with Interference Temperature Detector 171
Figure 7.8 – Spectrum Monitoring System 172

ACRONYMS

ADC	Analogue to Digital Converter
AM	Amplitude Modulation
AMPS	Advanced Mobile Phone System
ASIC	Application Specific Integrated Circuit
API	Application Programming Interface
ASIO	Audio Streaming Input Output
ATM	Asynchronous Transfer Mode
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
CCM	CORBA Component Model
CF	Core Framework (Relating to SCA)
CLR	Common Language Runtime
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
DAC	Digital to Analogue Converter
DAWN	Dublin Ad hoc Wireless Network
DC	Direct Current
DCOM	Distributed Component Object Model
DCS	Digital Cellular System
DDR	Double Data Rate
DECT	Digital Enhanced Cordless Telecommunications
DER	DSP-enabled Radio
DLL	Dynamic Link Library
DSP	Digital Signal Processing
EJB	Enterprise JavaBeans
FCC	Federal Communications Commission
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FM	Frequency Modulation
FPAA	Field Programmable Analogue Array
FPGA	Field Programmable Gate Array
FSK	Frequency Shift Keying
GHz	Gigahertz
GPP	General Purpose Processor
GMSK	Gaussian Minimum Shift Keying
GSM	Global System for Mobile Communications
I and Q	In-Phase and Quadrature
IDL	Interface Definition Language
IIOP	Inter-ORB Interoperability Protocol
IF	Intermediate Frequency
IO	Input/Output
IRIS	Implementing Radio In Software
ISI	Inter-Symbol Interference
JAR	Java Archive
JPO	Joint Program Office
JTRS	Joint Tactical Radio System
JVM	Java Virtual Machine
LAN	Local Area Network
MAC	Medium Access Control
MHz	Megahertz
MMITS	Modular Multifunction Information Transfer System
NCO	Numerically Controlled Oscillator
OFDM	Orthogonal Frequency Division Multiplexing
OO	Object Oriented
OOD	Object Oriented Design
OOP	Object Oriented Programming
ORB	Object Request Broker
OTAR	Over The Air Reconfiguration
PCI	Peripheral Component Interconnect
PCS	Personal Communications Services
PDA	Personal Digital Assistant
POSIX	Portable Operating System Interface
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memory
RCP	Reconfigurable Communications Processor
RCF	Radio Component Framework
RDL	Radio Description Language
RDS	Radio Data System
RISC	Reduced Instruction Set Computer
RF	Radio Frequency
SCA	Software Communications Architecture
SDR	Software Defined Radio
SFDR	Spurious Free Dynamic Range
SNR	Signal to Noise Ratio
TDMA	Time Division Multiple Access
UHF	Ultra High Frequency
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
WDL	Waveform Description Language
XML	eXtensible Markup Language

1 Introduction

1.1 Overview

This dissertation shows that a software-oriented component-based approach to software radio design can yield highly reconfigurable radio devices.

A software radio or software-defined radio is a wireless communications device that can be reprogrammed to allow it to communicate using different modulation schemes and frequencies without altering or replacing hardware [Mitola92]. The software radio uses a generic piece of hardware and digital signal processing (DSP) to manipulate radio signals. This allows the core of the communications system to be developed in software rather than analogue hardware components. The software approach results in flexible radio devices that can be easily reprogrammed allowing the functionality of the device to be changed. It also means that the communications techniques themselves can become flexible and adaptable. These advantages have brought about a decade of research and development in an effort to make the technology a reality. However, software radio is still in its infancy. Due to technological limitations, current software radios only scratch the surface in fully exploiting the potential of this new technology. This thesis focuses on recognising and developing the potential of the technology by demonstrating how highly reconfigurable radio systems can be created.

It is important to discuss what the term 'reconfigurable' means in relation to the research presented in the thesis. In the context of wireless communications the term 'reconfigurable' suggests a type of radio device that offers flexibility in the functions it provides, perhaps being capable of receiving at multiple frequencies or transmitting using multiple modulation schemes. However many different types of radio systems offer these functions and it can become difficult to determine whether one radio system is more reconfigurable than another. This raises a fundamental question; how can reconfigurability be measured? In this thesis reconfigurability is defined via three categories namely, application, structural and parametric reconfigurability. Defining such categories allows the level of reconfigurability of a device to be assessed. The more ways in which a device exhibits traits of each category, the more reconfigurable the device becomes.

Reconfigurability is a desirable property in a radio system as it enables a whole host of new capabilities. It allows many parameters, that have traditionally been fixed (such as frequency,

modulation scheme, power, bit rate, etc), to become variable. This allows truly flexible devices to be created and in turn facilitates the development of new applications. The three types of reconfigurability and a general discussion of reconfigurability itself are presented in Chapter 4.

This thesis uses a software-oriented methodology in tackling the problem of developing software for radio systems. It is therefore necessary to discuss explicitly the 'software' of 'software radio systems' in order to differentiate this research from other approaches. Current research and development in this field is driven (and also limited by) the capabilities and availability of hardware. The focus has been on developing fast, inexpensive hardware and therefore research into the software aspects of software radio have received less attention. It is argued in this thesis that a software-oriented approach to radio system design is essential in achieving reconfigurability. A central part of this software-oriented approach is the use of software components and a component framework, both of which are discussed in detail in Chapter 3. The research presented in the thesis shows how software components and component frameworks can be used to deliver better reconfigurability in radio devices. These concepts are discussed theoretically and also practically through a real-life implementation. Chapters 5 and 6 demonstrate the design, implementation and analysis of a software system called IRIS (Implementing Radio In Software) that was developed as part of this research. IRIS is a component framework that facilitates the development of highly reconfigurable software radio systems.

The remainder of this chapter provides an overview of the key ideas of this thesis. Section 1.2 introduces the basic concept of software radio. Section 1.3 discusses reconfigurability, how it is different from the canonical software radio, and briefly describes the differences between this research and other work in the field. Section 1.4 discusses the motivation for using the general-purpose processor as the target platform in this work. The contributions made by this thesis are summarised in Section 1.5. Finally, Section 1.6 describes the layout of this dissertation.

1.2 The Basic Concept of Software Radio

In a traditional analogue radio transceiver, signals are received and transmitted using analogue hardware components. The radio's hardware design is determined by its end-application, for example a two-way radio, an FM audio receiver or a BPSK data-transmitter will each have different requirements in operating frequency, modulation scheme bandwidth and power. Consequently, if for example a QPSK data transceiver is required for operation at 500MHz, then the analogue circuitry will be built to implement this design and this design only. In this case the hardware used is dedicated to the particular application and the operating parameters cannot be changed, modified or upgraded without altering the hardware design.

Figure 1.1 illustrates a traditional analogue radio receiver. The signal of interest is tuned and amplified at the reception frequency before being down-converted to an Intermediate Frequency (IF) using the superheterodyne approach [Armstrong24] (IFs and the superheterodyne approach are discussed in more detail in Chapter 2). At the IF, the signal is further amplified and filtered before the original signal is recovered. This approach to radio design has dominated since the early 1930s.

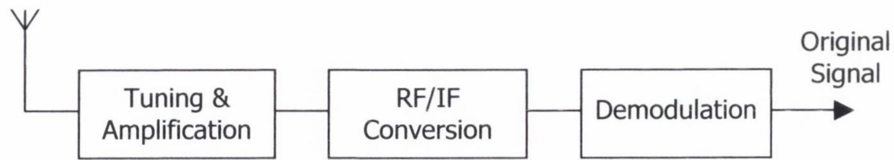


Figure 1.1 – Typical Receiver Using Traditional Analogue Hardware

In a software radio, dedicated analogue hardware is replaced with a combination of a minimal RF-front end, a digital converter and digital signal processing hardware. The functionality of the device is defined via software programming, therefore the operating characteristics of the radio can be reprogrammed and changed without altering any hardware. In contrast to the hardware radio of Figure 1.1, Figure 1.2 shows a diagram of a software radio-based receiver. In this scheme the IF signal is converted to a digital signal using an analogue to digital converter. Digitisation results in a stream of numeric samples that are processed mathematically using digital signal processing (DSP[1]) to recover the original signal. Likewise, the same approach can be used in transmitters. In this case DSP is used to synthesise signals digitally before being converted to an analogue signal for transmission.

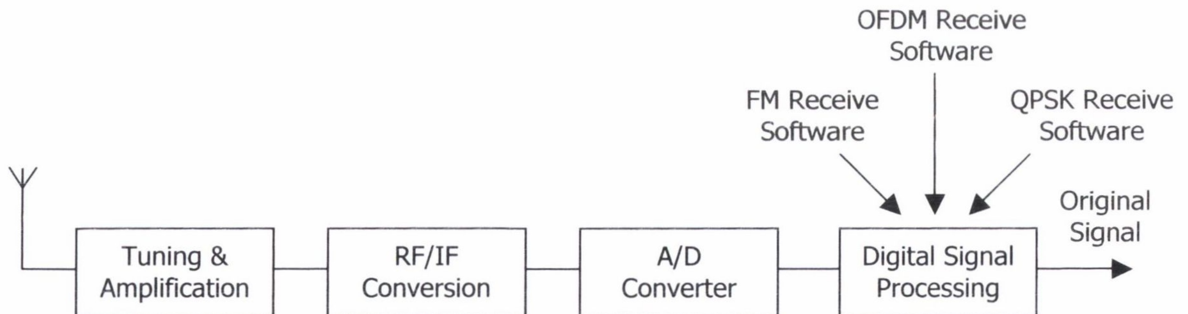


Figure 1.2 – Typical Receiver Using the Software Radio Approach

Two things make this type of design distinctly different to the analogue approach. Firstly, the radio signal is processed digitally, which makes it possible to process signals using methods that are difficult to implement with analogue electronics. Secondly, the signal processing algorithms can be upgraded, replaced and reprogrammed through software, which allows one piece of generic

[1] The term 'DSP (Digital Signal Processing)' should not be confused with the term 'Digital Signal Processor'. The latter will be referred to as a 'DSP Processor' to differentiate these terms.

hardware to act as many different radio devices. This approach to radio design, i.e. the ability to replace dedicated analogue radio hardware with a combination of digital hardware and varying software implementations, forms the basis of software radio [Mitola95].

Central to the software radio concept is the use of DSP for manipulating radio signals. In a software radio, DSP replaces the functionality previously implemented using analogue components and moves radio signals into the digital domain. For example, using DSP, a device such as an analogue low-pass filter can be implemented by a digital algorithm that achieves an equivalent result. Such a paradigm shift raises the question as to what exactly are the advantages of moving to DSP. Just as with many other engineering applications, however, it is well recognised that DSP techniques have many advantages over analogue signal processing.

Lapsley [Lapsley97] describes three ways in which DSP differs from analogue signal processing. Firstly, DSP systems exhibit insensitivity to their operating environment. In an analogue circuit operating conditions are dependent on component tolerances and temperature, whereas a working DSP system always produces consistent results. This fact means that DSP systems can, in the majority of cases, offer more predictable behaviour than an analogue design whose characteristics can be influenced by a variety of external factors. Secondly, DSP systems have the advantage of being insensitive to component characteristics. Physical characteristics such as size and component packaging can often influence the decision to use a particular analogue component. Also, economic factors such as component cost and availability can influence design decisions. DSP systems do not suffer from these limitations because designs are specified via mathematical procedures and not components. Finally, DSP has become a less expensive and overall more popular approach than analogue design because analogue electronic design tends to be much more difficult.

However, even though these advantages exist for DSP, it is always possible to develop an analogue device that outperforms even the most powerful DSP device. This raises another question as to what the fundamental difference is between these two approaches. The answer is that unlike analogue hardware, a DSP system can be reprogrammed to do many different things. A DSP algorithm can exist as a set of instructions, which can be changed and manipulated without altering hardware. This ultimately means that software can be used to implement DSP algorithms allowing generic programmable radio devices to be created. This cannot be done with a traditional analogue hardware-oriented approach to radio system design.

Technology has advanced to such a stage that it is now possible to design, implement and test a radio system in software rather than designing, building and physically prototyping analogue radio circuitry. Using software, instantaneous changes to a radio system can be made that previously

required a complete redesign. Software thus brings a significant change in how radio systems can be designed, built, tested and viewed.

1.3 Beyond Software Radio

It is possible today to build reprogrammable radio devices that are software-defined. The next step in the software radio space is a move towards reconfigurability [Pereira99, Drew2001, Dillinger2003]. The term reconfigurability has emerged over the past few years to demonstrate a shift in thinking in the software radio space. The reconfigurability concept is about making the software radio do more, applying the technology beyond the radio domain and ensuring its impact throughout the communications system [Pereira2000].

Whereas the software-defined radio approach can be used to define the air interface of a communications system, this definition is typically created only once. In practice the software defined radio concept has come to mean software upgrades, bug fixes and new features, rather than fully exploiting the capability of the technology. Reconfigurability on the other hand recognises that the software of a radio system does not have to be defined once, but can be changed and augmented any number of times to serve a greater purpose throughout the communications network.

This overall concept is best demonstrated by an example. Governments must regulate the use of spectrum to particular frequencies and modes of operation to ensure interference-free communication. However most of the time radio devices are completely underutilising the available spectrum. The user is limited to a particular frequency and bandwidth even though massive amounts of bandwidth exist across the entire RF band. These restrictions are often imposed across an entire country or regional area even though they are used exclusively in particular locations.

In a reconfigurable radio every parameter of the radio system is potentially variable and implemented in software. This means that a device can dynamically reconfigure itself to make better use of the available spectrum. A reconfigurable radio may increase its operating bandwidth to use additional spectrum when operating in a remote location. It may alter its power to avoid interference when in a crowded office block. It may negotiate with another node in the network to agree on a particular modulation scheme to suit its location. This type of capability is not possible with dedicated hardware solutions as they are not reconfigurable in the same way. It is also not possible with the canonical software defined radio as it typically signifies reprogramming of hardware to upgrade or fix bugs in software, rather than having built-in dynamic behaviour. In contrast the reconfigurable radio can change any operating parameter, instantaneously change the

structure of the radio system or automatically download new software to enable new features; the overall aim being to improve communication. The term 'reconfigurable radio' is therefore used in this thesis to describe devices that are more dynamic than the canonical software radio.

The approach in this thesis is quite different to other software radio research and no other previous work in the field has taken the particular approach presented here. This work is unique as it concentrates on reconfigurability and how to deliver this using component-based software. However, some other systems, although not focusing directly on reconfigurability, have similarities to this work either through their involvement in software radio or through their approach to DSP. The following discussion highlights work by others in the field of software radio. It concentrates on work that has elements in common with this thesis. A more thorough examination of these and other systems is presented later in Section 2.5.

SPECtRA is a programming library for software radio and was the first project to demonstrate working software radio implementations on GPPs [Bose99a]. The focus of that project was to demonstrate the feasibility of software radio on general-purpose processors (GPPs). The IRIS system developed as part of this work is also developed using the GPP as a platform, however the work in this thesis is distinctly different. Firstly, while a GPP has been used for this research, this thesis does not attempt to propose that the GPP is the best platform for building radio systems. Instead, it recognises that the GPP is the most convenient solution within current technical capability for demonstrating the concept of reconfigurability. In time as technology improves there may be other better platforms for developing radio systems, but the concepts of reconfigurability discussed in this thesis will still be applicable.

Secondly, SPECtRA is a C++ based programming library for developing software radio systems. In contrast, the IRIS system of this work is a component framework. This component framework formalises an approach to building radio systems and applies software engineering principles to their development. Finally, the SPECtRA system was not focused on reconfigurability. While it may be possible to use SPECtRA to develop a reconfigurable radio, the system itself has not been designed with this as a focus. In contrast, the IRIS system is built from the ground up to facilitate reconfigurability. It allows dynamic loading/unloading of software components and formalises the reconfiguration process. Further technical details of SPECtRA and other variations of this system are discussed in Chapter 2, Section 2.5.1.

The SCA (Software Communications Architecture) [JTRS2001] of the U.S. JTRS (Joint Tactical Radio System) project is a standard for military software radio systems. This SCA has its roots in one of the first ever software radio projects called SPEAKeasy [Lackey95] (SPEAKeasy is discussed later in Chapter 2, Section 2.2). The JTRS is a large comprehensive standard for defining

radio systems and concentrates on partitioning the system and defining interfaces between elements of a radio system. The SCA is very different to the IRIS system developed as part of this thesis. The SCA concentrates on military interests, in reducing the cost of their radio systems and introducing interoperability into their systems. The SCA does allow for limited reconfigurability within individual elements of the radio system, however the system as a whole is quite rigid. It is focused on aspects of communications that are not so relevant to this research such as developing tamper-proof radio systems. It also does not mandate a software approach; instead the interfaces defined can be implemented in hardware. For these reasons the SCA is unsuitable for exploring reconfigurability. The SCA is discussed in more detail in Chapter 2, Section 2.5.2.

There are also some generic signal processing environments that can be discussed in the context of this research. Among those is Ptolemy, a software project from Berkley MIT that provides an environment for modelling, simulation and design of signal processing algorithms [Buck94]. Central to Ptolemy is the concept of models of computation, a facility that provides a highly expressive environment for representing different types of signal-based systems. Although Ptolemy could be used to model and simulate specific algorithms for software radio it is distinctly different to the work of this thesis. Firstly, Ptolemy is a tool for modelling and simulation. IRIS is not a tool but a component framework for developing real software systems. Although Ptolemy can potentially generate source code for a variety of platforms the way in which it views its targets is quite different to IRIS. IRIS reuses blocks of signal processing logic as software components, whereas the blocks existing in Ptolemy exist at design-time only. These blocks are eventually collapsed down to an implementation that is fixed in function. In contrast, the IRIS system is designed so that the actual system developed can constantly reconfigure. While Ptolemy is a useful tool for developing signal-processing systems, it is not a suitable platform for exploring reconfigurability as its focus is developing and merging models of computation, a completely different paradigm that does not address the needs of reconfigurable radio systems. Other tools that fall into this category are Matlab and Simulink [Mathworks], and SPW (Signal Processing Worksystem) [Cadence2002] which are discussed in Chapter 2, Section 2.5.

In summary, the work presented in this thesis is different from other approaches to software radio as it focuses on reconfigurability. Reconfigurability is achieved through a software-based approach. Reconfigurability is necessary to deliver flexible radio systems capable of meeting the demands of future applications such as dynamic spectrum management. The reconfigurability concept is demonstrated in this thesis through the IRIS system, a component framework developed for GPPs. The next section discusses why the GPP has been chosen as the basis for this work.

1.4 The General Purpose Processor

The IRIS system developed as part of this work is designed to run on GPPs such as the Intel Pentium and the implementation presented in this thesis runs on the Windows platform. The GPP has been chosen as it provides the best platform for demonstrating the concepts of reconfigurability, which are the main focus of this research. This section discusses the motivation for this choice.

Mitola summarises the differences in platforms for software radio in the phase space diagram reproduced here in Figure 1.3 [Mitola99a]. This diagram plots various radio communication applications and how they are typically implemented according to bandwidth and hardware device. Mitola draws a tangent across the plane of radio applications and indicates a shift towards what he consideres the ideal software radio as technology improves over time. For example, 'B' shows how COTS (Commercial Off The Shelf) handsets typically process baseband signals using ASIC or FPGA technology. Likewise, 'X' indicates the ideal software radio (discussed in detail in Chapter 2, Section 2.4.1) and shows this to be a device implemented using general-purpose processors and operating with a digital access bandwidth in the GHz range. The plane cutting diagonally across the diagram indicates the current state of the art in technology and as the 'Technology' arrow indicates, increases in technological capability bring us closer to the ideal software radio.

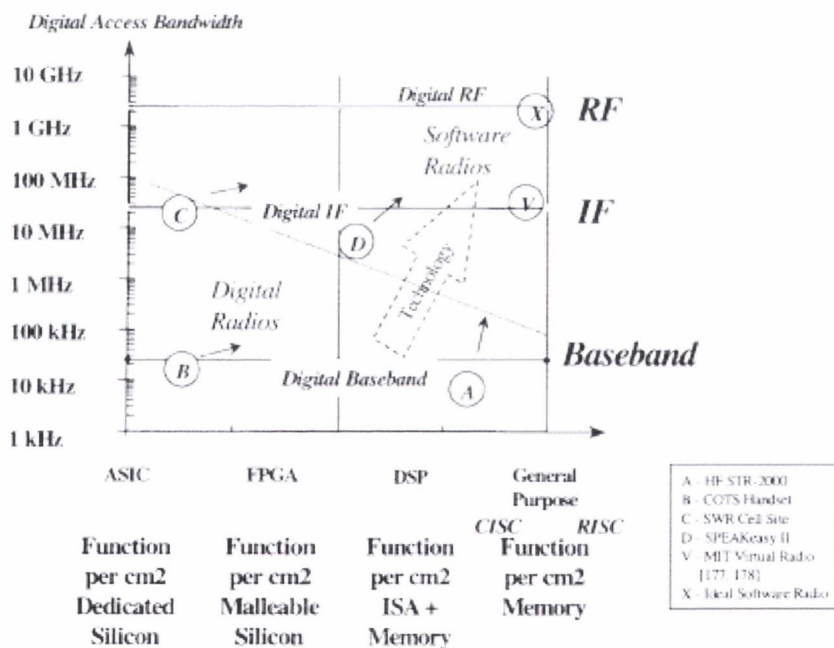


Figure 1.3 – Software Radio Phase Space Diagram [Mitola99a]

If the work of this thesis were to be mapped onto Mitola's phase space diagram it would sit towards the 'X' region. This is because this work uses the GPP as a platform for demonstrating

reconfigurability, and also, this work is more focused on the idealistic software radio rather than a hardware oriented approach.

It is worth discussing why the GPP is a suitable platform for demonstrating reconfigurability, as it is not typically chosen as a platform for real-time signal processing applications. Unlike most other work in the software radio space, this thesis is not concerned with developing the most efficient, low-power, low-cost device. Instead, its primary concern is demonstrating the fundamentals of reconfigurability. Although the GPP is limited in processing power and is unsuitable for embedded or low-power applications, the flexibility of the platform makes it an ideal candidate for demonstrating the concepts of reconfigurability. The GPP has the following advantages over more traditional embedded devices:

- *Readily Available Hardware:* Embedded systems require custom hardware to be designed and built which is a very costly and time consuming process. While any embedded system could potentially contain any combination of RAM, persistent storage or I/O peripherals, the system has to be designed specifically for these hardware components. Also, hardware and software interfaces (i.e. drivers) have to be developed to interface these components on the embedded target. In contrast, GPPs come in the form of readily available PCs. This hardware is relatively inexpensive (compared to the cost of designing an embedded system from scratch), requires no custom hardware design, and even the most basic PC contains large amounts of RAM and persistent storage as standard. RAM is useful for software radio as the high sample rates involved in radio systems result in large amounts of sampled data. This data can be buffered in memory. Persistent storage is also important as a software radio can use this space to store large files of waveform data and potentially any number of different radio configurations.
- *Advanced Languages and Tools:* The GPP computer is a pervasive technology thus many different languages and development tools exist for developing GPP software. By developing software radio on this platform engineers can take advantage of these advanced tools and languages. Unlike other platforms that require output to a target platform for testing, testing on a GPP environment is much easier as it can be performed alongside development.
- *Operating System:* The GPP uses an operating system that provides services such as memory management, concurrency and file systems which make it much easier to develop applications. These services relieve the programmer of having to deal with hardware specific memory layouts, etc which are common on embedded platforms.
- *Moore's Law:* Moore's Law, a popular observation on semiconductor technology, states that the processing power of semiconductors doubles roughly every 18-24 months [Moore65]. This means that an automatic increase in processing power becomes available every 18-24 months. This is significant as an embedded design typically requires a full redesign to increase its capability to this degree.

There are two particular limitations to using the GPP platform which are worth discussing; power consumption and real-time behaviour. The power consumption of a GPP is much higher than a DSP processor or an FPGA, therefore the GPP is unsuitable for low-power mobile applications. However, as discussed, this thesis is not concerned with developing the most power-efficient device, rather its focus is on demonstrating the concept of reconfigurability. With further research and with a focus on reconfigurability, a GPP processor could be developed that meets the needs of the reconfigurable radio with low-power design.

Real-time behaviour is also a concern for many, as operating systems such as Windows and Linux are pre-emptive and thus inherently non real-time. In these systems the kernel has full control over scheduling of processes on the computer and there is nothing to stop a radio application from being pre-empted by any other process running on the system. Ultimately this means that the system may not be able to meet its stringent timing requirements and thus would be defective as a communications system. There are two reasons why this concern is of lesser importance in the context of this work. Firstly, this thesis proposes that reconfigurability can make improvements over traditional radio system design, and perhaps it is possible to develop radio systems that are not so dependent on real-time constraints. Data transmissions for example are often transmitted in bursts, are irregular and can tolerate occasional errors. New types of radio systems developed with inherent reconfigurability could be designed to treat radio signals in the same way as data transmissions, therefore there would be a lesser need for stringent real-time behaviour. Secondly, while mainstream GPP operating systems do not support real-time operation, many commercial products exist either as add-ons, or separate operating systems, that provide real-time operation on a GPP (for example, VenturCom provide a real-time extension to Windows XP called RTX [VenturCom]). Future work could look at implementing a reconfigurable radio system on one of these real-time operating systems. Thus, for the purposes of this work the advantages of the GPP greatly outweigh the disadvantages and therefore it has been chosen as a platform for this research.

1.5 Contribution Summary

As stated in Section 1.1 this work shows that that a software-oriented component-based approach to software radio yields highly reconfigurable radio devices. That thesis is proven through the following five contributions:

A comprehensive overview of software radio technology

A comprehensive overview of software radio is provided in the thesis. It does this by analysing the history of the field, contrasting different terms and definitions, examining the technologies involved, and looking at previous work in the field. This is an important contribution because it looks at software radio from various perspectives and does not focus on specific applications such

as mobile telephony. This should serve as a useful guide to others carrying out research in this field.

Categories for assessing reconfigurability in radio systems

Three categories of reconfigurability, namely; application, structural and parametric reconfigurability are defined in the thesis. These categories allow the level of reconfigurability of a radio device to be assessed. This is an important contribution to the field of radio system design as it gives others the means to contrast and compare different radio systems in terms of their reconfigurable capability. This also serves as a means to defining the software requirements of the reconfigurable radio system.

Analysis of software design for radio systems

To analyse software design for radio systems this thesis looks at eight software engineering principles, namely reuse, abstractions, adaptability and flexibility, complexity, security, portability, real-time behaviour and finally upgrading and versioning. This unique perspective on software design for radio systems provides a valuable contribution as it highlights the differences between developing software for a software radio system and developing mainstream software. This is useful as it shows that in many cases the best practices in mainstream software are not necessarily applicable when developing radio systems. This contribution also demonstrates that component-based software is an effective way to achieve reconfigurability.

Design, implementation and analysis of a reconfigurable radio system

The design, implementation and analysis of IRIS provides a valuable contribution as it presents practical information that will help others to build reconfigurable systems. It demonstrates how the reconfigurability concepts presented in this thesis can be applied in practice. It is also important that IRIS has been developed on GPPs as this shows that this is a suitable platform for developing and experimenting with radio system concepts.

Case studies that apply the reconfigurable radio approach

Three case studies, that demonstrate how the IRIS system and hence the reconfigurable radio approach can provide unique capabilities that facilitate new and emerging types of radio systems, are presented. This is an important contribution as it brings together reconfigurability and practical problems in wireless design to prove that the software-based component-oriented approach taken in this work is a practical and effective way of developing highly reconfigurable radio devices.

1.6 Dissertation Overview

Chapter 2 presents the history of software radio, contrasts terms and definitions, describes the technical issues and discusses related work in this field. The software-oriented component-based approach presented in this thesis requires background knowledge in software engineering; this is presented in Chapter 3. Chapter 4 is the most important chapter in this dissertation as it presents the unique approach and key concepts that differentiate this work. That chapter defines the three categories of reconfigurability; application, structural and parametric that can be used to assess the overall reconfigurability of a radio device. This chapter then goes on to discuss all the issues involved in realising such a device by analysing the role of software in radio systems. Chapters 5 and 6 demonstrate how these concepts have been used to develop a real-life reconfigurable radio system called IRIS. IRIS is highly reconfigurable radio system that runs on normal PCs providing an ideal experimental platform for demonstrating the concept of reconfigurability. Chapter 7 shows how both the concepts of reconfigurability and the IRIS system itself are applicable to emerging wireless technologies. Chapter 8 summarises conclusions from this work and suggests areas for future investigation.

2 Software Radio

2.1 Introduction

The purpose of this chapter is to give a comprehensive overview of software radio technology and to demonstrate why the approach taken in this thesis is different to other work in this field. The chapter is broken down as follows:

Section 2.2 describes the history and evolution of software radio. This section first discusses an early software radio system called 'SPEAKeasy' and then plots the development of the technology through to the present day. Section 2.3 contrasts the various terms and definitions used to describe the software radio concept and arrives at a definition suitable for the work in this thesis. Section 2.4 discusses the role of hardware in software radio. It starts by discussing the 'Ideal Software Radio' and looks at the practical limitations involved in developing a software radio system. It also describes the various hardware technologies involved in creating software radio system and discusses relevant prior research relating to this thesis. Section 2.5 discusses work related to this thesis and demonstrates the need for a reconfigurable approach to software radio.

2.2 History

Mitola coined the phrase 'Software Radio' in 1991 and in 1992 he wrote the first publication on the topic [Mitola92]. This publication explained some of the basics of software radio discussing A/D and D/A conversion, sampling rates and hardware, but even more importantly predicted a decade of change from hardware to software-based radio systems.

Mitola's contribution in the early 90s was not that he invented software radio itself, but that he marked a shift in thinking by introducing this new term. In fact the software radio concept had been evolving for many years with digital-signal processing being increasingly used in many aspects of electronic design. The term 'software radio' marked the arrival of digital signal processing into the field of radio system design.

At the time software radio was seen as an ideal technology for military applications. It would provide flexibility and interoperability to organisations that relied heavily on communications

infrastructure. In 1991 the U.S. Department of Defence began a project called SPEAKeasy [Lackey95, Bonser98]. The aim of SPEAKeasy was to develop a common communications device allowing inter-communication among military allies. The basic problem it addressed was that multiple radio standards and implementations existed, with no interoperability between devices and no common hardware platform. A software-based signal-processing solution was seen as a way to overcome this problem. By manipulating radio signals digitally, they would be able to have one common hardware platform with various radio standards supported via different software programmes.

SPEAKeasy took place in two phases. Phase I proved the basic concept of software radio by demonstrating a reprogrammable piece of hardware capable of processing RF signals digitally. This hardware used four Texas Instruments TMS320C40 DSP processors as a signal-processing engine with RF signals digitised at the IF. Software was developed for the radio in the Ada language with some temporal and security sensitive elements programmed in assembly language. Phase II of the project expanded the programme and shifted the processing of RF signals from DSP processors to FPGAs. This gave more processing power for dealing with higher bandwidth radio signals, however the time required in re-programming the FPGAs was seen as a limiting factor of the design. Software implementations were also expanded to include fifteen different operating modes, allowing communication with a wide range of military waveforms.

While the technical aspects of SPEAKeasy were important in proving the basic concept of software radio, a more significant result of this work for the space as a whole was that it helped to consolidate many of the concepts being discussed at the time. The period from 1991 to 1995 saw the subject mature resulting in a better understanding of the capabilities, limitations and possibilities of the technology. This is apparent in the May 1995 IEEE Communications Magazine which contains a special issue on software radio. In this publication Mitola [Mitola95] discusses the 'Software Radio Architecture', which gave a high-level breakdown of the software radio system and how signal processing can be applied at each stage of the device.

An important aspect of Mitola's work on the software radio architecture (also discussed in more detail in [Mitola2000]) is that it addresses some of the key concepts that differentiate a software radio from its close relative, the programmable digital radio. A software radio places the A/D/A conversion as close as possible to the antenna allowing total programmability of RF bands, channel access modes, and channel modulation. However, these are often confused with software-controlled digital radios that allow the functions of the radio to be controlled via software. The two differ as a software-controlled digital radio although somewhat variable, is fixed in function, whereas a software radio can be redefined to do something entirely different.

In the mid-nineties, in particular in the U.S., interest in software radio began to emerge in commercial applications. This is seen in the 1995 issue of IEEE Communications Magazine in which most of the discussion is on applying software radio to mobile cellular communications, in particular PCS (PCS or Personal Communications Services is a term used in the U.S. to describe the family of mobile communications technologies including IS-54/IS-136 and IS-95). This is also reflected in other publications in the same magazine as Wepman [Wepman95] discusses A/D converter theory and Baines [Baines95] discusses the practicalities of developing real-time signal processing systems using available processors at the time. Again the focus of their work was largely on applying the software radio concept to mobile communications. It should be noted that most of the work at this time did not recommend well-defined practices, techniques or specific designs for software radio systems, instead much of this work presented discussions on existing communications theory and how it could be applied to the digital domain using software radio.

With the commercial industry mostly focused on applying software radio to mobile communications in an effort to reduce cost, some academics began to look at the software radio concept itself, with a view to exploring the new capabilities this technology provides. In particular the SpectrumWare group at MIT took a unique approach to the software radio concept [Tennenhouse95]. Instead of concentrating on hardware, they looked at the radio from the software perspective. They built radio systems using standard workstations (i.e. PCs containing GPPs) and off the shelf components. While their prototype system was quite limited in signal processing power and highly power inefficient, the advantage of their system was its flexibility [Bose99a]. They recognised that unlike other signal-processing approaches involving FPGAs and DSPs, the capabilities of their device would scale with Moore's Law.

Moore's Law [Moore65], a popular observation on semiconductor technology, states that the processing power of semiconductors doubles roughly every 18-24 months. The SpectrumWare team recognised that given time, Moore's law would improve hardware capabilities, therefore the major focus on system design should be software. (In June 2003 Vanu Inc., a commercialisation of the SpectrumWare group, carried out field trials of a GSM base station built using commodity PC servers powered by dual Xeon 2.8 GHz processors [Steinheider2003].) Others also started to look at the challenges involved in re-implementing existing radio standards in software. For example, Akos describes a software-based GPS receiver using GPPs [Akos97]. Although this system was incapable of processing these signals in real-time, it was only a matter of time before better processors emerged allowing real-time signal-processing of radio signals to take place.

Throughout academia, commercial and military interests there have been different opinions as to how to apply software radio technology or what the wide reaching implications of it are [Pereira2000, Tuttlebee99a]. In the U.S. the migration from analogue to digital communications

took place from 1996 and resulted in the deployment of multiple competing digital standards for PCS. This resulted in numerous incompatible networks across North America each providing different services. Software radio was seen as an important enabling technology for creating a terminal that allowed users to interoperate among these networks. In the U.S. software radio development has been concentrated on the mobile terminal (the user's mobile phone). The aim here has been to develop reprogrammable radio devices that can interoperate among the various U.S. standards. This emphasis on the terminal makes it difficult to take full advantage of software radio. Mobile terminals have stringent requirements on power and cost that limit the opportunities to maximise the use of software radio technology. In contrast, Europe adopted one standard, GSM (Global System for Mobile Communications), which has been in use since 1991. Europeans are therefore able to roam seamlessly throughout much of Europe and elsewhere. This has meant that there has been less urgency for software radio technology in Europe's 2G networks [Tuttlebee98, Tuttlebee99b].

These trends are also reflected in the organisations that promote software radio. The MMITS (Modular Multifunction Information Transfer System) Forum was established in the US in 1996 to promote multi-mode terminals capable of interoperating with AMPS and multiple PCS standards. In 1998 it changed its name to the 'SDR Forum' [SDRForum] to symbolise a widening of scope, yet its focus is still on delivering a solution to a lack of interoperability in the US. In Europe, without such an interoperability problem, the wider implications of software radio have been of more interest. This has elevated software radio research from a purely RF/hardware technique to an all-encompassing technology that impacts the whole network [Pereira2000]. Pereira [Pereira99, Pereira2001] suggests that software radio has implications across the whole networking environment. Drew [Drew2001] discusses similar arguments.

The interest in software radio in Europe prompted the European Commission to fund research into many aspects of this technology. Several research and development programmes funded projects in software radio, in particular ACTS (Advanced Communication Technologies and Services), Esprit, and IST (Information Society Technologies). The table below lists some of the most relevant projects carried out in Europe with a brief description. The breadth of scope in these projects demonstrates the impact software radio has had on the communications industry.

CAST	Configurable radio with Advanced Software Technology
Concentrated on adaptive radio access including a demonstration of key functional blocks in a software radio.	
DRIVE	Dynamic Radio for IP-services in Vehicular Environments
Interoperability of standards such as GSM, GPRS, UMTS, DAB, DVB-T with emphasis on multi-media delivery to vehicular applications.	
FIRST	Flexible Integrated Radio Systems Technology
Demonstrated the feasibility of multi-mode terminals for 2nd and 3rd generation mobile systems.	
MOBIVAS	Mobile Value Added Services
Looked at using software defined radio for delivering new value added services.	
PASTORAL	Platform and Software for Terminals: Operational Re-configurable
Using FPGAs to deliver a re-configurable, real-time platform for third generation mobile terminals.	
SLATS	Software Libraries for Advanced Terminal Solutions
Developing software libraries for GSM and W-CDMA on a DSP platform.	
SODERA	Reconfigurable Radio for SDR for 3rd Generation Mobile Terminals
A feasibility study into the best RF architecture suited for reconfigurable radio.	
SORT	Software Radio Technology
Looked at the basic hardware building blocks required to realise a software radio with a focus on GSM and W-CDMA.	
TRUST	Transparently Reconfigurable Ubiquitous Terminal
A wide-encompassing project primarily focused on the user's terminal but incorporating many investigations into system architecture and reconfigurability for multi-standard devices.	
WIND-FLEX	Wireless Indoor Flexible High Bitrate Modem Architecture
Investigated the development of a high bit rate radio system for indoor applications.	

Further information on these projects can be found at [Cordis].

Figure 2.1 – European Funded Projects Relating to Software Radio

Looking to the future there are also many other untapped applications of the technology both within mobile communications and the wider space of telecommunications in general. Starting with mobile communications, software radio has only been fully considered for 2G and 3G applications. There will however at some stage in the future be options other than these systems with 4G (fourth generation) possibly changing the common ways networks operate [O'Mahony2002]. The general perception for future generation systems being based on more intelligent flexible networks [Ribeiro2001]. There are differing viewpoints on what form such networks will take, but the general consensus is that devices will be capable of seamless, high-bandwidth networking anywhere in the world [Gazis2002]. While this type of capability is currently provided by cellular-based connectivity consisting of base stations and mobile terminals, newer research into areas such as ad hoc networking suggests that networks could form without such fixed infrastructure

[Johnson96, Perkins99, O'Mahony2001, Doyle2002c]. This concept originates from the efforts demonstrated by DARPA's PRNET [Jub87], except that today the focus is on mobility using low-cost lightweight radios. Research and development in this field is currently concentrated on developing more efficient protocols for larger and more mobile populations. Whatever form these networks take, it is evident that software radio will have an important role to play in providing flexible radio communication.

Outside of personal mobile communications and military applications, software radio has received less attention. Applications that use radio communication are only beginning to see the benefits of this technology (e.g. aeronautical applications [Cummings99a]). There are various reasons for this slow adoption. Firstly, the mobile communications industry has dominated research and development in radio technology since the inception of the first analogue cellular systems. Those with most to gain from the adoption of software radio have naturally pushed the development of the technology. Secondly, no universal or open platform exists for the development of software radio at present, only some proprietary offerings mainly targeted at mobile communications. In time as software radio matures and costs decrease, other industries will be able to employ software radio at a reasonable cost. Examples of these are emergency services, aeronautical, maritime, public safety, security, location-based services, broadcasting and transportation.

In general though, current research in the software radio space is still mostly concerned with delivering the hardware platforms that will power software radio systems of the future. This is where existing research and the work of this thesis start to differ. This thesis takes a similar approach to the SpectrumWare group in that it takes a more software-centric view to the development of software radio systems, but a unique approach in that it concentrates on reconfigurability and component-based software.

2.3 Terms and Definitions

With differing perspectives on software radio, no clear definition has emerged that satisfies all uses of the technology. Software radio will be employed in a variety of applications from 2G, 3G and 4G, and also to a host of other wireless applications. A consequence of these various approaches is a multitude of terms used to describe the technology. To the casual observer terms like software radio, software-defined radio, digital radio, reconfigurable radio and cognitive radio are interchangeable. However, while they do refer to the same underlying technology, each of these terms represents a different viewpoint when examined in the literature. This section discusses the meaning of these terms, discusses the definition of software radio and arrives at a term and definition suitable for this thesis.

In the majority of software radio literature analysed, no one has distinctly made a comparison between the terms 'Software Radio' and 'Software-Defined Radio (SDR)'. A cursory look through the literature suggests they are interchangeable, however taking the literature as a whole they are quite different. The word 'Defined' in SDR suggests that the software for a radio is defined once, a typical example being a radio system implemented in software using programmable ASICs, FPGAs and to a lesser extent DSP processors. While such a device would be reprogrammable, its end application is usually the same, for example a GSM base station, a D-AMPS mobile terminal or a DAB (Digital Audio Broadcasting) receiver. The use of SDR technology means these devices can be reprogrammed to correct bugs or perform minor upgrades, however the radio will not typically perform any function outside the original specification. The 'Software Radio' however tends to be a more general term covering a type of device that can be reprogrammed to perform many different types of applications. An example of this is the work by SpectrumWare; they used the term 'Software Radio' as their system allowed the creation of any number of applications using generic hardware [Bose99b]. Consequently the purity of software radios is also often referred to [Tuttlebee99b], with a 'Pure Software Radio' meaning a software radio approaching the capabilities of the 'Ideal Software Radio' [Mitola99e].

The term 'Digital Radio' has been used but it is a broad term and can be confusing in the context of software radio. Earlier radio designs often featured aspects of digital signal processing in the form of audio processing, filtering or by virtue of their use of digital modulation [Fines95]. The term becomes ambiguous however in the light of software-based radio systems, as SDRs and digital radios are built using the same fundamental technologies, i.e. ASICs and FPGAs. The difference lies in the viewpoint of those who design, build and sell these devices. The digital radio designer perceives the radio device as a completely digital hardware based device, designed and optimised for a particular application. The SDR-based designer views the radio system somewhat similarly in terms of hardware, but with a strong focus on reprogramming the software implementation.

The term 'Reconfigurable Radio' has emerged recently to emphasise the reconfigurable nature of software radio technology. The viewpoint here is that software radio should impact not just at the physical layer but should provide opportunities for new applications and services higher up in the protocol stack [Pereira99]. Similarly Ikonomou [Ikonomou99] addresses the issue by stating that software radio concepts now extend well beyond the simple reconfiguration of air interface parameters but extend through the network into service creation and application development. Chapter 4 discusses reconfigurability in more detail as the concept of the reconfigurable radio will be expanded on and used extensively in this thesis.

Finally, the concept of the 'Cognitive Radio' was introduced by Mitola [Mitola2000, Mitola99b]. The cognitive radio is similar to the reconfigurable radio concept but broader in scope as it referrs

to a more futuristic radio device. The cognitive radio is seen as an intelligent software radio device in that it can make informed decisions about its environment, perhaps in which modulation scheme or frequency allocation it uses. The cognitive radio augments the software radio through Radio Knowledge Representation Language and can manipulate the protocol stack to make better decisions about radio use. The cognitive radio concept was first developed with military applications in mind and is particularly suited for introducing advanced levels of security and associated military interests into software radio. Recently cognitive radio has started to emerge in discussions about new policies for spectrum management in the U.S. [FCC2002].

In addition to ambiguous terms, software radio also suffers from a multitude of contrasting perceptions about the exact definition of software radio. Mitola defines the software radio as follows:

> *'A software radio is a radio whose channel modulation waveforms are defined in software. That is, waveforms are generated as sampled digital signals, converted from digital to analog via a wideband DAC and then possibly upconverted from IF to RF. The receiver, similarly, employs a wideband Analog to Digital Converter (ADC) that captures all of the channels of the software radio node. The receiver then extracts, downconverts and demodulates the channel waveform using software on a general purpose processor.' [Mitola]*

Buracchini discusses the software radio concept and identifies the need for a common definition. He suggests that software radio should be defined as follows:

> *'Software radio is an emerging technology thought to build flexible radio systems, multiservice, multistandard, multiband, reconfigurable and reprogrammable by software' [Buracchini2000]*

Buracchini's definition is perhaps too simplistic to fully describe the software radio. The overall problem encountered throughout the literature is that the software radio concept brings a particular approach to building radio systems rather than a concrete system design. The viewpoint on the technology can thus be different depending on how someone wants to apply the software radio approach. Instead of one all-encompassing definition, the SDR Forum have classified radio systems into the following five tiers [SDRForum2]:

Tier 0 – Hardware Radio

'The radio is implemented using hardware components only and cannot be modified except through physical intervention.'

Tier 1 – Software Controlled Radio (SCR)

'Only the control functions of an SCR are implemented in software - thus only limited functions are changeable using software. Typically this extends to inter-connects, power levels etc. but not to frequency bands and/or modulation types etc.'

Tier 2 – Software Defined Radio (SDR)

'SDRs provide software control of a variety of modulation techniques, wide-band or narrow-band operation, communications security functions (such as hopping), and waveform requirements of current and evolving standards over a broad frequency range. The frequency bands covered may still be constrained at the front-end requiring a switch in the antenna system.'

Tier 3 – Ideal Software Radio (ISR)

'ISRs provide dramatic improvement over an SDR by eliminating the analog amplification or heterodyne mixing prior to digital-analog conversion. Programmability extends to the entire system with analog conversion only at the antenna, speaker and microphones.'

Tier 4 – Ultimate Software Radio (USR)

'USRs are defined for comparison purposes only. It accepts fully programmable traffic and control information and supports a broad range of frequencies, air-interfaces & applications software. It can switch from one air interface format to another in milliseconds, use GPS to track the users location, store money using smartcard technology, or provide video so that the user can watch a local broadcast station or receive a satellite transmission.'

While the categorisation of radio systems into different tiers is a good way of distinguishing different types of radio systems, the actual definitions of each type are not that useful. For example their definition of the SDR does not mention digital signal processing and describes 'software control', clearly this is in disagreement with the general consensus that SDRs implement all radio functionality using DSP. Also, their description of the 'Ultimate Software Radio' is quite narrow in focus as it seems to concentrate on the capabilities of a users terminal. Specific applications such as GPS are mentioned when in fact the ideal and ultimate software radios should be capable of

communicating with any other radio system regardless of frequency, modulation scheme or communications standard (the ideal software radio is discussed in more detail in Section 2.4.1.)

Lehr also recognises the confusion over software radio terms and suggests a more technical definition in that the term 'software radio' (as opposed to software-defined radio) should be reserved for systems that digitise the signal at the IF stage or further towards the antenna [Lehr2002]. This approach differentiates the software radio from radio systems that use software in their design but not for the processing of radio signals directly. While this is not a definitive statement it does raise an important issue as to where digital conversion should take place suggesting that this may be a way of categorising different types of radio systems.

Without concentrating on specific technologies or applications the U.S. Federal Communications Commission (FCC) define software radio as:

> *'A software defined radio is a radio that includes a transmitter in which the operating parameters of the transmitter, including the frequency range, modulation type or maximum radiated or conducted output power can be altered by making a change in software without making any hardware changes.' [FCC2001]*

The FCC definition represents the regulatory view of the software radio and this is demonstrated by its concentration on the transmitter, the goal of the regulator being to avoid interference. It is thus an incomplete and unsuitable definition for this discussion.

From these different perspectives it is clear that it is difficult to arrive at an exact definition that satisfies every viewpoint. To address the lack of clarity about software radio systems this thesis proposes the following definition that better defines the software radio concept:

> *'A software radio is a device that digitally converts radio signals in order to processes them using a software programme. Digitisation must occur close enough to the antenna to allow any function of the radio to be altered dynamically at runtime.'*

This definition is distinct from others for the following reasons:

- This definition requires that a software programme is used to process signals digitally. This differentiates it from other hardware-defined approaches.
- This definition requires digitisation as close as possible to the antenna to allow variability in all radio functions. This is included to ensure that the software radio

definition does not extend to devices that process radio signals digitally yet do not offer variablitiy in this function. It is this variability that makes the software radio concept unique.

- Finally, this definition uses the term 'runtime'. Runtime is a software concept making it clear that all functionality must be implemented in software rather than software-controlled hardware.

In discussing the more reconfigurable software radio, Chapter 4 also defines a definition of the 'Reconfigurable Radio'.

2.4 Hardware

Section 2.2 discussed the history of software radio without dwelling on specific technical principals or examples. This section presents all the technological knowledge required to understand the software radio concept. This section starts by looking at the 'Ideal Software Radio', often seen as the ultimate goal in radio system development. This is followed by a look at some practical limitations explaining why the ideal software radio is not possible at present. The remainder of the section looks at the various hardware technologies required to realise a software radio.

2.4.1 The Ideal Software Radio

Before reviewing specific technologies and techniques for software radio, it is useful to look at what most perceive as the ultimate goal in software radio technology, often termed the 'Ideal Software Radio' [Mitola92]. Figure 2.2 shows a diagram of the ideal software radio. Central to the ideal device is that the signal is digitally converted as close as possible to the antenna with all radio functionality implemented using DSP, with only a minimal essential amount of analogue hardware used.

In this scheme the radio would be able to transmit and receive extremely large bandwidths. In the receiver scenario this would allow the radio to digitise the entire RF band with DSP used for all receiver functionality including tuning, filtering and demodulation. In such a radio device it would be possible to receive on multiple frequencies simultaneously with each individual signal possibly using different bandwidths and modulation schemes.

Likewise, in the transmitter scenario, DSP software would be used to generate a wideband signal capable of transmitting anywhere in the RF band. It would be possible to simultaneously modulate multiple signals using different frequencies, bandwidths and modulation schemes. Effectively the analogue RF-front end of the ideal software radio would act as a physical gateway to the electromagnetic spectrum with all radio functionality implemented by DSP software.

The ideal software radio is a long-term goal. Although this goal may never be reached or warrant the investment to reach it, it does underpin the direction of software radio, i.e. moving DSP as close as possible to the antenna. Any developments in analogue electronics, DSP or radio engineering that bring us closer to that objective will deliver great opportunities for new types of wireless devices and hence new applications and services.

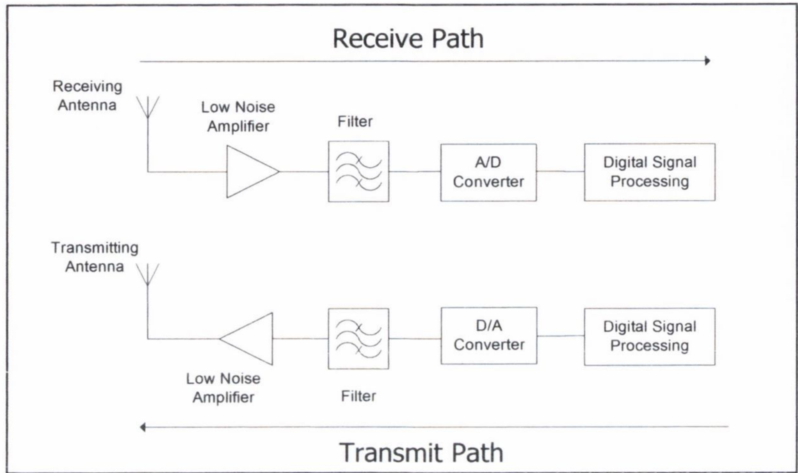


Figure 2.2 – The Ideal Software Radio

2.4.2 Practical Limitations

While there may be some existing technologies capable of bringing us closer to the ideal software radio (for example superconductors have been suggested for use in software radios [Semenov99, Brock2001, Brock2002, Fujimaki2001]) these are typically not feasible in cost or complexity and are not suitable replacements for existing radio hardware. To achieve the practical software radio with today's technology the designer must use conventional analogue techniques in combination with newer DSP hardware. The challenge is to strike a balance between analogue circuitry, digital conversion and DSP. The amount of each technology used will be dependent on the particular application but also on the capability and cost of the devices available. The most common approach taken today is to employ conventional analogue radio electronics for manipulation of higher frequencies with digitisation and digital signal processing occurring at lower frequencies [Hentschel99].

Figure 2.3 shows a more practical architecture for the software radio. In the receiver branch the signal of interest is mixed with a local oscillator that down converts the signal to a lower frequency. The signal is then digitised at this lower frequency. Likewise in the transmitter the signal is modulated via DSP and mixed with a local oscillator to up convert the signal to the transmit frequency. This design ensures that the demands on analogue to digital converters (ADCs) and

digital to analogue converters (DACs) are much less at these lower frequencies. Also, the DSP hardware can process data at a reasonable sample rate, much lower than the actual operating frequency of the radio.

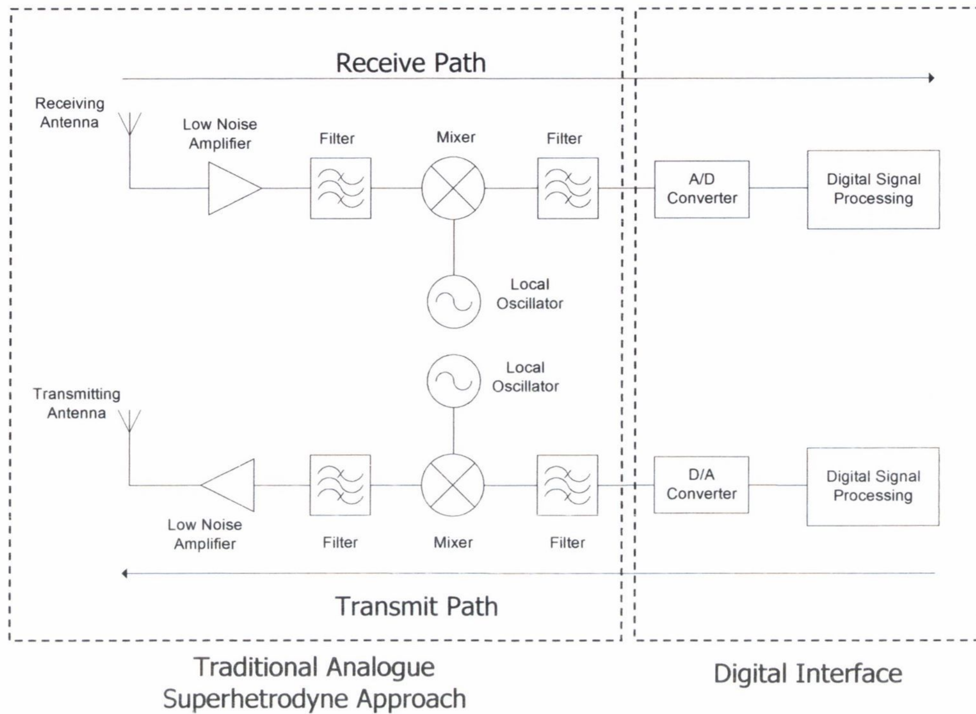


Figure 2.3 – A More Practical Software Radio Solution

In Figure 2.3 the left side of the diagram represents the analogue portion of the radio and this configuration is a typical superheterodyne architecture [Armstrong24]. In the receive path of this architecture the signal of interest is tuned, amplified and then down converted to a common frequency called the Intermediate Frequency (IF). The use of an IF allows a receiver to maintain selectivity and sensitivity across multiple receive frequencies. Selectivity refers to the receiver's ability to reject all frequencies except the frequency of interest. This will determine how well a receiver can receive a signal in the presence of other signals and noise. Sensitivity refers to how well a receiver can receive weak signals. Due to the physical nature of analogue components it is difficult to build a receiver that maintains selectivity and sensitivity across multiple frequencies and this difficulty increases with frequency. Converting a signal to the IF goes some way to solving these problems. The IF is a common frequency enabling analogue circuitry to be optimised for selectivity and sensitivity at this one frequency, ultimately reducing the amount of unwanted noise. In addition this frequency is usually lower than the actual receive frequency thereby simplifying analogue design.

A few IFs have become standard allowing manufacturers to produce components optimised for these frequencies, examples of which are 455kHz, 10.7MHz, 70Mhz, 140MHz. IFs are chosen

primarily to suit the bandwidth of the intended application but may also be influenced by factors such as component specifications and noise performance.

In the transmitter a similar approach is used. Signals can be modulated and then up-converted to a common IF frequency. This IF can then be translated and amplified for transmission at the required frequency. Many variations can be made on this architecture; some approaches use multiple mixing stages especially in high frequency applications. Also, different types of filtering can be used depending on the demands of the application. In summary, the central idea behind the superhetrodyne approach is the use of mixing and filtering stages to translate signals to more manageable frequencies and this approach can be applied to a vast number of applications in radio design.

The wide acceptance of the superheterodyne architecture has meant that it has become a popular choice in the migration towards software radio. In both receivers and transmitters, the approach has been to introduce digitisation at the IF frequency. As Figure 2.3 illustrates, this hybrid approach uses analogue circuitry for high frequency operations with DSP performed at lower frequencies. Moreover this strikes a balance that makes software radio more realisable and affordable using current technology.

Another architecture gaining more recognition in software radio applications is direct conversion, also known as Zero-IF. Its architecture is much the same as Figure 2.3 except for the frequencies used by the local oscillator. In the receiver this scheme means that the signal of interest is down converted directly to baseband bypassing any use of an IF frequency. Likewise in the transmitter a baseband signal is directly up-converted to the signal of interest. (Gu discusses Zero-IF in the context of software radio [Gu2002]). While this may seem advantageous, there are various problems associated with this technique, in fact these problems are the reason the superheterodyne approach dominates most designs. In particular, direct conversion results in a large DC offset in the signal which can make it difficult to recover the original signal. This DC offset is caused by a mismatch between analogue circuits which is temperature and time dependent introducing a variable error into the signal. Also, oscillator leakage, self-mixing, flicker noise and other inconsistencies can result in a corrupted signal [Patel2000]. This technique is however gaining popularity over the superheterodyne approach as it has better immunity to adjacent channel interference and is quite tolerant to variations in input power [Haruyama2001].

This discussion demonstrates that there is no single or best way to develop the hardware of the software radio system. Many areas of expertise from RF-hardware design to DSP processing devices must undergo significant development to meet the demands of the software radio and to move it towards the ideal software radio. For this reason there has been a variety of research and

development on tackling this problem. The software radio hardware research space can be divided into three areas of research split according to their function as shown in Figure 2.4; these are the RF front-end and antenna, digital conversion and DSP hardware. The following three sections discuss these important aspects of software radio and highlight relevant research in each one.

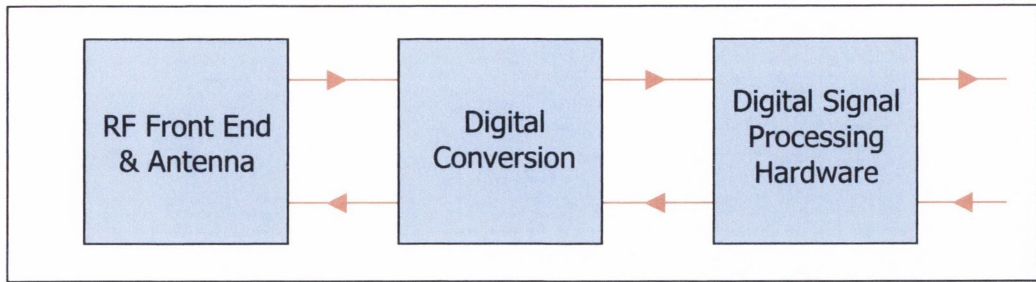


Figure 2.4 – Research and Development in Software Radio

2.4.3 Front-End Technologies

Even from the early days of software radio, developing the RF front-end has been recognised as a significant challenge [Mitola95]. The aim in software radio is to build a general-purpose front-end that acts as an interface between the antenna and DSP hardware. This is a significant change to existing RF front-ends. For example, existing multi-standard mobile phones (tri-band phones) contain three separate receiver chains for each standard [Tsurumi99]. Using the software radio approach these separate devices would be replaced by a single generic architecture.

The ideal device would allow the reception and transmission of arbitrary frequencies and bandwidths, but in practice hardware limitations mean that these parameters have to be constrained to a particular application. An example of this is the differing dynamic range requirements of GSM and W-CDMA systems. GSM has stringent requirements on signal to noise ratios (SNR) but the spread spectrum nature of W-CDMA means the SNR can be relaxed. Providing a common RF-front end for a device that can operate within both these standards would thus present a significant challenge with today's technology.

The requirement to digitise as close as possible to the antenna is not possible with current digital converter technology nor by the speed of current DSP processors or FPGAs. Cummings discusses this, pointing out that even if digital converters enabled digitisation at high frequencies such as 2 GHz for example, a DSP processor would have to operate at 2500GHz to process these signals [Cummings2002b]. This is clearly outside the capabilities of today's devices so currently practical RF front-ends must be customised for a particular application.

Hentschel [Hentschel99] discusses the tradeoffs associated with front-end design. Like most approaches he suggests that a limited band be selected out of the full band by means of analogue conversion and IF filtering. Beach [Beach2002] addresses the same issue and discusses the requirements and specifications of RF front-ends for software radio applications such as GSM 900, DCS 1800, DECT, UMTS, Bluetooth and Hyperlan/2. He discusses the tradeoffs associated with using different architectures concentrating on direct conversion and multiple conversion types. He suggests that with the current capabilities of technology a practical front-end for software radio is best achieved using a multiple conversion architecture (or superhetrodyne approach) as direct conversion can cause problems with wide bandwidth signals. Once conversion is completed, sample rate conversion can also be an issue [Abu2003, Hentschel2000].

In working towards more advanced front-ends a variety of research has presented designs and techniques for improving them. The challenge is to deal with problems such as linearity, image rejection, efficiency and power. As examples, MacLeod [MacLeod2001] recognises the importance of image filtering and amplifier linearity in front-end design. In similar discussions Kenington [Kenington2002] describes linearised transmitters and Morris [Morris98] describes the use of polynomial pre-distortion for improving amplifier linearity across a wide band of frequencies. Brinegar [Brinegar98] discusses the use of a flexible filter for a software radio application and for higher frequencies Streifinger [Streifinger2003] discusses front-end development at microwave frequencies above 10GHz. Some test beds have also been proposed and Schacherbauer [Schacherbauer2001] presents a wideband front-end capable of receiving a 5MHz bandwidth from 800MHz to 2200MHz. Mobile applications in particular pose significant challenges for front-ends. Kenington [Kenington2000] highlights the issue of power consumption in A/D converters for mobile software radio terminals. Cummings [Cummings2002] recognises that a 'sweet spot' must be found in front-end development for software radio so that power consumption, size and cost can be optimised. Further discussion on these issues can be found in [Wiesler2002, Cummings2002b, Salkintzis99].

Antennas are often overlooked when considering the RF front-end and in the area of software radio in general. Smart antennas represent the forefront of antenna research. Using digital beam forming, a smart antenna uses an array of antennas to increase the carrier-to-interference ratio in a wireless link [Razavilar99]. Smart antennas have become important in cellular applications in that they allow cell capacity to be increased by allowing the polar pattern of an antenna to be modified dynamically. Using these techniques two transmitters can transmit to the same receiver on the same frequency, with the receiver adjusting its polar pattern to match the incoming signal. Smart antennas and software radio are complementary technologies having the potential to greatly increase the flexibility of radio systems. Research combining these technologies includes

applications in direction finding [Kennedy95], evaluation test beds [Green2002] and their use in base stations for mobile communications [Pérez2001].

2.4.4 *Digital Conversion*

Digital converters, i.e. analogue to digital converters (ADCs) and digital to analogue converters (DACs) are very important aspects of any software radio as they form the boundary between the analogue and digital domain. Digital converter technology is being pushed to its limits by the requirements of software radio. Software radio requires converters that can not only sample at very high frequencies but can also offer a suitable dynamic range (i.e. bit-depth or word size) for representing signals [Wepman95].

The converter must provide an adequate signal-to-noise ratio (SNR), something that can be difficult to achieve with linearity problems and quantisation noise. Another important parameter is Spurious Free Dynamic Range (SFDR) that specifies the ratio in dB between the output of a converter and the peak spurious signal, an important parameter in judging whether a weak signal can be received in the presence of a strong one.

Providing a suitable SNR and SFDR exist, a converter's sample rate is the next important parameter. This ultimately determines the frequencies and bandwidths of signals that can be used by the system. Direct sampling can be performed by using a sampling rate at double the signal of interest. However, by exploiting the Nyquist theorem the sampling rate can be significantly reduced [Nyquist24]. The Nyquist theorem states that the sampling rate must be double the bandwidth of the signal, allowing converters to operate at a lower rate. This can also serve as a mechanism for down-conversion by simultaneously converting the IF frequency to baseband. It should be noted though that it is not just sufficient to lower the sample rate. The converter must have sufficient analogue input bandwidth and sample and hold circuitry to track and thus sample the higher frequency signal. The main point to note here is that the sampling rate, SNR and other parameters, and consequently the converter used, has an impact on the overall design of the software radio. According to the needs of the intended application the converter and overall architecture can be designed in creative ways meaning many different types of software radio architectures are possible.

Brannon [Brannon2002] gives a comprehensive overview of digital conversion for software radio. Concentrating on mobile communications he notes that the state of the art in digital converters lies at sampling rates in excess of 100MHz with typically a 14-bit word size, but that 16-bit devices sampling at 120MHz are in increasing demand. Devices with higher dynamic range are constantly being required to allow the recovery of weak transmitted signals in the presence of strong ones.

Brannon also discusses the emerging technology of Sigma-Delta converters. These can be used to create highly optimised integrated circuits combining many RF/IF functions into one device. Further discussion on these topics can be found in [Mitola99d, Fettweis2002, Abeysekera2002].

2.4.5 Digital Signal Processing Devices

Following digital conversion at the required frequency, radio signals exist in the digital domain. Depending on where digital conversion occurs in the radio and what the bandwidth of the signal is, the amount of data produced will vary. Some designs will digitise the signal at baseband thus the DSP will perform modulation and demodulation. Other designs will digitise close to the antenna requiring functions such as channelisation or direct down conversion to be performed in the digital domain. There are almost no limitations as to which radio functions can be implemented in the digital domain, but obviously the necessary processing power must be available and this becomes the limiting factor.

To perform DSP some form of semiconductor is required. A variety of devices have emerged to fill this role, in particular these devices have become the mainstream technologies of choice when implementing DSP systems: the ASIC (Application Specific Integrated Circuit), FPGA (Field Programmable Gate Array), DSP processor and more recently a range of reconfigurable processors.

ASIC: The ASIC (Application Specific Integrated Circuit) is a semiconductor device specially designed for a particular application [Smith97]. The ASIC cannot be reprogrammed and implements a one-off design that is often mass-produced. Because they are application specific, ASICs can be highly optimised for power and performance thus these devices are often used in applications requiring the best performance for example in graphics calculations or high-speed networking. They can also be more cost effective on a large scale as silicon area can be optimised. In conjunction with an A/D or D/A converter, an ASIC device can manipulate digitised signals and this combination is used in many existing applications in digital audio, graphics and control systems.

FPGA: Unlike the ASIC the FPGA (Field Programmable Gate Array) offers the advantage of re-programmability. The FPGA can be reprogrammed many times allowing the functionality of the device to be changed as required. An FPGA consists of general-purpose logic cells that can be reprogrammed and interconnected to form a particular application. Hardware description languages such as VHDL [IEEE2000] and Verilog [IEEE2001] are used to programme the FPGA although some newer languages such as System-C [System-C] and Handel-C [Chappell2002] offer more high-level programming constructs. The reprogrammable nature of these devices means they can be readily tested and also reprogrammed in the field to correct errors. Due to these advantages the

FPGA has become the device of choice where low-level digital hardware in conjunction with reprogrammable control is required. Although the FPGA in conjunction with design tools allow optimisation to take place, the FPGA cannot achieve the same performance or power efficiency of the ASIC. When used in conjunction with the A/D and D/A converters the FPGA can act as a general purpose device and has become a platform for many applications in such diverse areas as video manipulation, networking and telecommunications.

Also in the FPGA family is the FPAA (Field Programmable Analogue Array). Instead of interconnecting logic cells, the FPAA offers inter-connectable analogue blocks that can be used to create reprogrammable analogue circuits. This is useful in applications such as high-end filter design as in some cases it can be more cost efficient to implement such an algorithm using analogue components rather than using DSP.

Finally, another approach in the FPGA family is the hybrid FPGA-CPU. This is a device that contains both a CPU and an FPGA. The CPU can reprogram the FPGA thus it is possible to offload processing from the CPU to the FPGA for performance critical applications. Examples of hybrid FPGA-CPU systems are Virtex II Pro platform from Xilinx [Xilinx] and the Excalibur platform from Altera [Altera]. In the short term it is possible that type of device that will be useful for software radio systems, as this solution is quite cost effective and offers a good price versus performance ratio for commercial applications.

DSP Processor: A DSP processor is a processor specifically designed for signal processing applications [Lapsley97]. The DSP processor has emerged to fill a gap in the market for a device that offers a good price versus performance trade-off, and allows for the efficient, low-power implementation of signal-processing algorithms. DSP processors offer intrinsic support for multiply-accumulate and fixed point calculations which are common requirements for signal processing algorithms, but which are more difficult to implement using ASICs and FPGAs. Software for DSP processors has typically been developed using proprietary assembly languages and these are often different for each device family. More recently higher-level compilers for languages such as C and C++ have emerged from DSP processor vendors thus greatly simplifying development. DSP processors have become a mainstream popular choice for DSP applications but still lack the processing power of the FPGA and the ASIC.

Reconfigurable Processors: At the signal processing stage of the software radio the trend has been to move from ASIC designs to more reconfigurable devices such as the FPGA and DSP processor. There are however some new types of processors emerging specifically targeted at communications applications. These devices have emerged to address limitations of the FPGA and DSP processor, but are also specifically targeted at the requirements of 3G standards. The limitations they address

are the reconfigurable nature of FPGAs and DSPs. Although reconfigurable, they usually have to be taken offline to be reprogrammed. Newer chips such as the Chameleon Reconfigurable Communications Processor (RCP) offer a general-purpose architecture that can be reconfigured on the clock cycle introducing rapid reconfigurability [Burns2003]. The RCP is aimed at high capacity 3G base station applications. Another example is that of the Adaptive Computing Machine from QuickSilver [Watson2002, Master2002]. Like the RCP this device allows rapid reconfiguration but at a lower rate and is more suited to handset applications. This device allows the creation of custom data paths and uses specific techniques for improving the performance of multiplication and additions enabling DSP algorithms to be implemented more efficiently. A final example is the Sandblaster SB9600 Processor from Sandbridge Technologies [Glossner2003], another high-speed reprogrammable device catering for the needs of baseband processing for applications such as GPS, Bluetooth and WLAN. This device again demonstrates a move towards more high-level programming languages as it allows high-speed low-level programming to be achieved in C++ and Java.

From this discussion it is evident that there are many devices available for signal processing. For commercial applications, the one chosen will depend on the intended application and the cost of the device. In terms of software radio research the FPGA is quite popular and has served as a platform for a great deal of research into signal processing for software radio. As examples, Rice [Rice2001] shows how maximum likelihood phase synchronisation can be implemented with an FPGA. Seskar [Seskar99b] discusses FPGA-based architectures for interference cancellation in software radio. Ahlquist [Ahlquist99] discusses an FPGA approach to implementing error coding techniques. Abeysekera [Abeysekera2002] uses an FPGA to implement a sigma-delta architecture and Honda [Honda2001] discusses a technique for reducing the BER (Bit Error Rate) for software download using an FPGA. All these examples demonstrate existing techniques from radio technology migrating to the digital domain, with a focus on development with high speed FPGAs.

The ASIC offers full custom design for low-cost, high volume applications and thus has seen less interest in the software radio space as most research is of an experimental nature. The trend has been to use FPGAs and DSPs which both offer tools that make system design much easier. Many test beds have been based around the use of DSP processors, examples can be found in [Ellingson98, Patti99, Reichhart99, Dixon2001]. Power has been a particular concern for mobile applications and Gunn [Gunn99] addresses this issue discussing a low-power DSP subsystem. Other approaches are also evident and Kokozinski [Kokozinski2002] suggests that analogue and digital designs should be integrated on the same chip, something that may be possible with technologies such as the FPAA.

As discussed in Chapter 1 the work in this thesis is based on using GPPs. While other platforms have been considered, the GPP offers the best environment for demonstrating the concepts of reconfigurability. The next section discusses related work to this thesis and in particular highlights other work that has used the GPP as a platform.

2.5 Related Work

There has been limited work done on developing software radio systems on GPPs. As discussed in the previous section, most efforts have been concentrated on the development of FPGA and DSP designs. The work outlined in the following sections is a summary of related work that has either directly involves software radio on GPPs or closely related work that offers further insight into the topic.

2.5.1 SPECtRA and Variants

As briefly outlined in Chapter 1, the SpectrumWare group at M.I.T. was the first group to investigate the use of GPPs for software radio [Bose99a]. As part of this work the SpectrumWare group demonstrated SPECtRA (Signal-Processing Environment for Continuous Real-Time Applications), the first software architecture specifically designed for the development of software radio systems on GPPs using Linux as the operating system [Bose99b]. After some initial work SPECtRA was redesigned as PSpectra (Parallel SPECtRA), a system designed to achieve higher performance via multi-threading [Vasconcellos2000]. While somewhat internally different, the main objectives and characteristics of the SPECtRA and PSpectra environments are the same. Central to both designs was the aim of developing a toolkit for writing signal processing applications. The main characteristics of the Spectra designs was the use of a modular programming environment, infinite streams and the separation between in and out of band paths.

- Modular Programming Environment – Signal processing algorithms such as demodulators and encoders are coded into reusable modules by implementing C++ classes. Modules can be either *sources* for producing data, *sinks* for consuming data or *processing modules* for performing signal processing. Each module can have a different set of inputs and outputs. By connecting together modules it is possible to construct signal processing applications.
- Infinite Streams – Data flow between modules is accomplished via infinite streams. Each module 'sees' an infinite stream of data which means that a module can request the arbitrary number of samples it requires to perform processing. A data-pull model is used to move data through the system. Using this technique a sink starts the data flow by issuing a request for a number of samples. This request propagates through the system with each module calling on its downstream neighbour to produce the desired number of samples. This technique has a benefit

in allowing unnecessary samples to be discarded thereby reducing required processing power. The technique works by lazy evaluation in that samples are only generated when absolutely needed.

- In-band and out-of-band paths – The Spectra environments differentiate between code for performing signal processing functionality and code for controlling the general operation of the system (see Figure 2.5). In-band code consists of the modules themselves and connectors. Connectors provide the infinite stream abstraction and act as the binding between modules. A simple set of rules governs the connection of modules via connectors which are also C++ classes. Out-of-band code concerns the maintenance of the system and involves code for creating and modifying the topology of the system, communication among modules that does not involve signals (e.g. setting a sample rate), handling user interaction and monitoring system performance.

It should be noted that PSpectra contains extra functionality for creating more complex multithreaded designs. It also allows the building of meta-modules which encapsulate multiple modules into single modules.

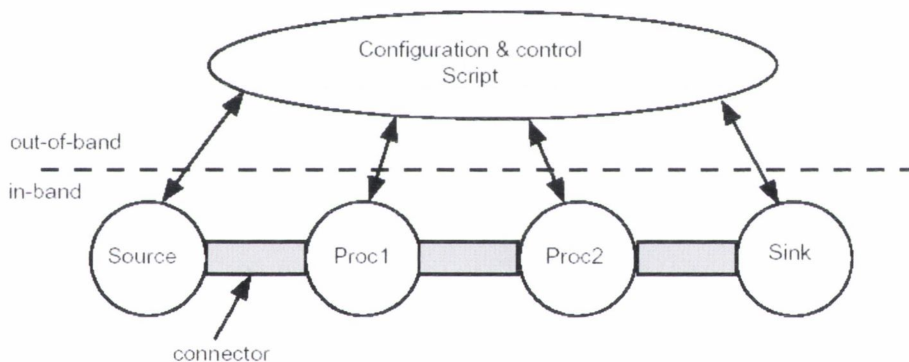


Figure 2.5 – SPECtRA In-Band and Out-of-Band Paths [Bose99b]

Using the Spectra libraries SpectrumWare demonstrated various software radio implementations including some analogue schemes, digital modulation and television receivers. Another contribution of the SpectrumWare group was their approach to algorithm development. With software radio systems built using commodity PCs with different resources available to an embedded system, a different approach could be used in developing many traditional communications algorithms. For example, Welborn discusses a technique for waveform synthesis for software radio, a technique particularly suited to GPP based radio systems which have access to large amounts of RAM (Random Access Memory) for storing pre-computed values [Welborn99a].

Following this work others have built on the PSpectra environment. The GNURadio project is an effort to build open-source software for software radio [GNU] that also runs on Linux. This group has used the PSpectra library as a basis for their project and have implemented a digital television

receiver among other schemes. EPSpectra [Kim2001a, Kim2001b] is another extension to PSpectra that uses the language Esterel to improve the real-time capabilities of PSpectra.

Following on from the work of SpectrumWare at MIT, Vanu Inc are a commercial venture working on the development of software for software radio systems [Chapin2002]. Vanu Inc. have successfully used GPPs to implement existing cellular standards such as GSM, thus demonstrating that GPPs are a practical hardware platform for software radio [Steinheider2003]. Their approach too has been built on the work of SpectrumWare but beyond similarities with SpectrumWare (SPECtRA, PSpectra, etc), only limited details are available on the specifics of how their systems work. One aspect that is documented is a language called Radio Description Language (RDL) [Chapin2001], a Java-based language for building software radio systems. This language allows high level programming and control of signal processing functions and forms the basis of their GSM implementation.

The difference between this thesis and the variety of work discussed in this section is that neither SPECtRA, PSpectra, GNURadio nor EPSpectra were built with reconfigurability as a focus, rather as systems to demonstrate concepts of software radio. In contrast, the IRIS system presented in this thesis is designed specifically for demonstrating and allowing experimentation with the concept of reconfigurability.

2.5.2 Software Communications Architecture

The U.S. military have been active in software radio research and development since the early days of the SPEAKeasy project as discussed in Section 2.2. Currently this effort is being led by the U.S. defence's Joint Program Office (JPO) under the Joint Tactical Radio System (JTRS) programme. A result of this work has been the development of an open standard for the development of software-based communications systems called the SCA (Software Communications Architecture) [JTRS2001, JTRS2002, Melby2002]. By creating an open standard that addresses both military and commercial applications the JPO hope that the development of SCA compliant commercial technologies will lead to reduced costs, increased interoperability and upgradeability.

Due to the diverse requirements across military and commercial applications, an open standard targeting one particular hardware platform would never be successful. Instead the SCA is an implementation independent standard that specifies a set of rules that constrain the design of communications systems. The SCA has been structured to [JTRS2001]:

- Provide for portability of applications software between different SCA implementations.
- Leverage commercial standards to reduce development cost.
- Reduce development time of new waveforms through the ability to reuse design modules.
- Build on evolving commercial frameworks and architectures.

The software structure of the SCA is based around an 'Operating Environment' which consists of a Core Framework (CF), CORBA middleware and a POSIX (Portable Operating System Interface) based operating system. The CF is an architecture that defines software interfaces that provide for the deployment, management, interconnection and intercommunication of software elements. The SCA inherently supports distributed computing be it in the form of inter-chip communication or across a network. CORBA (see Section 3.5.2) is used throughout the specification as an interoperability mechanism. CORBA acts as a 'logical software bus' allowing interconnection among the modules of the system. There has been some debate over the use of CORBA in such systems due to performance problems, however these problems have been shown to be tolerable in some circumstances [Bertrand2002]. The SCA relies on the use of a real-time POSIX-based operating system for providing base services such as multi-threading and memory management.

The SCA is a very comprehensive standard. It dictates interface definitions for every aspect of a communications system. Whereas software radio is usually concerned with the physical layer of the communications stack, the SCA is a broader specification as it specifies interfaces for physical, link and network layers of the communications stack. The software structure of the SCA is shown in Figure 2.6. Moving from left to right the diagram shows how an SCA compliant system is partitioned from the physical layer RF stage right to I/O applications. Each section of the system is viewed as a software component. CORBA is used for all interaction among components but some devices, for example FPGAs or DSPs, may not be capable of CORBA communication. For this reason 'Adapters' are specified which allow non-CORBA components to interact within the system. The logical software bus is shown which allows intercommunication among modules and the base services of operating system and hardware elements are shown at the bottom of the diagram.

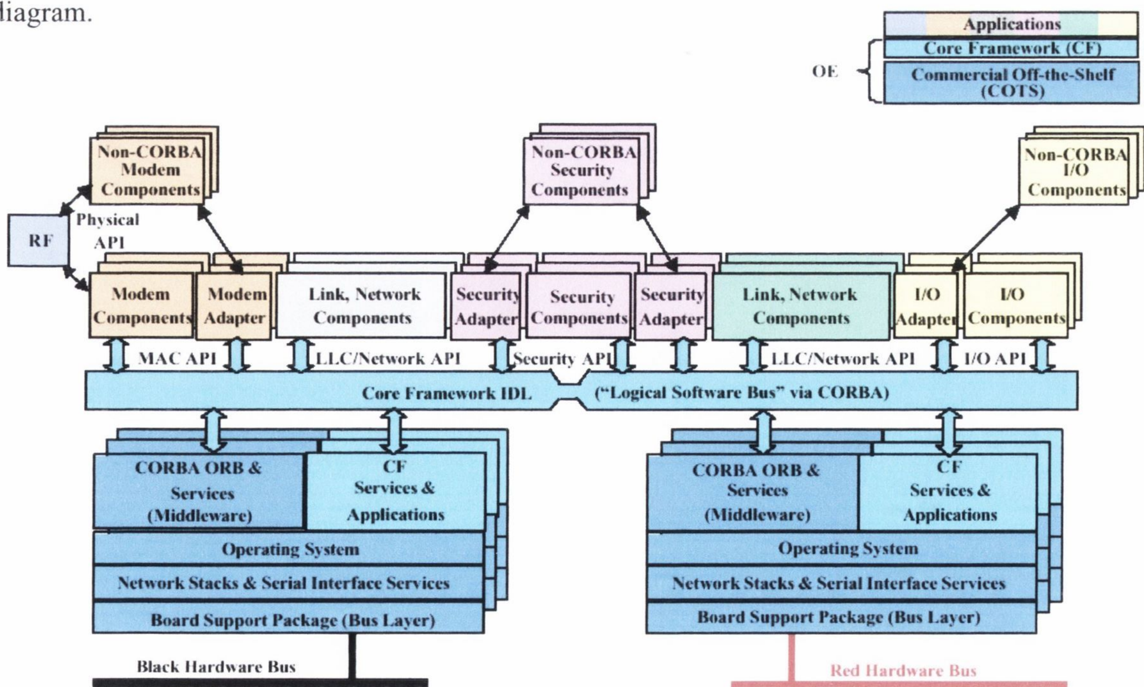


Figure 2.6 – Software Structure of the SCA [JTRS2002]

Communications applications built using the SCA are based around the use of 'Resources'. A resource is a single abstraction for many of the software components in the system. Examples include a *LinkResource* for components involving link layer processing and devices such as *ModemDevice*, *I/ODevice* and *SecurityDevice* all of which can be used in the same way. The internal implementation of resources is application dependent and hidden, thus Resources provide a black box abstraction for reuse.

Although not specifically designed for reconfigurability, the SCA is of interest in this thesis as it uses an object-oriented and component-based approach to the problem of developing software for software radio. The SCA could be viewed as a component framework with each of its resources defined as a different software component. Beyond this analogy though the system features limited reconfigurability. The SCA is a good example of a 'Software Defined Radio' as opposed to 'Software Radio' (as per the discussion in Section 2.3). Once a radio standard is implemented on this platform it is rigidly fixed and only limited reconfigurability can take place within the sub-elements of the system. The system as a whole cannot be reconfigured to do something completely different. For this reason it is unsuitable for demonstrating the concepts of reconfigurability presented in this thesis.

2.5.3 DSP Design Tools

While not built directly for software radio, it is useful to contrast the work in this thesis against some DSP design tools.

Ptolemy is a software project from Berkley MIT that provides an environment for modelling, simulation and design of signal processing algorithms [Buck94]. Central to Ptolemy is the concept of models of computation, a facility that provides a highly expressive environment for representing different types of signal-based systems. The reason Ptolemy is somewhat related to this work is that the system presented in this thesis also provides an environment for developing signal-based systems. However, there are some distinct differences between Ptolemy and the work in this thesis.

Firstly, Ptolemy is a tool for modelling and simulation. IRIS is not a design tool but a component framework for developing real reconfigurable radio systems. Although Ptolemy can potentially generate source code for a variety of platforms the way in which it views its targets is quite different to IRIS. IRIS reuses blocks of signal processing logic as software components, whereas the blocks existing in Ptolemy exist at design-time only. These blocks are eventually collapsed down to an implementation that is fixed in function. In contrast, the IRIS system is designed so that the actual system developed can constantly reconfigure. At its core IRIS supports application,

structural and parametric reconfiguration, concepts that do not enter into the Ptolemy design paradigm.

SPW (Signal Processing Worksystem) from Cadence for example is a tool for capturing, simulating and verifying DSP designs for FPGAs. This tool provides the full tool flow for developing SoC (System on Chip) and FPGA designs. It provides a visual block-based user interface for constructing any type of FPGA design. This type of tool is extremely different in function to the IRIS system. As with Ptolemy, SPW is primarily a design tool with integrated simulation and testing, whereas IRIS is a component framework for building real reconfigurable radio systems. The IRIS component framework hosts radio applications much in the same way an operating system hosts a user's application. For this reason the tool flow approach of SPW is quite different.

Other tools that fall into this category are Matlab and Simulink [Mathworks]. Matlab and Simulink are simulation tools for modelling, simulation and development of signal processing algorithms. Both of these are different to the work presented in this thesis as they are design tools whereas the IRIS system is a component framework for implementing real reconfigurable radio systems. These systems have not been built specifically for developing reconfigurable radio systems. These tools and other approaches are discussed further in Section 2.5.

2.5.4 Other Approaches

Using a somewhat different approach to SPECtRA and the SCA is the work of Srikanteswara [Srikanteswara2000a, Srikanteswara2000b]. Srikanteswara presents a software radio architecture designed for reconfigurability using FPGAs. The system uses an FPGA which can be reprogrammed at runtime. Signal processing functionality is implemented in processing modules that can be swapped and reconfigured at runtime to dynamically change the functionality of the radio. Functionality is divided into layers and processing of data occurs in a similar way to a communication stack. Data propagates through the system using stream-based processing which uses a common bus for transferring data and control information. Self-steering streams weave through layers, each of which performs processing on the data. Data is thus transferred as packetised data and can contain either control information or signal data. The packet can contain configuration information on how the data should be processed via 'embedded variables'. Other 'non-embedded variables' can provide variability independently of the data stream.

Although completely hardware based, this system again demonstrates the common approach and advantage of using separate processing modules for implementing radio systems. Just like the PSpectra approach, this system allows different radio configurations to be created by interconnecting generic processing modules. Also, modules can be swapped and reconfigured at

runtime allowing the system to dynamically adapt. It should be noted that beyond the conceptual view, this type of system works in a completely different way to a GPP based system. GPPs have a different architecture in that programmes are loaded from RAM and executed sequentially via an instruction set. On the other hand the interconnections within an FPGA are physically re-adjusted each time a new configuration is reprogrammed. Thus, an FPGA cannot achieve the same levels of reconfigurability possible with the GPP as each new iteration in configuration would require a change in physical hardware. It should be noted that if the reprogrammable features of the FPGA are not used and it is programmed once to act as a GPP, it too would feature the same levels of reconfigurability. However, this would be a more costly process than using readily available hardware as discussed in Section 1.4.

In the evolution towards software radio it is not surprising to find a wide body of research based on migrating existing radio techniques into the digital domain. Many existing techniques from the analogue domain require a new approach for digital implementation. As examples, Ikemoto [Ikemoto2002] discusses the use of adaptive channel coding schemes using finite state machines, these being a concept from the digital domain. Similarly Harris [Harris2001] discusses the development of multi-rate digital filters for symbol timing and Zhao [Zhao2002] discusses the use of an existing scheme, GMSK (Gaussian Minimal Shift Keying), in a receiver implementation. Also Yang [Yang2002] describes techniques for implementing broadband frequency hopping multi-carrier systems in the context of software radio and Thara [Thara2002] discusses the use of Turbo coding. All these cases demonstrate existing radio techniques being migrated to more software radio orientated, DSP-based applications.

A significant focus for software radio has been its capability in allowing software download or OTAR (Over The Air Reconfiguration). The basic idea here is that a software radio can reconfigure itself by downloading new software from a remote location. There are various issues associated with this including security [Mehta2001, Michael2002] and mode switching [Cummings99c]. Chapter 7 (Section 7.2) will discuss OTAR in more detail through a case study.

An emerging area of research is the concept of waveform description languages (WDL), languages capable of providing a portable mechanism for describing software-defined waveforms. Willink [Willink2002] discusses the composition of such languages and discusses the design of such a language for describing waveforms in a platform independent way. The main aim of this approach is to provide a language that facilitates the expression of software radio concepts without the overhead associated with other general-purpose functional or descriptive languages. Chapin also discusses a similar approach with RDL (Radio Description Language) [Chapin2001]. This language is Java based and operates on general-purpose processors allowing the control of signal processing functions and higher protocol level functionality.

2.6 Summary

This chapter has presented a comprehensive overview of software radio technology. The history of software radio and the discussion of terms and definitions demonstrate that the scope of the technology is large and that it impacts radio system design in many different ways. The discussion of hardware demonstrated both the ideal and practical software radio, and presented an overview of the hardware required to realise a practical software radio system within today's technological capability. The final section narrowed the focus towards the work in this thesis by discussing relevant research relating to this thesis. From this discussion it is evident that research into software for software radio systems is varied. The work discussed demonstrates a variety of approaches and it is clear that more work is required to consolidate many of the ideas being discussed. In addressing such needs, this thesis makes its contribution to this area by now focusing on the development of software for reconfigurable radio systems.

3 Software Engineering

3.1 Introduction

This chapter presents an overview of software engineering and provides background information on a range of principles and techniques for developing good quality software systems. Later chapters will use these principles in discussing the development of highly reconfigurable radio systems.

Software engineering has evolved as a standalone discipline of engineering as software needs to be properly engineered to produce reliable software systems. The problem of complexity was widely recognised in the early days of software technology. In the late 1960s attempts to develop large-scale software systems resulted in seriously flawed systems that often contained errors that were difficult to fix. Frequently these systems did not effectively solve the problem being addressed. These issues became known as the 'software crisis' [Dijkstra72].

At the time, the popular belief was that software problems are intrinsically complex (such as mathematical calculations), but this observation was incorrect and in reality most software systems are complex due to the vast number of details that must be dealt with. It is the systematic management and abstraction of these details that forms the basis of software engineering. Three decades later there is a better understanding of why software can be complex and significant work has been carried out on developing methodologies, principles and techniques for developing large robust software systems.

The practices of software engineering are not typically applied to software radio. Most of the time software development for a software radio system is done using hardware description languages for FPGAs, or C implementations for DSP processors. In this environment the application of software engineering techniques is sacrificed in favour of optimisation, i.e. performance (faster execution speed) and smaller code size. Faster performance translates to cheaper processors, as more efficient code requires less processing power. Smaller code size translates to cheaper devices, as less storage memory or RAM is required. Optimisation usually means breaking encapsulation, reducing code reuse and reducing the maintainability of the code, thus abandoning many of the principles that underline modern software construction. However, to meet the demands of increasingly complex radio standards more and more code is being written. Without proper use of software engineering

techniques, software radio has the potential to fall foul to its own 'software crisis', ultimately resulting in unstable and unreliable radio systems.

For this reason it is important to apply the principles of software engineering in software radio and this has been a major focus of this thesis. While modern software engineering dictates practices in everything from managing people to project coordination, it is the technical practices of software engineering that are more relevant in this work. Object-oriented techniques and component-based software are the techniques used in tackling the problems of complex software development. By applying these techniques to software radio, software can move from being complex and error-prone to being manageable, reliable and stable.

In addition, another advantage of using software engineering techniques is that software can become better structured. Using these techniques software can be built that is reusable, flexible and adaptable. A *reusable* piece of software is constructed in such a way that it can be easily reused by others thus eliminating re-implementation. A *flexible* piece of software offers variability in how it performs its function and provides a simple mechanism for doing so. An *adaptable* piece of software can be used in scenarios that were unforeseen at design time. This work attempts to build software for software radio that exhibits these traits.

The remainder of this chapter presents an overview of software engineering techniques. It begins by briefly discussing object-oriented software but moves on to concentrate on component-based software. Existing component technologies are discussed, as the system presented later in this thesis is a component-based software system for software radio. The overall purpose of this chapter is to provide background information on the software approach employed in this thesis. The reader familiar with software principles may wish to skip to Section 3.6, which summarises the main points of this chapter.

3.2 Object-Oriented Software

3.2.1 Overview

Before object orientation, the dominating approach to programming was the functional decomposition approach of procedural languages such as C and Fortran. Using functional decomposition a problem is approached from the top-down with each problem broken down into sub-problems. Functions consisting of algorithms are written to solve each problem and a hierarchy of these functions form the resulting solution. While functional decomposition is a fundamental technique of software design, when developing a software system of considerable scale this structured approach can run into difficulty. Programmes can become difficult to maintain and extend, overall reducing the quality of software produced.

The object-oriented approach works differently to functional decomposition. Instead of breaking down problems into functions and algorithms, a problem is addressed by identifying objects that play a role in the system. The system is built by defining a set of objects and defining relationships between them. Object-oriented programming has its roots in the languages Simula-67 [Dahl70] and Smalltalk-80 [Goldberg83]. These languages were the among the first to use the concept of an object. In its purest sense an object is something that has state and behaviour. Objects communicate with each other through message passing which may alter the state of the object. The behaviour an object provides is defined by its interface or set of commands it provides. Object-oriented languages allow a system to be built around concepts and constructs of the real world as opposed to concepts intrinsic to computers such as algorithms and hardware. Object-orientation is thus a way of modelling and viewing software systems with the term object-oriented design (OOD) used to describe the design process required to develop such software.

In the practice of OOD, two design techniques are of particular importance; UML and design patterns. In designing the object-oriented system some method of describing object-oriented designs is required. UML (Unified Modelling Language) has emerged to fill this role [OMG2002]. UML is a graphical language used for defining the relationships between objects and use-cases of a software design. UML will be used later in this thesis to help explain software designs. Design patterns are used in object-oriented programming to capture the solutions of recurring problems [Gamma95]. Each pattern describes the solution to a problem that reoccurs frequently. By recognising reoccurring problems and applying the appropriate patterns, a software designer can apply tried and trusted principles to a design thereby creating more robust software. These techniques provide the designer with mechanisms for communicating software designs.

Closely related and often confused, object-oriented programming (OOP) is a different practice to OOD. Whereas OOD is about modelling and viewing a software system, OOP is about how to actually implement an object-oriented design in software. The two practices are distinct because each programming language implements object-orientation in different ways. While an exhaustive discussion of object-oriented languages will not be presented here, the main concepts of OOP exist through encapsulation, inheritance, interfaces and polymorphism.

Encapsulation provides modularisation in a software system. An object uses encapsulation to shield its internal data from modification by another object. Instead each object exposes an interface by which other objects access its data or request it to perform some operation. Using this mechanism the internal implementation of behaviour and the data itself are essentially hidden.

Inheritance allows one object to inherit the characteristics of another object. By doing so an object can reuse the functionality of another, thus it is possible to create hierarchies of objects. While the

concept of inheritance itself is simple, the implementation and use of inheritance is a subject of much debate. Problems such as the fragile base class or diamond dependencies can occur if inheritance is not used with care.

An *interface* defines the interaction of objects. An interface specifies the operations an object will support and thus offers a way to express the functionality of an object independently of an implementation. The interface is thus an important tool during both OOD and OOP. In OOD it allows designers to express the functionality of a software object without having to worry about how it should be implemented. During OOP interfaces can be used to enforce a design, as code will fail to compile unless the implementation adheres to a set of interfaces.

Polymorphism is the ability of an object to appear in multiple forms, depending on context. For objects to be polymorphic they must inherit from the same base class and implement the same functionality. A typical scenario involves the use of an abstract base class to represent some entity, for example a 'Vehicle'. Other classes can inherit from this base class to create different types of objects, e.g. 'Aeroplane' or 'Car'. A polymorphic language allows the programmer to interact with any sub-classed 'Vehicle' object without knowing whether it is an 'Aeroplane' or 'Car'. When used in conjunction with interfaces, polymorphism can offer a powerful construct for hiding the implementation of an object. Using polymorphism many different objects may expose the same interface with the details of each object hidden in the implementation.

Objects are commonly defined via the *class* construct. The class is a construct that allows the specification of an object via the data it stores and the methods it exposes. Central to the class is encapsulation, or data hiding. The class offers a set of access modifiers that allow access to data and methods to be restricted. Using these modifiers unnecessary internals of a class can be hidden from external clients. Whereas a class defines the blueprint for an object, an instance is a manifestation of that object. Thus from one class definition, multiple instances of an object can be created. A class is said to be abstract if it does not implement all the methods specified in its class definition.

3.2.2 Object-Orientation for Software Reuse

Object-orientation inherently supports software reuse. OOP allows the construction of software objects that can be reused by others to solve similar problems. The advantages of software reuse in OOP are not often apparent in small software systems. Often, it can seem unnecessary to use classes, especially when programming a small system in which it is known that classes will never be reused. Effective reuse only becomes apparent in large software systems. Most large systems rely on the definition of basic objects that are reused extensively throughout the system. Software

construction at this scale can leverage object reuse to great effect in reducing the amount of code used, and also increasing the simplicity and thus maintainability of the system.

For software to be effectively reused it is not sufficient to just place code into a class, there are many other factors that determine how reusable an object really is. While some of these factors may be loosely defined by elements of taste or aesthetics, others are well-recognised principles and have been formalised in the literature. Of the latter we consider here the concepts of granularity, coupling and cohesion.

Granularity

There is an inverse relationship between software granularity and software reuse. Objects are designed to represent and solve problems for a particular domain (or application). The larger a software object is, the more domain-specific the object will become. A software object is not likely to be reused if it caters too specifically to a particular domain or application. Conversely, a fine-grained object can potentially be reused more because it provides limited functionality and is not so domain-specific. Choosing the correct context and granularity for a software object will thus determine how often the software can be reused.

The effect of granularity is evident in the class libraries available for programming with languages such as Java. The Java class library offers a wide range of classes ranging from domain-independent to domain-specific. Fine-grained classes representing primitive types such as strings and integers are used extensively throughout the whole class framework, whereas larger more domain-specific classes can only be reused in that domain's context, for example graphics or networking classes.

Cohesion

Cohesion is the measure of the level of logical relationships in a piece of software. Balen [Balen2000] defines cohesion to be 'a measure of the level of logical relationships between methods of a class and also a measure of logical relationships among sub-systems'. Yourdon [Yourdon79] has defined cohesion using various terms, namely coincidental, logical, temporal, procedural, communicational, sequential and functional. These definitions identify the characteristics of cohesive software modules at various levels of cohesion from weak to strong.

In general it is advantageous to strive for highly cohesive software objects. In a highly cohesive software object (or functionally cohesive module according to Yourdon and Constantine) the elements the object expose will be related in that they all contribute to solve a common problem. An object with weak cohesion haphazardly associates elements that share no common purpose. While the obvious approach to software construction always suggests using cohesive elements, the practicalities of design make this difficult to achieve. A cohesive object is more likely to be reused

than an object with weak cohesion, as the cohesive object will be logically structured as a unit that addresses a particular problem.

Coupling

Whereas cohesion is a measure of the relationship between software elements, coupling measures the dependencies among elements. VanVliet [VanVliet2000] defines coupling to be 'a measure of the strength of the inter-module connections'. Similar to cohesion, various terms have been used to define various levels of coupling, namely content, common, external, control, stamp and data ranging from tightest to loosest. Tightly coupled objects are objects that require a lot of dependencies on other objects to function. A loosely coupled object has weak dependencies in that it can function independently of other objects.

In general loosely coupled objects are preferred over tightly coupled ones. A loosely coupled object can offer better software reuse, as it can be adapted for use in many different scenarios without having to maintain inter-object dependencies. In practice creating loosely coupled objects is difficult because functionality is gained by bringing together objects to form new objects. By bringing together objects new dependencies are formed which tightens coupling. One common solution is to re-implement the functionality of other objects to avoid dependencies but this goes against software reuse and can increase the level of cohesion.

3.3 The Principles of Software Components

Following on from object-oriented techniques, the software component is a more comprehensive unit of abstraction that also attempts to address the problems of complexity and reuse in software systems. This section defines software components, their difference to objects and how they can be constructed.

3.3.1 Defining the Software Component

Whereas object-orientation is well understood as a methodology for designing and programming software, the concept of the software component is a newer concept and thus has various meanings throughout the literature. Various definitions have been proposed:

> *'A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties' [Szyperski2002]*

> *'A software component is a static abstraction with plugs' [Nierstrasz95]*

'Reusable software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation and a defined reuse status.' [Sametinger97]

'A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction' [Booch87]

While definitions vary in scope, the essence of the software component is that of a reusable piece of software with well-defined functionality that exposes well-defined contractual interfaces. It is interesting to note that Sametinger includes documentation in the definition of the software component and that Szyperski includes reference to use of components by third parties. This suggests a higher-level construct than the object, in that a software component is not simply a software implementation but addresses a wider range of issues in how its functionality is described and how the component is used and made available.

In Szyperski's comprehensive text on component software [Szyperski2002] he identifies the software component via the following characteristic properties:

1. *A component is a unit of independent deployment*
   For a component to be independently deployable, the component must be self-contained. It must be packaged into an independent unit and must be well separated from its environment and from other components.
2. *A component is a unit of third-party composition*
   Components are primarily designed, implemented, tested, and subsequently used by people who do not have the desire or expertise to write the software for themselves. For these reasons, people who use the component should not be (or should not have to be) aware of any of the construction details of the component.
3. *A component has no externally observable state*
   Components often represent heavyweight pieces of functionality in a system. Often there will only be one instance of a component in a process. A component should have no persistent state, i.e. the component should be identical to copies of itself. This is in contrast to a software object, in which its identity is defined by its state.

Again, these properties suggest that the software component is a very different construct to the object. For example, Szyperski suggests that there should be only one instance of a component whereas objects are specifically designed so that they can be instantiated multiple times. Overall,

object-orientation is a technique for building the internals of a piece of software. Component software is about how this implementation is packaged and deployed for use by others.

3.3.2 Objects vs. Components

Objects and components are often confused. This section discusses the difference between objects and components, and how they facilitate different methods of reusing software.

The differences between objects and components can be summarised as follows:

- In OOP, every object is constructed in a different way and specific knowledge of the object is required to avail of its functionality. In contrast, in component technology components that implement completely different functionality often use the same interface. In this way components are much more standardised ways of exposing functionality.
- Objects are often language and platform specific making binary interoperation difficult. Many component standards are built specifically for language and platform independence allowing many different types of components to interact.
- In objects, dependencies may have to be sourced so that the component will function. A component usually contains everything it requires to function reducing the amount of dependencies it requires.
- In object-oriented languages an object is often statically linked into an application requiring the application to be rebuilt if significant changes are made to the object. In component-based programming components are mostly dynamically linked and interchangeable allowing different components to be used without recompilation. This approach can be used for software maintenance and upgrading, or as a technique to enforce a contract between different pieces of software.

The component concept embodies a particular viewpoint on how software should be reused. In other contexts reuse can be as simple as the copying of source code or using a set of library routines. In OOP, objects are reused via instantiation or inheritance. Even though these facilities do work, OOP and other methods often fail in achieving effective reuse of code.

For example, objects are often seen as bad elements of reuse. While a software component can be built using an object-orientated language, object-orientation itself says nothing about how to package the software into a reusable unit. Also, it says nothing about how the component's interface will be exposed to the outside world, or how it interacts with other types of components. The concept of the software component goes some way in solving the problems of reuse. Software components satisfy the need in software engineering to be able to package and subsequently reuse a piece of software.

A useful characteristic in discussing reuse is the level of visibility exposed by software. Visibility in this context is the visibility a programmer has of the internals of a piece of software. This visibility is often referred to in terms of 'white box' or 'black box' abstractions. In a black box abstraction, the programmer using the software only has knowledge of an interface and its specification. All other details are hidden. A white box abstraction may still enforce the encapsulation (hiding) of functionality but will allow the functionality to be extended or modified via mechanisms such as inheritance. Other terms are also used to refer to levels of visibility that lie between white and black abstractions, for example a 'glass box' abstraction allows the internals of a piece of software to be viewed without allowing modification of functionality.

Software components are seen as black box units of abstraction whereas objects can be viewed as white box abstractions. Components do not reveal their internal functionality and operate via their interface only. Objects on the other hand are white box because they allow their functionality to be changed via inheritance. White box abstractions and thus objects in general are seen as bad elements of reuse. The problem lies in OOP's use of inheritance for software reuse. Inheritance allows changes to be made to the internals of an object and this introduces the possibility that programmers might reinterpret or simply break an object. For this reason software components are designed to be black box units in that they do not typically allow extensibility though inheritance.

Another important issue when considering reuse is granularity. Granularity for software components is similar to the concept of granularity for objects (see Section 3.2.2). Whereas objects are seen as finer grained elements reused in the construction of software, components are larger entities that enable software reuse at a much larger granularity, perhaps reuse of a complete software system.

3.3.3 Constructing Components

A component is constructed and hence defined via:

- The interface it exposes.
- The dependencies it requires to operate.
- The meta-data it exposes.
- How it is deployed.

Each of these is discussed in the following sections.

Interfaces

An important characteristic of a software component is how it defines its interfaces. The interface in the context of the component is quite different to the interface defined by OOP. In OOP the interface is defined as part of the programming language. Whereas the interface in OOP defines its

relation to other objects within the context of a particular language, the interface of the component defines its interaction with other components that may be implemented in different languages and may even exist across networks on different platforms. Therefore in the context of components, interfaces define a component's interaction with the outside world.

The interface is often seen as a contract, the analogy being that breaking the contract, or the way in which components interact, is equivalent to breaking the software. Thus, all components must use a well-defined interface. Interfaces have an important role to play in quality, as overall software quality can depend on how well interfaces are defined and how well both clients and providers adhere to these interfaces.

Dependencies

In OOP new objects are created by bringing together existing ones. A dependency identifies the relationship between objects brought together in this way. If object A is constructed by combining objects B and C, then object A has a natural dependency on B and C; in other words, object A cannot exist without B and C. It follows that to reuse a piece of object-oriented software, knowledge of its dependencies are required and these dependencies must be present for the software to work. Dependencies are difficult to avoid, but one of the aims of component-based software is to shield programmers from having to deal with the particular dependencies of a component. This is reiterated by Szyperski's definition of the component being 'a unit of independent deployment'. Thus, a self-contained unit should contain all dependencies necessary for its correct function.

Meta-Data

Meta-data is information about information. In terms of component technology meta-data is an important facility for self-description. Using meta-data a component can export any information about the services it offers, the interfaces it exports, what its dependencies are, etc. Information about a component can by dynamically queried and this can be done programmatically without human intervention.

Meta-data is a useful facility in loosening the coupling between software elements. Instead of having to statically link pieces of code together during development, meta-data can allow dynamic discovery and the use of new components that become available at runtime. This allows a system to be extensible in that it can load new functionality without recompilation.

Object-oriented languages such as Java and C# support meta-data via reflection. Reflection allows an external client to query all information about a compiled class. C# has more advanced support for meta-data in that it supports attributes. Attributes allow arbitrary pieces of meta-data to be

included alongside compiled code. Components may support their own methods for exposing meta-data or they can use specific language facilities such as reflection.

Deployment

An important aspect of a component technology is to define how a component is deployed, as this will ultimately determine how the component is made available and used. Deployment in this context refers to how the component is integrated into new systems and the mechanisms involved in availing of a component's functionality. There are various techniques for deploying a component. Firstly, there are binary compatibility standards. These types of components can be constructed in any language, but to expose functionality it must expose its functionality using a particular binary format. Secondly, some approaches are language specific and require the component to expose its functionality via the constructs of a particular language. Finally, there are distributed components that are made available via communications networks. In this case the component is accessed via a communications protocol and this can be useful in allowing intercommunication among components across languages and platforms.

Whatever the deployment option the component technology must have well-defined mechanisms for ensuring that providers know how to construct components in a proper way and that clients of these components know how to access and use them.

3.4 Component Composition

Components are only of use if they can be combined with other components to form useful applications. How components are assembled together is called *component composition.*

3.4.1 The Component Framework

Szyperski provides the following definition of the component framework:

> *'A component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level' [Szyperski2002]*

A component framework (sometimes called a container) addresses the need to be able to plug components together to form useful applications. A standardised framework eliminates the need to handwrite code to combine components together. A framework will usually provide a mechanism for interconnecting components allowing them to be combined in a generic way.

There are two ways in which components can interact in a framework, either through wiring (connection oriented programming) or through contextual frameworks. In a connection oriented framework the 'plugs' of components are connected together and information flows directly from one component to another. In a contextual framework communication is achieved via services. In this case components communicate via services specifically designed to manage communication between them.

Components and frameworks are often confused. Lumpe argues that components cannot exist without frameworks and that a component by itself is meaningless without the context of a framework [Lumpe99]. An example is user interface components. On its own a user interface control is useless but when combined with other controls and placed on a window, useful applications can be created. It is true that even if a concrete component framework exists it is pointless to construct a component without knowing how it will interact with other components to form a useful system. Lumpe therefore provides the simplistic definition that '*a software component is an element of a component framework.*'

A common analogy used to describe component frameworks is a stereo system. A stereo system can be supplied in various components (CD, tape, tuner, etc), which are then wired together to form a system. The audio interconnections among components adhere to specific electrical standards. Likewise the electricity supplied to each component is standardised. These standards, the electrical connectors and the components themselves have to adhere to a 'framework' so that they can interoperate effectively.

To allow interoperation among components the component framework should be able to interact with and control components. A framework has to be built to accommodate a particular component standard or possibly multiple standards. In fact, realistically the framework will be built first with subsequent components implemented and tested against the component framework to ensure they are functioning correctly. A component framework may also include mechanisms for automating component composition. This may involve some type of either scripting language, programming language or glue [Schneider99]. The particular facility available for component composition will be dependent on whether the framework allows composition of components at compile time or runtime.

3.4.2 The Component Architecture

The component architecture is a particular set of rules governing the use of components and a component framework. [Szyperski2002] provides the following definition of the component architecture:

'A component system architecture consists of a set of platform decisions, a set of component frameworks, and an interoperation design for the component frameworks.'

Whereas a component framework allows component composition, the component architecture dictates the overall system-wide policies concerning the use of components. The component architecture is concerned with defining the overall principles of a software system and this architecture will dictate policies on functionality, performance, reliability and security. As such, a system may include multiple component frameworks all conforming to the same architecture. The component framework may itself be a component; in this case the component architecture may dictate the interconnection among multiple component frameworks.

3.5 Existing Component Technologies

Later in this thesis the concepts of component-based software will be used to build a component framework for software radio. To gain more insight into how these techniques can be put to use, this section examines three existing component technologies, namely: Java components, CORBA components and Microsoft based components.

3.5.1 Java Based Components

The Java language [Gosling96] is one of the most popular languages in use today. It is an object-oriented language that allows cross-platform operation using a virtual machine. The Java Virtual Machine (JVM) executes byte-code produced by the Java compiler. Various platform vendors can thus support Java by implementing a suitable JVM. Java has become a popular language and is in widespread use throughout desktop, server-side, internet and mobile applications.

Java is a very suitable language for developing software components. Java inherently supports features such as reflection and advanced networking capabilities which provide a rich infrastructure for developing software components. Thus, various component technologies have emerged for the Java platform. Of these two are of particular interest in this discussion, JavaBeans and Enterprise JavaBeans (EJB).

JavaBeans

A Java bean is a Java software component [Sun97]. A Java bean encapsulates functionality into a unit called a bean. JavaBeans address the need in Java to have some way of packaging functionality and resources into a module that can be repeatedly reused. A bean is primarily used for packaging graphical controls, but it is also suitable for creating general-purpose Java components. What differentiates a Java Bean (or Java component) from a standard Java class are the standardised facilities a bean uses to expose its functionality. Every bean must contain:

- Properties: A bean can expose properties that can be used to configure an instance of a bean. These properties are configured using set and get methods for each property.
- Events: A bean can provide or consume events. Events allow a Java bean to asynchronously react to or control other external users of the bean.
- Methods: A bean exposes all other functionality through standard Java methods (or functions)

In addition, JavaBeans relies on some features of the Java language to facilitate component constructs:

- Reflection (meta-data): The Java language supports reflection that allows external clients to query information about a bean. Using reflection it is possible to find out programmatically what properties, events and methods a bean exposes.
- Packaging: A Java bean can be programmed via a number of Java classes. In addition, resources such as graphics may be required. JavaBeans allows all code and resources to be packaged into a Java archive (JAR) file.

The JavaBeans standard applies many of the concepts of software components discussed in previous sections. In particular:

- Interfaces: The Java bean exports a well-defined standardised interface consisting of events, properties and methods.
- Unit of deployment: The Java bean is packaged as an independent unit (a JAR file) incorporating all the code and resources required for its operation.
- Meta-data: The Java bean supports reflection which allows users of the bean to query information about the bean programmatically. This allows an external client to dynamically query the capabilities of a bean.
- Black box: A Java bean represents a black box abstraction, in that knowledge of its internal operation is not required to make use of it as a component. Although it may be technically possible to inherit from a Java bean class, this is not normally done and the problems of white box reuse are thus avoided.

The JavaBeans standard does not however dictate a particular framework for composition of components. It is a connection-oriented component model and communication between beans requires connecting events mechanisms together. This usually requires a custom container to be built for each application. The JavaBeans standard is a minimal standard and so it faces other limitations. In particular there is no support for distributing Java beans via a network which makes it unsuitable for large enterprise scale applications. For this reason the Enterprise JavaBeans standard was created.

Enterprise JavaBeans

Despite similar names, JavaBeans and Enterprise JavaBeans (EJB) [Sun2001] work very differently as component technologies. Fundamentally they address different types of applications. JavaBeans defines lightweight usually graphical components whereas EJB defines a whole infrastructure for developing distributed transaction-oriented applications [Monson2001]. Thus the EJB component model deals with issues such as security, persistence, transaction management and distributed computing. The EJB standard defines three different types of components namely: entity, session and message-driven components. These component types are specifically designed for use in building data centric business applications.

Of particular interest in the context of this thesis is how EJB components are constructed and composed together to make useful applications. EJBs are programmed against a set of interfaces and base classes that define the functionality a bean should provide. These classes and interfaces are designed is such a way that the developer can concentrate on business logic without having to worry about the specifics of transaction processing or networking. The EJB standard defines an infrastructure that factors out these difficult aspects of programming into generic services that are used among all components. This approach of factoring out common functionality is closely related to aspect-oriented programming [Kiczales97].

Like JavaBeans, an EJB is packaged into a JAR file as its unit of deployment. This JAR file contains all the required code and resources the EJB requires to operate. EJB uses a deployment descriptor which defines what components should be included in the application and all the configuration required for these components to work together. The deployment descriptor is defined using XML (eXtensible Markup Language). EJBs are deployed via a container, a type of component framework (see Section 3.4.1) that allows component composition. The container is responsible for hosting the EJBs, providing the infrastructure that allows component functionality to be exposed to the outside world and allowing intercommunication among components. EJB containers are standardised via the EJB specification and various EJB containers (also known as application servers) are provided by different vendors. Standardisation ensures that EJB components are guaranteed to work within any container.

In terms of component technologies, JavaBeans are similar to EJB in how they are packaged (JAR files) and in their use of black box abstractions. They do however differ in the following ways:

- Interfaces: Whereas JavaBeans provides one type of interface consisting of events, properties and methods, EJB defines three different kinds of components each of which defines its own interface.
- Meta-data: EJB defines an interface that allows a container to query information about the capabilities of a component via meta-data

- Deployment: EJB uses an XML deployment descriptor file to dictate how components are assembled to form applications.
- Framework: EJB defines a standardised framework for component composition based on an EJB container.
- Distributed: EJB supports the distribution of components across a network.

3.5.2 CORBA Based Component Technologies

CORBA (Common Object Request Broker Architecture) is a standard maintained by the OMG (Object Management Group) that allows software from different environments and platforms to interact. CORBA was introduced in 1991 to address the growing lack of interoperability among languages, implementations and platforms. CORBA is an open standard for the production of distributed object systems. CORBA provides a mechanism whereby objects can communicate with each other regardless of where they are located, be it in the same programme, different programmes on the same machine or on separate machines [Balen2000].

Central to CORBA is an IDL (Interface Definition Language) and the ORB (Object Request Broker). IDL is a language for the specification of interfaces allowing the developer to specify what functionality an object will expose. CORBA defines mappings from IDL to many languages therefore interoperation across language boundaries is possible. The ORB is the system that performs the communication among CORBA objects. The ORB uses IIOP (Inter-ORB Interoperability Protocol) for communication among ORBs and thus allows the various objects to communicate. CORBA incorporates several services that are used in combination with the ORB to facilitate distributed object architectures.

When CORBA is used to expose the functionality of an object, this object can be viewed as a software component in terms of:

- Reuse: Once a CORBA object has been exposed (either locally or across a network) it can be reused by multiple clients.
- Interfaces: CORBA does not specify a particular common interface for CORBA objects as each object is allowed to expose its own interface. However, in CORBA an object's interface is translated to a common format so that objects written in different languages can interoperate. Although not strictly a well-defined interface in the component sense, this common format can be viewed as a type of well-defined interface in its own right. CORBA also supports events via an events service which allows asynchronous messaging between objects.
- Meta-data: Meta-data (or meta-information as it is often called in the context of CORBA) allows dynamic discovery of objects.

- Black box: A CORBA object provides a black box abstraction of sorts. Using a CORBA object only requires information about its interface and thus the internal operation of the object are hidden.

While CORBA objects can be viewed as software components, CORBA was not strictly designed as a component model and thus has many limitations. For this reason the CORBA Component Model (CCM) specification was created by the OMG.

CORBA Component Model

While CORBA itself provides an infrastructure for wiring objects together, the CCM goes a step further in providing an infrastructure for deploying components. The CCM is aimed at the same types of applications as EJB, and in fact EJB components can be used in conjunction with CCM components within the CCM standard. Like EJB, the CCM defines different types of components that represent the building blocks of enterprise applications namely: service, session, entity and process components. It also defines a container model, a packaging and deployment model and support for transactions and persistence. Again, these elements of enterprise data centric applications are of lesser interest in the context of this thesis, but a lot can be learned from looking at how CCM components are constructed and how they are composed into applications.

The following features of a CCM component are of most interest [Gschwind2002]:

- Facets: A facet is the interface that a component exposes. A CCM component can contain multiple facets.
- Receptacle: A receptacle is a way to specify what interfaces a component requires from other components. Alternatively, a receptacle of a component specifies what facets of another component it will use.
- Events: CCM components can both provide (publish) and consume (subscribe to) events. This provides an asynchronous way to pass information between components.
- Attributes and configuration: A CCM component can be configured via attributes which are identified using named values.

A CCM component is packaged into a single redistributable file called a CCM assembly. Like EJB the CCM uses an XML configuration document to describe these components and how they should be deployed.

To host CCM components the CCM defines a container (i.e. a component framework). This is similar to an EJB container. The container provides interfaces for providing transaction, security, persistence and notification services. The facets, receptacles, event sources and sinks allow components to be connected together to form the application.

3.5.3 Microsoft Component Standards

Microsoft have produced a variety of standards for creating software components. These standards address both standard application development and distributed enterprise applications.

COM

COM (Component Object Model) is a standard by Microsoft for creating reusable software components. COM and its predecessor OLE (Object Linking and Embedding) are language independent standards. Unlike the Java approach which uses a common virtual machine, the COM approach is to use a set binary standard. COM components can be implemented in any language, but must conform to the binary format set out by Microsoft.

A COM object enforces a black box abstraction by exposing its functionality through a simple interface mechanism. Every COM object must implement the same well-defined interface called IUnknown. This interface allows the user of a COM object to programmatically acquire the information required to use the COM component. IUnknown must always supply the three methods: `QueryInterface()`, `AddRef()` and `Release()`. Using `QueryInterface()` a client wishing to use the component can query a table to obtain references to the interfaces supported by the component. Using the reference a client can make use of the component. The `AddRef()` and `Release()` methods are used to implement reference counting which allows the component to keep track of instances of the component.

COM is a simple standard used primarily for application development. Other component standards such as ActiveX build on COM by exposing different interfaces [Chappell96]. The COM standard itself does not dictate any particular framework for combining COM components and this is usually left to the application developer. Although simple, COM demonstrates the use of some fundamental concepts of component software:

- Reuse: COM components support black box reuse of software.
- Interfaces: COM supports a well defined interface structure. The binary standard used by COM can be more efficient than IIOP used by CORBA or the use of a virtual machine which can introduce performance overheads.
- Meta-Data: A COM type library can be supplied that allows a client to dynamically discover information about the interfaces a COM component exposes.

DCOM (Distributed Component Object Model), an extension to COM, allows COM components to be used over a network and facilitates the creation of distributed component based applications.

COM+

COM+ was the first technology to combine support for transaction monitors and ORBs (Object Request Brokers). A transaction monitoring system forms a type of operating environment for applications in which it automatically manages transactions, resource management and fault tolerance. An ORB permits objects to be used across a network allowing the application to be distributed. COM+ combined these principles facilitating the use of COM objects in this environment, which was particularly important for enterprise business applications. COM is analogous to JavaBeans, and COM+ is analogous to EJB.

COM+ applications can still be built individually today but the services of COM+ have been integrated into the new .NET platform. Of particular interest is a reoccurring paradigm among component models and component containers. EJB, CCM and COM+ all factor out common services from component implementations. This simplifies component development by making these services universal to the architecture of the component system.

.NET

The .NET framework was introduced by Microsoft as a general purpose framework for creating applications. The core of the .NET framework is the CLR (Common Language Runtime). Similar to the Java Virtual Machine, the runtime allows the execution of a platform independent binary code called MSIL (Microsoft Intermediate Language). Whereas the Java Virtual Machine has typically only been used to execute Java programs, the .NET CLR is specifically designed to provide enough facilities so that compilers can be easily written for any languages.

The .NET framework itself does not dictate a particular component model as it encompasses a broad technology base for developing many different types of applications [Löwy2003]. .NET itself can be used to build different types of component models and inherently supports many features that make this easier. Of these the following are of interest:

- Language independence: .NET allows multiple languages to interoperate through the CLR allowing components implemented in different languages to interoperate at a binary level
- Packaging: .NET allows code and resources to be packaged into a unit called an assembly
- Interfaces: .NET is built on object-oriented principles therefore it supports the infrastructure required to develop components with well defined interfaces
- Attributes and reflection: .NET has inherent support for meta-data via attributes and allows reading of this data through its reflection APIs. This provides powerful support for allowing dynamic use of components at runtime.
- Remoting: .NET supports distributed objects which are useful in building distributed component-based applications

3.6 Summary

This chapter has provided an overview of software engineering, in particular the principles of component-based software. It has shown that software engineering provides principles and techniques for dealing with software complexity, and for developing software that is adaptable, flexible and reusable. The main principles covered in this chapter can be summarised as follows:

Principles	Description
Object-oriented analysis	Object-orientation promotes the building of quality, robust software
UML	UML provides an effective tool for graphically modelling an object-oriented design
Design Patterns	Ensure that tried and tested paradigms are used throughout the design
Classes, Inheritance, Interfaces and Polymorphism	The basis of object-oriented programming
Granularity, Cohesion and Coupling	Useful metrics for designing good quality objects, i.e. objects that help to reinforce the quality and stability of software
Component-Based Software	Component-based software promotes the packaging and reuse of software
Black box	Software components feature black box abstractions which hide the client from the internal implementation of a component
Granularity	Component granularity is important in that it affects the reusability of a component
Interfaces	A component must expose a well-defined interface
Dependencies	An individual component should have minimal dependencies but components may sometimes be interdependent
Meta-Data	Meta-data and reflection are important facilities in allowing the dynamic discovery and use of a component
Deployment	A component model should specify how a component is packaged and deployed
Architecture	The architecture of a component technology should specify the rules associated with building applications for the domain the architecture addresses
Frameworks	Components are pointless without frameworks. A component technology should provide a framework that allows components to be connected together to form useful applications.

Figure 3.1 – Summary of Software Engineering Principles

The reason for discussing software engineering in this chapter has been to assemble techniques and practices for tackling the problem of developing software for software radio. This approach has been taken because software engineering practices are not typically applied to software radio and as explained in Section 3.1, history has shown that ignoring these principles in favour of optimisation can lead to un-maintainable, complex and expensive software.

By looking at these principles and seeing them in use in technologies like EJB, COM+, .NET and CCM much can be learned about how to build quality software. For example, all these technologies share a common characteristic in that they support base services. The function of these services is to factor out common functionality required by many components. Instead of each component having to re-implement this functionality they can reuse these base services, thus greatly simplifying the implementation of components. A good example of a base service is synchronisation. The basic problem may be that multiple components require serialised access to a resource. Instead of each component requiring knowledge of how to negotiate, acquire and release the common resource, a 'synchronisation' service built into the fabric of the component framework automates the process. This makes it seamless and trivial to gain access to shared resources. Services are useful as they demonstrate how the concept of a component framework can be used to simplify the development of complex systems, while allowing software to be reusable, flexible and adaptable.

Another common trait from the component frameworks analysed is their use of well-defined mechanisms for performing operations such as firing an event, calling a method or deploying a final system. Each system as part of its architecture and framework defines a set of principles, each of which must be adhered to if software is to function correctly. This effectively enforces a set of rules both on the programmers that write components and those that write the framework. For example, if an event is fired by creating a block of data and placing it in a queue then this should be the one and only way this is possible. A component framework will not allow any circumvention of this rule. This approach results in less ambiguity and ultimately more robust software.

A conscious decision has been made during this work to not reuse an existing component technology such as COM, EJB or .NET. Some component frameworks although useful in demonstrating the principles of component software would be completely unsuitable for developing radio systems. For example, EJB has been specifically designed for developing multi-tier business applications that are based around databases, business processes and content delivery to users. This is clearly a completely different type of end-application to radio and so the EJB semantics would make this pointless. EJB is also a Java based language and although the approach presented in this thesis is not particularly concerned with code performance, in the current state of the art Java itself is rarely used for high data rate signal processing on GPPs.

In general component models such as COM+, CCM and even the support provided by .NET have been designed to accommodate the needs of developers building mainstream information-based business applications, everything from banking systems to user applications. Thus, the base services they provide do not meet the needs of radio applications. These models have not been designed with signal processing in mind. They have no built-in semantics for representing a signal,

performing a mathematical routine or interfacing with hardware. This thesis presents a system that fills this void.

Another compelling reason not to use these component models is that to fully explore the use of a component-based approach to software radio requires a fresh look at both software development and component models in the context of radio. To use a component model designed for building a type of software unrelated to radio systems would make it difficult to fully explore this space. Instead of trying to fit a radio system into a component model not designed for this purpose, the approach taken has been to develop a component model completely suited to radio systems.

In summary, software engineering techniques such as services, the mechanisms of events, etc and component models are not currently used in radio systems, although as discussed, it is this type of software that is increasingly required. The remainder of this thesis demonstrates how this can be done. Chapter 4 discusses the particular type of radio system being built, namely the reconfigurable radio. The reconfigurable radio concept is completely dependent on the software engineering principles presented in this chapter. Many of the principles of the reconfigurable radio are built on the premise of the component-based approach and the reconfigurable radio is only fully realisable using these techniques.

4 Reconfigurable Radio

4.1 Introduction

This chapter presents a discussion of reconfigurability in software radio systems, which is the core concept of this thesis. Section 4.2 discusses reconfigurability in detail and defines the three categories of reconfigurability; application, structural and parametric. Section 4.3 provides a detailed discussion on all the issues surrounding the development of software for radio systems. Section 4.4 discusses the possible architectures for developing a system that is highly reconfigurable.

4.2 Reconfigurability

The term 'reconfigurability' is used extensively throughout the literature and refers to many different facets of software radio reconfiguration in both the hardware and software domains. For this reason, the following discussion is presented in order to precisely characterise what is meant by reconfigurability in this thesis.

4.2.1 Reconfigurability From Hardware to Software

Figure 4.1 depicts a graph that has been created to illustrate the level of reconfigurability of the various software radio hardware solutions which were introduced in Chapter 2. On the graph two types of reconfigurability are considered. A device can be considered to be reconfigurable if its functionality can be changed (blue line). A device can also be considered to be reconfigurable if the way in which the functionality is performed can be altered (red line). Using these definitions the devices listed on the graph have varying degrees of reconfigurability.

As an example consider an ASIC. It performs one particular dedicated function which cannot be changed. However, the ASIC does allow the parameters of the function it performs to altered. For example, a GSM baseband processor ASIC cannot be used to process the baseband signals of any other radio standard but it will offer variability in how this function is performed, perhaps by allowing the output power to be changed. However, such changes require dedicated hardware such as a microcontroller to be used therefore giving it a 'Low to Moderate' score in how it allows the altering of functionality (red line).

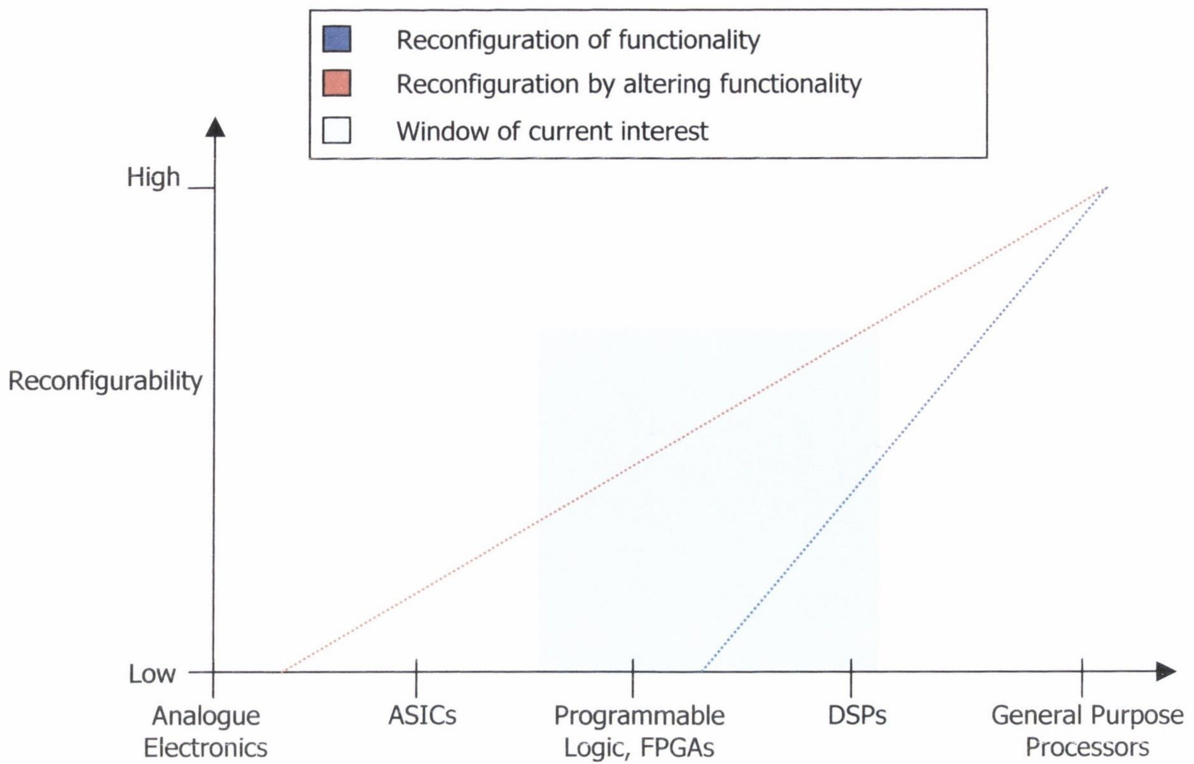


Figure 4.1 – Level of Reconfigurability for Various Signal Processing Devices

The FPGA is another example, a device that can both change its functionality and offer variability in how this functionality is performed. While it scores moderately high in terms of altering functionality, it scores low in its ability to change its functionality. Although some FPGAs allow full or partial dynamic reprogramming, in practice FPGA development is a longer and more complex process than say DSP or GPP software development. This is because FPGA designs are highly bound to the type of FPGA being used and require decisions on routing and placing of functionality on the physical device. Dynamic reprogramming makes this task more difficult as some of the reconfiguration scenarios may be unknown at design time making it difficult to allocate resources on the FPGA. In contrast a device such as the GPP does not require more silicon real estate to implement new functionality, only additional software programs. Also, reconfiguration of an FPGA can have an impact on the other circuitry surrounding an FPGA, so special considerations and thus limitations in reconfigurability are common.

The shaded window surrounding FPGA and DSP technology represents the current state of the art, i.e. these hardware devices are the main focus for developing software radio solutions today. While these technologies offer the performance and real-time behaviour required by today's radio standards, the trade-off that results in using these devices is a limit in reconfigurability.

In terms of this discussion it is important to expand on the role of software in reconfigurable devices. Not all of the devices in Figure 4.1 run software in the conventional sense. For example, FPGAs require the use of a hardware description language such as VHDL or Verilog. Some other

languages exist also including System-C [System-C] and Handel-C [Chappell2002] that offer more software-like semantics for expressing FPGA functionality. Beyond these semantics though, these languages simply offer higher level constructs for expressing hardware functionality. They are hardware specific and there is limited abstraction between language and hardware. Thus FPGAs cannot be reconfigured to the same degree as a purely software-based device. For example, the FPGA cannot reorder the way in which it processes two signals without reprogramming hardware.

Software used in DSP processors is more similar to software running on a GPP rather than the FPGA. As in the case of the FPGA, the conventional method of programming DSP processors has moved from assembler languages to more high-level languages such as C. The difference however is that DSP processors execute instructions whereas FPGAs take a reprogrammable logic approach. DSP processors are therefore better suited for reconfigurable tasks. However they still have features that limit the levels of reconfigurability possible.

DSP processor designs are highly bound to the particulars of the processor and the surrounding hardware. This is required to achieve constraints in real-time behaviour and to optimise power and performance. This however limits the devices ability to reconfigure. For example, a maximum level of reconfigurability would allow a software radio device to change its own functionality by applying new algorithms and loading new code as required. In the DSP processor this would be difficult to achieve as any change to code could affect the hardware-oriented aspects of the design such as real-time behaviour and performance. Also, since DSP code is highly bound to hardware, most DSP implementations maximise the use of hardware by manipulating low-level aspects of the system such as bus access and caches. Thus, it can be difficult to allow any aspect of the system to change dynamically without breaking or disrupting another.

From this discussion it is evident that devices can be reconfigurable to varying degrees, yet the term reconfigurable loosely applies to them all. The purpose of this work is to make advances towards the creation of a more ideal software radio. Chapter 1 discussed the choice of platform for this research, the GPP. This choice also has major implications for the level of reconfigurability that can be achieved in a software radio. Using the GPP provides a flexible environment and allows the development of software radio systems with exceptional levels of reconfigurability. The GPP shields software development from the particulars of hardware with many hardware specific functions such as virtual memory and multi-threading handled by the operating system. This simplifies software development and allows many different software implementations and hence software configurations to be used interchangeably. RAM and persistent storage are also important factors in that they allow vast amounts of different configurations and code to be stored for reconfigurable purposes. In this context reconfigurability is thus software-based, and although there

may still be some hardware aspects involved such as control over an RF front-end, the majority of radio functionality is implemented in software.

4.2.2 Reconfigurability Defined

Following on from the previous discussion it is useful to strictly define reconfiguration using three distinct categories, namely, application, structural and parametric reconfiguration.

- *Application Reconfiguration* - At this level the whole radio can be reconfigured by replacing the software of the software radio. This type of reconfiguration can allow a radio to completely change the application it performs. For example this might involve the same hardware being reconfigured from being a two-channel analogue FM transceiver to being a 10 channel digital BPSK transceiver.
- *Structural Reconfiguration* – Structural reconfiguration allows components to be added, replaced or reorganised while the radio is operating. For example we may decide to change the way in which a signal is processed, perhaps introducing two stages of filtering instead of one. In this case the radio will still perform the same function but reconfiguration may have benefits in improving signal quality, power consumption or performance.
- *Parametric Reconfiguration* – Software allows the individual parameters of signal processing functionality to be changed dynamically. For example we may want to change the coefficients used by a filter or change a particular frequency setting. This level of functionality will allow individual elements of the radio system to be exposed for reconfiguration during the operation of the radio.

Just as important as each degree of reconfigurability is the time each one takes. For example, one system may support application reconfiguration in that it can be sent to a factory to be reprogrammed, another may perform the same reconfiguration seamlessly without any loss of communication. In reconfigurable radio these changes should occur at runtime and reconfiguration should occur as fast as possible without any loss of communication.

The degrees of reconfigurability discussed here are somewhat different to those in current mainstream or commercial approaches to software radio as discussed in Chapter 2. Those approaches base their reconfigurability solely on the capabilities of hardware whereas reconfiguration in this discussion is based entirely on reconfiguration in software. For this reason the term 'Reconfigurable Radio' is used for the remainder of this thesis to differentiate the existing disparate variety of approaches to software radio from the more software-centric, GPP-based approach taken in this thesis. Thus for the purposes of this work the reconfigurable radio is defined as follows:

'The reconfigurable radio is a software radio with a minimal-hardware RF front-end with the remainder of processing performed using general-purpose processors. The software of a reconfigurable radio allows application, structural and parametric reconfiguration.'

4.2.3 The Benefits of a Reconfigurable Radio

The primary benefit of reconfigurable radio is that it allows traditionally fixed operating parameters to become variable. This allows radio systems to become more flexible in how they communicate. Flexibility has been constrained in the past due to the characteristics of the communications channel, an environment that is noisy, lossy and corruptive to the transmitted signal. To achieve communication in this medium strict standards for operation have been required. These standards often limit radio systems in realising the full potential of the medium. For this reason most wireless standards are fixed in modulation scheme, bandwidth, frequency allocation and power. With reconfigurable radio, these constraints can be somewhat relaxed, as it is possible to build flexible terminals that constantly reconfigure themselves to suit their environment. Reconfigurable radio offers an unprecedented opportunity to create devices that can offer better, more reliable communications. Consequently many operating parameters which are usually rigidly fixed, can now become adaptable, for example:

- Propagation: Channel characteristics such as multi-path fading require additional processing by a radio. Using reconfiguration the radio can dynamically change how it deals with these issues.
- Power: The radio can dynamically alter its RF-power output to suit its operating environment.
- Location: According to its location the radio can dynamically change many parameters that may improve its ability to communicate.
- Modulation scheme/bandwidth: the radio can dynamically change the modulation schemes and hence the bandwidth it uses to communicate. This can be varied to different degrees from one off changes to dynamic modulation changes even during transmission.
- Frequency: With a general-purpose terminal capable of operating on a large range of signals, terminals will be able to dynamically change they frequency used to communicate.
- Algorithms: The radio can dynamically reconfigure itself to use different algorithms to process signals. This will allow it to change its operation to suit many different scenarios.
- Power consumption: By dynamically changing the way in which a radio processes signals it will be possible to vary the power used by the radio device which is important for battery powered mobile equipment.

Also there are intrinsic technical advantages and opportunities possible using the reconfigurable radio approach, for example visibility and rapid development.

Visibility: To test or calibrate an analogue radio system typically requires probing a circuit with an oscilloscope or spectrum analyser. The signal has to be isolated in a particular part of the circuitry and then interpreted via the general-purpose tools available. Often, due to the use of analogue integrated circuits and prefabricated modules, the signal of interest is not available as an output because a module implements several stages of the radio design. For example, a baseband processor chip may amplify an IF signal, down convert it to baseband and perform demodulation. All this functionality occurs internally within the chip and often the signal of interest cannot be isolated.

Within a software radio all signals exist digitally and are available at runtime. This is useful for two reasons. Firstly, the radio system can have built-in validation. As all signals are accessible the system itself can perform verification of signal integrity at various stages in the radio. This can be done both during development and after the radio is deployed. Secondly, using graphical tools these signals can be directly accessed. Not only can these signals be viewed in the traditional way (for example using oscilloscope traces and spectrum analysis), but also new ways of graphing and interpreting these signals are possible without building new hardware. This can be useful for either exploring how existing radio technologies work or as a tool for creating new types of radio systems. Thus, as these two examples demonstrate, the software radio brings an increased level of visibility to radio system internals.

Rapid Development: Speed of development is often overlooked when discussing software radio. Speed in this context refers to how long it takes to design, implement, test and deploy a radio system. Analogue radio systems have to be physically built before they can be properly tested. Simulation can go some way in reducing the need for a physical prototype, but nevertheless at some stage in the design an analogue prototype must be constructed. In software radio, beyond the RF front-end there is no need to build a physical prototype at every stage in the design. The fundamental paradigm shift here is that the prototype is the software under development. Instead of incremental physical prototypes that can take months to design and build, the radio system is tested during the development process, rapidly increasing the entire development process.

This fundamentally changes the radio system development process. In this environment the complexity of a radio system is now contained in its software programs rather than its hardware. Increasing demands on functionality require additional software development rather than additional hardware design. The process can be made even quicker through software reuse. Designers can add

on new features to the radio system by reusing third party software components. This eliminates the need for them to re-implement functionality themselves, thus rapidly reducing development time.

4.3 Software for Software Radio

The following topics are of major concern when designing a piece of software:

1. Reuse.
2. Abstractions.
3. Adaptability and Flexibility.
4. Complexity.
5. Security.
6. Portability.
7. Real-time Behaviour.
8. Upgrading and Versioning.

It is important to discuss each of these topics in the context of designing a reconfigurable radio.

4.3.1 Reuse

Software reuse will become just as important in radio systems as it has become in mainstream software. In the DSP of radio systems many signal processing algorithms and functional algorithms occur frequently throughout different radio designs and standards. For example, BPSK (Binary Phase Shift Keying) modulation occurs frequently throughout many types of communication and thus a properly constructed piece of software implementing BPSK can be reused in multiple applications without re-implementation. This raises the question as to what is the best way to reuse elements of software radios. To address this issue, the role of software granularity, cohesion and coupling must be analysed (as discussed in Section 3.2.2).

The *granularity* chosen in a software radio design will be a determining factor in how well software objects can be reused. Using a fine granularity, DSP software would be broken down into fundamental units that represent the building blocks of DSP systems. For example, one approach could be to break down DSP algorithms into adders, multipliers, or multiply-accumulate elements. However, this approach does not leverage effective reuse as the objects are too small and generic and do not contain enough domain specific functionality to be labelled as software radio components. The user of these components would have to introduce too much 'glue code' and thus reuse would be lost.

A larger granularity is also possible, for example on the system scale where components encapsulate systems such as two-way radios or GSM base stations. Although these are suitable as

methods for distributing or replicating a complete system, they are not suitable elements of reuse for building the software of software radio systems. This type of object is too big and thus only reusable in completely domain specific applications.

The granularity balance for software radio can be struck by viewing the system as reusable radio parts, each implemented in software. These parts each implement a different aspect of common radio functionality such as the QPSK modulator, low-pass filter or speech-encoder. A component with this granularity holds enough functionality to warrant reuse but is not application specific enough to limit its usefulness and thus it can be applied in a wide range of different scenarios.

Cohesion is important in software radio in that the reusable pieces of software that make up a software radio system should be logically related in such a way that they enable effective building of quality radio systems. In an individual component, the functionality or methods it exposes should be logically related and contribute towards the same problem. This is a problem in mainstream software as diverse functionality can be implemented by building objects in similar ways, thus bad cohesion is a result of exposing functionality haphazardly. Software radio on the other hand is domain specific, thus most objects in the software radio system will be DSP algorithms and thus common cohesive DSP interfaces can be used to expose the functionality of an object.

Cohesion also plays a role in larger scales. The sub-systems and elements that make up a software radio system should be cohesive in that they all relate to the problem of software radio in the same way. For example, using elements of different granularity throughout the system would result in bad cohesion, as it may be difficult to combine fine and coarsely grained elements to form the radio system. Objects should be logically designed employing the particular style of the software system being used.

Coupling is a very important aspect of developing reusable software for software radio. As discussed in Section 3.2.2, the level of coupling will dictate how dependent a software element is on other elements. In software radio, coupling is especially significant as it has an important consequence to DSP software that does not typically appear in mainstream software. When designed well, mainstream software is easily tested. In the majority of software, the correctness of software is typically boolean in that the software either performs its function correctly or it does not, examples being, 'the numbers were added correctly', 'the e-mail was sent', 'the disk access failed'.

However in DSP software errors are not so apparent. A piece of software can be functioning perfectly, but in reality it is not producing the correct result. Thus, DSP software typically requires

additional testing from the DSP domain to determine whether the software is functioning correctly, for example, testing the signal to noise ratio or performing frequency analysis. Dependencies and hence coupling add to this problem, as traditional approaches to ensuring correct functionality across dependency boundaries are boolean-based logic rather than DSP-aware constructs. For this reason software-based DSP systems can suffer more from dependency problems than other software in that changes to reusable elements can have unnoticed or undefined effects across a software radio system. Also, DSP algorithms themselves are not standardised in any way so algorithms such as filters implemented by two different programmers may not produce the same result.

To illustrate this problem consider the tightly coupled software element shown in Figure 4.2. This software element is a simple channel extraction implementation using the three stages of mixing, filtering and decimation to extract a signal of interest from a wideband source. The common approach of object-oriented design would be to delegate operations such as filtering to other objects, as filtering is reused in many scenarios throughout radio design. The difficulty arises when some aspect of a reusable element is changed, for example, the programmer notices a bug in the filter windowing function and fixes it. In this case there is a risk that the expected behaviour of the channel extractor and other elements that depend on this filter will be altered.

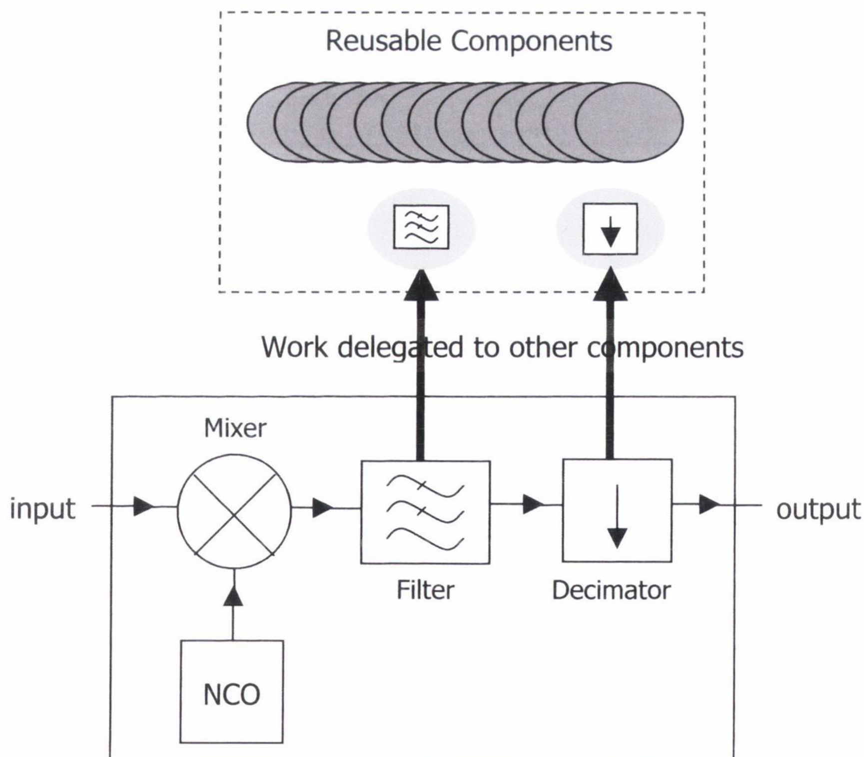


Figure 4.2 – A Tightly Coupled Software Component

In the channelisation example this may result in an altered frequency response, ultimately having a knock on effect throughout the system. The underlying reason this can occur is because the original implementer of the channelisation algorithm would have based their design on a filter that they assumed would always produce the same values. Thus changing the characteristics of this filter can lead to unexpected results in any component that reuses this element. The main point to note here is that even though a piece of software works correctly and consumes and produces valid data, this does not mean the data is correct when analysed and interpreted using DSP.

The channel extractor is a very simple example, but there are more subtle cases where this type of dependency-caused error can take place. For example, consider the following hypothetical example of a decimator algorithm. A programmer decides to improve the performance of a reusable decimator algorithm by changing the number of decimation stages used in the algorithm. While the decimator still performs its function within specification and the code still exposes the same interface, the programmer has inadvertently introduced noise into the system by introducing multiple stages of processing. This has a knock on effect throughout the system in multiple places making the source of the problem difficult to find. Another interesting angle on this problem, but with the same negative result occurs if the programmer actually improves the noise performance of an algorithm in some way. Again, this can also have a knock on effect as other elements in the system are designed and tested against the noisier decimator, and thus a new decimator producing different signals could result in undesired functionality such as glitches.

To combat these types of problems, again a balance has to be struck, this time between coupling and quality. If a complex algorithm is subject to change and interpretation then it should be encapsulated into a reusable object and not delegated out to external objects. Although some reusability is lost, this approach will yield better results overall for such applications. If extensive delegation is to take place then strict practices must be adhered to ensure the integrity of the software. This may involve documentation procedures or if possible the software environment and hence the mechanism for reusing elements should be DSP-aware in that inconsistencies can be easily detected.

Finally, aside from technical issues, it should be noted that reuse and hence properties such as granularity, coupling and cohesion also have an economic consequence. While many organisations will practice reuse for internal development, reuse is also important on a larger scale as it fosters a market for buying and selling software. Take granularity for example; too small a granularity and elements will not contain enough functionality to warrant their sale. Too big a granularity and elements will be too application specific therefore reducing their market potential. Thus for economic and technical reasons, software for software radio systems should ensure that objects are properly designed for reuse to ensure their overall success.

4.3.2 Abstractions

How software for radio systems is viewed plays an important role in ensuring its success and contributes towards an effective level of reuse. As discussed in Section 3.2.2, visibility is an important aspect of partitioning software into reusable elements such as objects or components. Visibility specifies how much the internal workings of a software object are exposed to the users of that object. 'Black box' abstractions shield the user of the object from the internals of that object. Likewise, 'white box' abstractions allow the internals of the object to be extended through mechanisms such as inheritance.

It is important to consider what type of abstractions will be used in the construction of software for software radio. Black box abstractions offer the potential to allow the development of software radio systems without requiring specific knowledge of how the system works. This could occur in various ways. For example, the whole software radio system itself could be treated as a black box software component. The developer would use a well-defined interface to create new software radios. In this case the developer would be shielded from the operation of the software radio system thereby protecting the internal workings from disruption.

Another way of applying the black box abstraction is to view the various reusable objects of a software radio system as black box components. In this way these components are combined together to form various software radio solutions. The developer who combines these components does not require knowledge of how these various reusable elements work, just how to make them work together.

While the advantages of black box reuse are evident, there are some problems with black box reuse when developing software for software radio. Performance is always a primary concern and the overhead of maintaining strict interfaces through black box abstractions could hinder the performance of the radio system. In this case white box abstractions may be more appropriate in that the reusable elements become more flexible and can be altered to improve performance.

Overall in designing software for software radio we must strive to use black box abstraction as much as possible and only break this abstraction when the specifics of software radio pose no other alternative.

4.3.3 Adaptability and Flexibility

Adaptability will enable a piece of software to be reused beyond its original design. In a software radio this corresponds to developing signal processing algorithms that can be adapted for use in new applications. A prime example of an adaptable element is an FIR filter. FIR filters are used

extensively throughout many DSP systems. For an element such as an FIR filter to be adaptable it must facilitate its use in many different scenarios. For example the filter should be able to work with different sources of signals, possibly represented using different data types or supporting various methods for processing data. If a software radio is created out of adaptable elements then the level of adaptability will determine how often this element and hence code can be reused in other applications.

Flexibility allows a software radio to offer variability in how it performs its function. Whereas adaptability facilitates reuse of an element in different scenarios, it is mostly concerned with the technical issues of how an element exposes its functionality. Flexibility on the other hand concerns the actual functionality the element provides. Flexibility in the case of the FIR filter will ensure that the FIR filter provides enough control over the FIR algorithm itself. A suitable level of flexibility will allow us to change the window of the filter or specify our own windowing function. Flexibility will allow us to specify a range of increments for changing the cut-off frequency of the filter. Flexibility may even offer us the functionality of designing the filter coefficients for us. Flexibility and adaptability go hand in hand in creating reusable software for software radio.

4.3.4 Complexity

Complexity is a problem that faces any large software system. As radio technology continues to move towards more software-based implementations the amount of software required to build a radio system will continue to increase. Without proper management complexity will start to emerge in software radio systems in the form of bad quality and difficult to maintain software.

This problem can manifest itself in many ways. As demand increases for new wireless applications and increased capability, new and more complex DSP algorithms will be developed to meet the needs of these applications. Whether implemented on reprogrammable hardware or high-level software, the amount of software being implemented for software radio systems will continue to increase. As well as increasing DSP code, there will also be an increase in code enabling new capabilities such as software download and interoperability. If the current approaches being used for FPGA and DSP processor development are carried forward into these future systems, (as discussed in Section 3.1) a 'software crisis' of sorts could emerge in the domain of software radio. These approaches based on functional decomposition and hardware-bound languages do not encourage the building of quality software for large systems.

To deal with the problems of complexity in software radio the methodologies of software engineering must be brought to bear on the problem. A combination of object-oriented and component-based software approaches should be used. However, most of these techniques are

tailored for building software that values stability, robustness and maintainability over performance whereas performance is often a critical issue in software radio applications. Although a piece of software may be well built and highly reliable it may not be feasible to use the software if it requires an impractical amount of processing power.

A balance must be struck between achieving the required code performance while also managing the complexity of the software. Complexity ultimately increases the cost of a device as the more complex it becomes the more costly it is to maintain. Thus, some designers may decide to invest in more powerful hardware allowing them to reduce complexity by using software engineering methodologies. This may prove more cost efficient in the long term, as the cost of more expensive hardware may be less than that of maintaining a complex product over many years.

4.3.5 Security

Security is an important topic that has always surrounded telecommunications. Wireless communication is prone to eavesdropping and hence security of communications over the wireless channel is particularly relevant. Cryptography is therefore often employed to secure wireless communications. Physical modulation techniques are also used to prevent denial of service attacks an example being spread spectrum technology, which can be used to prevent radio jamming. As well as the existing threats of eavesdropping and denial of service, software radio introduces a new unique challenge to securing wireless communications. This challenge is radio viruses.

It could be possible to build a 'radio virus' or 'radio worm' similar to the viruses and worms written to infect computers on the Internet. Attackers could exploit weaknesses in the implementation of a software radio to gain control of the device. A similar type of attack occurs today on the Internet by viruses and worms that exploit buffer overruns. A buffer overrun is caused by a bug in a program allowing an attacker to overwrite a buffer in computer's memory. This can be exploited by writing a malicious program into the computers memory giving the attacker full control over the device. The first buffer overrun attack occurred in 1998 with the Morris worm [Eichin89]. A survey of buffer overrun techniques can be found in [Cowan2000].

In the case of the software radio, bugs in signal processing software, or the underlying operating system as in the case of a GPP, could permit an attacker to send signals that manipulate a buffer overflow in the radio device. For example, a digital communications standard such as GSM expects fixed sized frames of data with a standardised frame structure. Any receiver that does not check the values in the received frame structure correctly could be open to attack. Attackers could send malicious frames containing non-standard values thus exploiting weaknesses in the system and

giving them full control of the terminal. A denial of service attack could pit radio terminals against each other or against base stations to disrupt communication by flooding the spectrum with unnecessary transmissions. The attacker may not even have to manipulate a particular buffer overflow weakness; a specially crafted transmission may be enough to 'confuse' the radio system and render it useless.

Although radio systems do not face this threat today, it could become a serious threat if software radio terminals become more standardised and ubiquitous. If software radio systems become commonplace and are used for a variety of applications then there will be an abundance of terminals and hence more potential and incentive for an attacker to find weaknesses in a device. To prevent these types of attacks software radio systems must be designed to incorporate secure, formally validated techniques to prevent denial of service attacks, unauthorised modification of software and to maintain communications privacy. Software downloading to radio terminals must also be secure and thus code for software radio systems needs to be distributed securely. It must be digitally signed [RSA78] to ensure that code loaded remotely is from the correct author.

Many different approaches can be used to secure software radio systems in the future. The best deterrent will be good software designs that inherently support security and practices that leverage good quality software, as bad quality results in flaws that can be exploited. Other complementary procedures may have to be introduced such as code validation and rigorous testing procedures. Overall, a secure software radio system will require vigilance and recognition of possible threats.

4.3.6 Portability

Portability enables software to work on multiple platforms. With current software radio technology portability is difficult. The variety of hardware platforms and software techniques means that it is difficult to build a single piece of software that will run on many platforms. Portability is still a problem in general-purpose computing where there is standardisation amongst computer manufacturers and languages enabling elements of cross-platform and source-level portability. Signal processing hardware however has not yet reached this level of standardisation and code for DSP processors and FPGAs, etc are mostly manufacturer specific. While it is possible to programme some of these devices using either C or variants of the C language, a practical implementation typically requires hardware specific instructions and hence proprietary development languages.

Portability will continue to be an ongoing challenge in both general-purpose computing and software radio. Improving portability for software radio systems will reduce the cost of developing, maintaining software thus allowing for better quality software.

4.3.7 Real-Time Behaviour

Many radio standards dictate the use of strict timing and latency requirements for communication. The software of a software radio system must be able to facilitate the real-time nature of whatever scheme is being implemented. This has been a primary driving factor for the DSP processor in that this device facilitates the development of real-time code and allows the developer to be deterministic in how long operations will take. In the GPP however, this poses a significant challenge. Typical GPP systems are based around the use of general purpose operating systems that typically do not meet the latency requirements of existing radio standards. For example, the GSM standard requires timing of TDMA (Time Division Multiple Access) frames in the order of microseconds whereas the thread scheduler of Linux and Windows offers only tens of milliseconds accuracy.

There are however some approaches in alleviating these problems (as discussed in Section 1.4). A real-time operating system can be used to allow microsecond-level timing on GPPs. Even without such an operating system more accurate timing can be aided by the generic front-end, and by specific implementations of drivers which facilitate the type of accuracy required by these applications. A different approach altogether would be to relax the need for such stringent timing and to leverage the flexibility of a software-based radio system in meeting the demands of the application. For example, in a data communications system the requirements on timing may not be as stringent as they are in a voice system. In this case algorithms could be used to introduce functionality that compensates for inaccurate timing through signal processing or buffering techniques.

4.3.8 Upgrading and Versioning

A primary motivating factor for software radio has been the promise of general-purpose radio devices that allow functionality to be upgraded. There are however practical challenges to making this type of upgrading a reality. Specifically, the possibility that multiple versions of the same air interface exist poses a significant problem in that it could hinder effective communications.

There are two ways in which this problem can manifest itself. Firstly, incompatible versions of software can cause the system to fail. For example, a radio device that is partially upgraded may download a new speech encoder or modulator algorithm. This new software is however incompatible with the remainder of the system and thus crashes. The problem is further complicated if a system consistently upgrades its software by downloading new pieces of software. Without proper management each device could contain different combinations of software in configurations unforeseen by the manufacturer. This type of problem has plagued mainstream

operating systems for some time and software radio systems need proper management to avoid this type of problem.

The second type of upgrading problem is more subtle in that although the software may be functioning properly, the use of multiple software versions may make communication error prone or impossible. This type of error usually happens when an initial release of software is followed by an upgrade. For example, consider the case of a mobile phone that implements a common standard such as GSM. After deploying the handset the manufacturer realises that some aspect of the software is not fully standards compliant and taking advantage of the software radio capability posts new software for download which the handsets automatically retrieve. It is inevitable that software upgrades cannot propagate instantaneously to all handsets due to bandwidth and the high probability that some handsets will be powered off. Consequently, the radio system must have a mechanism to deal with handsets having different versions of components of the software radio system in order to ensure that the operation of the network is not comprised.

One way to overcome these problems is to employ a versioning system especially designed to maximise communication. This type of system would force terminals to upgrade software when appropriate. It could also manage compatibility issues in providing information about which components are valid combinations. Component sets could be validated for compatibility by checking their versions. Also, security could be involved in that communication is only allowed if the terminal uses particular software versions.

Although rapid reconfiguration has obvious advantages, these examples discussed illustrate that if software versioning and upgrading are required then an infrastructure needs to be in place to avoid these errors. This has been acknowledged in the literature, particularly in the development of software download for mobile phones [Bucknell2002]. These systems have more control over the software a terminal uses for communication in that the terminal is constantly connected to a base station which can trigger various forms of software download. This does not however deal with more distributed approaches in applications such as ad hoc networking where there is no central infrastructure. Chapter 7 addresses these issues in more detail by discussing case studies in both software download and wireless networking.

4.4 Developing a Reconfigurable Radio

4.4.1 System Design Considerations

There are various ways that the development of a reconfigurable radio system can be approached. Software radio implementations such as the C++ based PSpectra system are based around the use of a class library that offers common signal processing classes that can be reused to form a

software radio application. This results in separate executable programs, one for each radio implementation. Although this approach works well there are certain limitations, in particular when considering reconfiguration. While these libraries support constructs for binding together reusable classes to form radio implementations, they have not specifically included support for reconfiguration. Whereas application reconfiguration is accomplished via separate standalone executables, both structural and parametric reconfiguration requires the programmer to implement separate code for each application. Thus, each standalone executable is implemented in a different way to form the software radio.

An alternative approach that supports reconfiguration better is to factor out as many domain-specific operations as possible into a software framework. This removes the need for each programmer to re-implement the same constructs for each software radio application (see Figure 4.3). Using this approach the radio is not a standalone executable that interacts with the operating system but a radio configuration used to configure a component framework. This design contains all the information required to build the radio system including signal processing parameters, structural designs and any additional code not covered by this domain-specific framework. By inherently supporting application, structural and parametric reconfiguration in the component framework itself, it becomes much simpler to develop a reconfigurable radio.

This approach is quite different to say a DSP processor platform. The DSP processor provides an efficient processor for executing signal processing algorithms, however it does not dictate any particular constructs or style for the structure of the software. Developers are free to manipulate the capabilities of the device in any way they see fit. Also, software is typically developed in assembly language and C, but these languages themselves do not dictate any type of software design. The difference with the framework approach is that it explicitly dictates how the software should be constructed with the aim of improving the quality of the system as a whole.

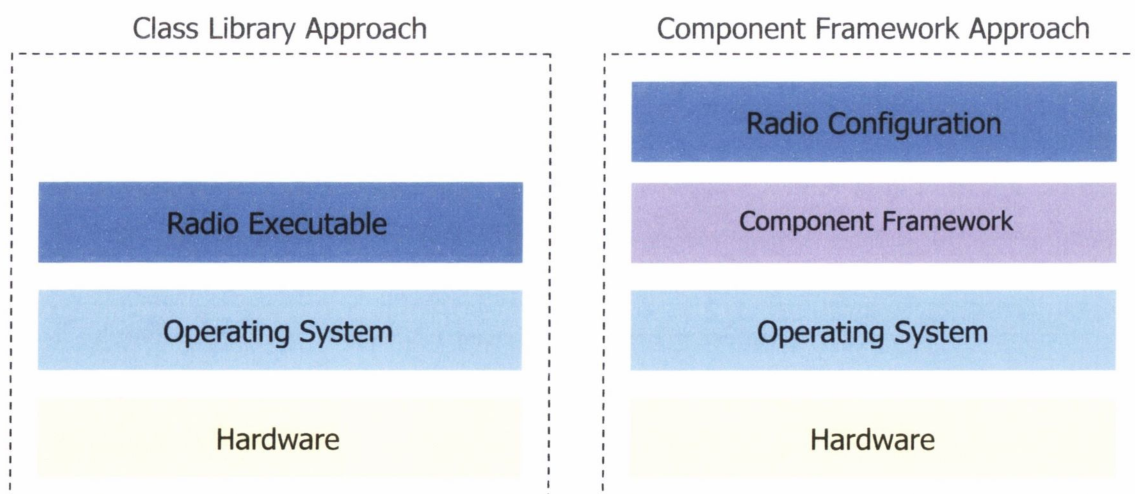


Figure 4.3 – Different Approaches to Reconfigurable Radio System Design

In terms of the signal processing capability of the reconfigurable radio, the framework must be extensible in that it allows any type of signal processing functionality to be used in the system. This signal processing functionality should be incorporated in such a way that allows it to be easily used by different radio configurations. The framework should also be extensible in that it allows new signal processing functionality to be added easily, but also allows radio configurations to include additional code that can interact with signal processing algorithms. Thus, the framework needs interfaces and constructs that allow this type of integration to occur easily.

4.4.2 Enabling Reconfiguration

As discussed in Section 4.2.2, the core requirement of the reconfigurable radio is the ability to enable application, structural and parametric reconfiguration. These facets of the reconfigurable radio are the enabling techniques to a whole host of new software radio applications. Enabling these applications requires the framework to both inherently support each type of reconfiguration and to enable monitoring and control. The following sections discuss how this can be achieved.

Enabling Application Reconfiguration

Application reconfiguration will allow the framework to change the type of radio system it implements (for example QPSK transceiver, GPS receiver, TV transmitter, etc). Using the component framework approach, each radio application is differentiated by the configuration passed to the framework. This configuration must specify all the details required to implement the radio design. The following elements are required in the configuration:

- Signal Processing – The configuration must specify which signal processing algorithms should be used to implement the design.
- Structure – The configuration must specify how the signal processing algorithms should be combined together to form the reconfigurable radio.
- Parameters – The configuration should include the parameters that configure the operation of signal processing algorithms, for example frequency settings or filter taps.
- Code – The configuration should include any code required to implement specific operations for the radio application.
- Packaging – The configuration should be packaged into a unit which can be easily deployed to reconfigurable radio devices.
- Information – The configuration should include informational data about itself and its purpose, possibly allowing a user or other software agent to decide whether to use the application.

Enabling Structural Reconfiguration

In enabling structural reconfiguration the infrastructure needs to be implemented in a flexible way that allows the structure of the radio to be changed. For structural reconfiguration to be feasible the software of the reconfigurable radio has to be inherently built with this feature in mind. Also, the adaptability and flexible nature of signal processing algorithms should facilitate this.

Structural reconfiguration can be implemented in various ways. A simple approach is to use offline reconfiguration in that the software radio infrastructure supports the creation of different types of structures. For example, if a software radio implementation consists of filters, mixers and modulators then the software radio infrastructure should facilitate the combination of these elements in any order.

Of more interest in this thesis is dynamic reconfiguration which is structural reconfiguration occurring while the device is operational. Enabling this type of reconfiguration requires a more sophisticated approach. Specifically the infrastructure has to maintain the integrity of the radio application during the reconfiguration process. The system must ensure that new configurations are valid and do not cause the system to become unstable. Also, where possible the system should attempt to continue operation during the reconfiguration process. This will only be possible in cases where reconfiguration does not drastically change the functionality of the radio.

In changing the structure of the radio, the infrastructure should inherently support software download. This will allow new functionality to be downloaded and integrated into the structure of the radio without having to alter, recompile or stop the radio system.

To facilitate the changing of the radio's structure, the infrastructure should expose an API (Application Programming Interface) that allows programmers to write code that can alter the structure. This should include methods to edit the configuration via adding, removing or changing the order of signal processing algorithms.

Enabling Parametric Reconfiguration

Parametric reconfiguration will primarily be concerned with allowing the parameters of signal processing algorithms to be changed thus enabling reconfiguration. This requires an infrastructure that facilitates the exposure of parameters. Thus, each signal processing algorithm will use different parameters, exposed in a consistent way via a standardised parameter interface. This interface should provide all the functionality to allow any type of parameter to be read or changed. Also, the structure of the algorithms themselves must be able to cope with changing parameters, for example

if a frequency setting is changed a lookup table may have to be recalculated. The infrastructure should inherently support mechanisms for enabling all of these tasks.

Monitoring and Control of Reconfigurability

The three types of reconfiguration discussed offer extremely flexible radio systems, however they are useless without some way to control and monitor their use. As discussed in the previous section a framework is required that hosts the radio system, provide explicit design rules and allow control of the radio system as a whole. This framework can provide information about the radio system by allowing the monitoring of system functionality such as viewing signals. It can also provide external control functions by exposing a control interface. This can be used by other software systems that use a reconfigurable radio as a sub-system.

4.5 Summary

This chapter has analysed all the issues surrounding the development of software for software radio systems. The term 'reconfigurable radio' has been defined to differentiate the approach taken in this thesis from others. Reconfigurability has been analysed and broken down into the three categories of reconfigurability; application, structural and parametric. These categories allow the level of reconfigurability of a device to be assessed and provide useful guidelines for determining the requirements of a reconfigurable radio system. The next chapter presents the design of a reconfigurable radio system that is built using component-based software and features the three categories of reconfigurability.

5 The IRIS Reconfigurable Radio

5.1 Introduction

This chapter describes the design of IRIS (Implementing Radio In Software) [Mackenzie2002b, Doyle2002a, Mackenzie2003]. Sections 5.2 and 5.3 provide a high level overview of the IRIS system. Section 5.4 discusses the component-based approach taken in designing radio components and how they can be defined in software. Section 5.5 discusses the component framework used to compose these components together to form a reconfigurable radio system. Section 5.6 discusses control logic, a mechanism provided by the IRIS architecture for defining the inter-relationships between components. Finally to demonstrate how components, the component-framework and control logic fit together, Section 5.7 provides a worked example of developing an FSK transceiver using IRIS.

5.2 IRIS Overview

IRIS has been built to demonstrate the concepts of reconfigurability as discussed in the previous chapter. The purpose of IRIS is to both demonstrate this through a practical example and from this to gain insight into the problem of developing software for reconfigurable radio systems.

IRIS is a component framework designed to run on GPPs. Signal processing components are written in C++ and each component implements a generic signal processing algorithm or encapsulates some other sub-system such as a hardware device. Radio systems are created by both instructing the component framework to assemble components in a particular way, and by defining the interrelationships between instances of components. The IRIS system is highly structured and the mechanisms for building radio systems are well defined within the IRIS architecture. Basic radio systems can be built by combining existing components. More complex designs can be addressed by writing new components and writing control logic, essentially application-specific code that defines the interaction among a particular set of components in the radio system. The IRIS system uses XML as a configuration mechanism and control logic can be written in either C++ or Java.

Figures 5.1 to 5.3 give an indication of what is possible with IRIS. The diagram in Figure 5.1 demonstrates how both transmitter and receiver architectures are specified via the same generic configuration mechanism. IRIS uses the same type of configuration mechanism to realise every type of radio system. For instance, it does not constrain radio system design via entities such as receiver, transmitter, transceiver, etc; each radio system is made from generic components. This approach is quite different to other approaches such as the JTRS SCA which defines concrete interfaces for every element of the radio system (see Chapter 2, Section 2.5.2).

These two examples demonstrate the level at which IRIS addresses the development of radio systems, the DSP level. The development of an air interface can often be intertwined with other aspects of the system in particular other elements of the protocol stack. Often (as in the JTRS) the DSP of the radio system is closely coupled to networking features such as the MAC (Medium Access Control) or Data Link layers of the protocol stack. While IRIS can be used in this context also (and this will be demonstrated in Chapter 7), its primary function is to facilitate the construction of the DSP systems of a reconfigurable radio.

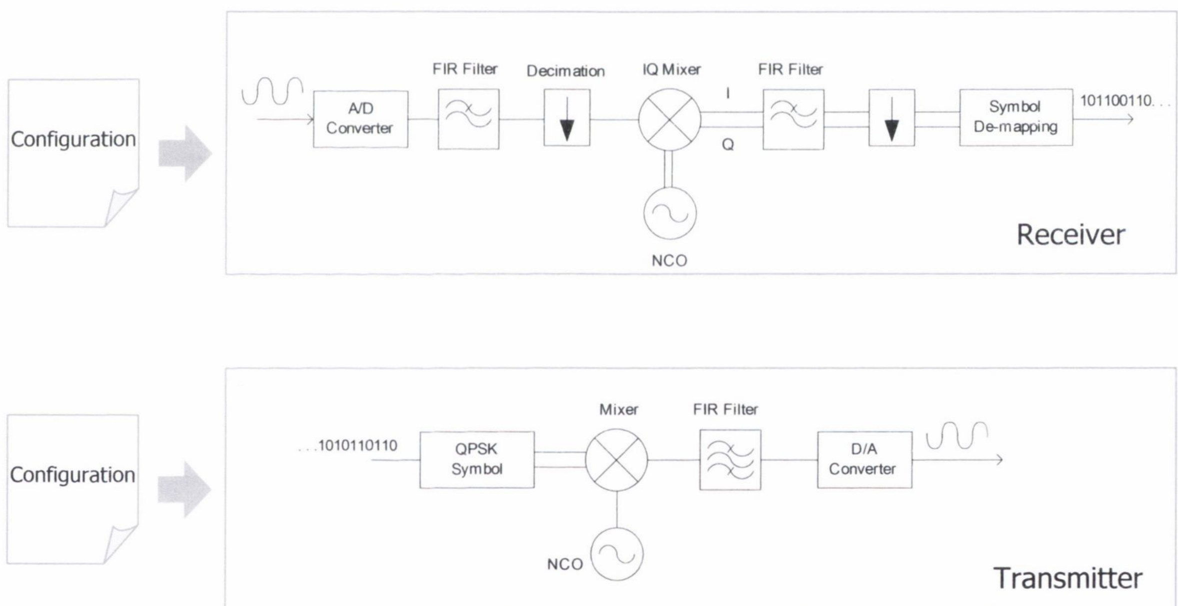


Figure 5.1 – Receiver and Transmitter Example

While the examples in Figure 5.1 illustrate more typical software radio applications, the example in Figure 5.2 demonstrates how the IRIS system goes further. In this example, a similar configuration is used to create a radio but it also includes functionality for dynamically reconfiguring both parameters and the structure of the radio. This allows the creation of truly dynamic designs in which the radio can change its functionality at runtime as desired. The uniqueness of this approach is that support for application, structural and parametric reconfigurability is inherently built into the system and handled by the IRIS Component Framework.

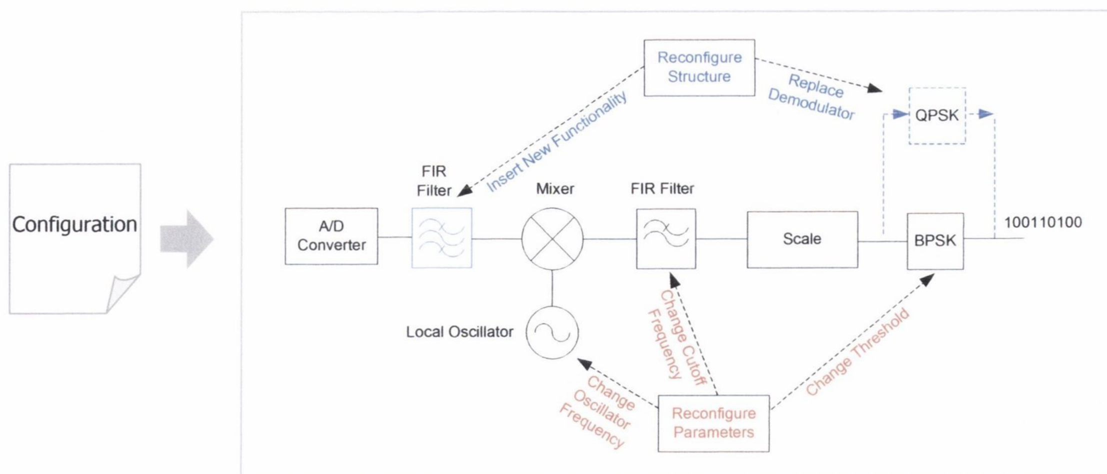


Figure 5.2 – A Reconfigurable Radio System

The final example in Figure 5.3 demonstrates how the IRIS system can facilitate an environment for experimentation and rapid development of radio systems. In this example a radio configuration is used to create a test scenario for experimenting with the effects of adding noise to a FSK (Frequency Shift Keying) signal. The configuration not only specifies the structure of the radio architecture but a user interface that allows dynamic user interaction with the system. The IRIS architecture inherently supports this type of functionality.

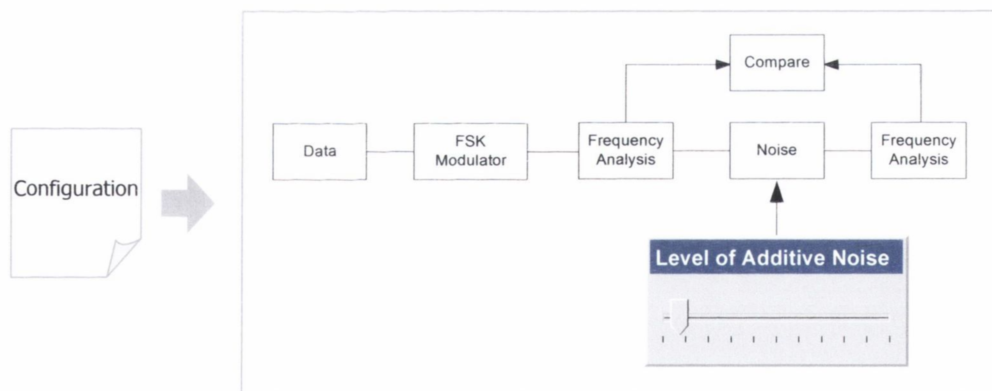


Figure 5.3 – A Reconfigurable Radio with User Interaction

5.3 IRIS Architecture

Following a large amount of experimentation with software architectures and software design, an architecture for IRIS was created. (The term architecture in this context refers to the definition discussed in Section 3.4.2; an architecture being a superset of principles prevailing a system design.) Figure 5.4 illustrates the IRIS architecture and introduces all the main entities involved in

its design. In addition to defining the general paradigms of the system, the IRIS architecture consists of a Component Framework, a component model and rules for creating control logic and radio configurations.

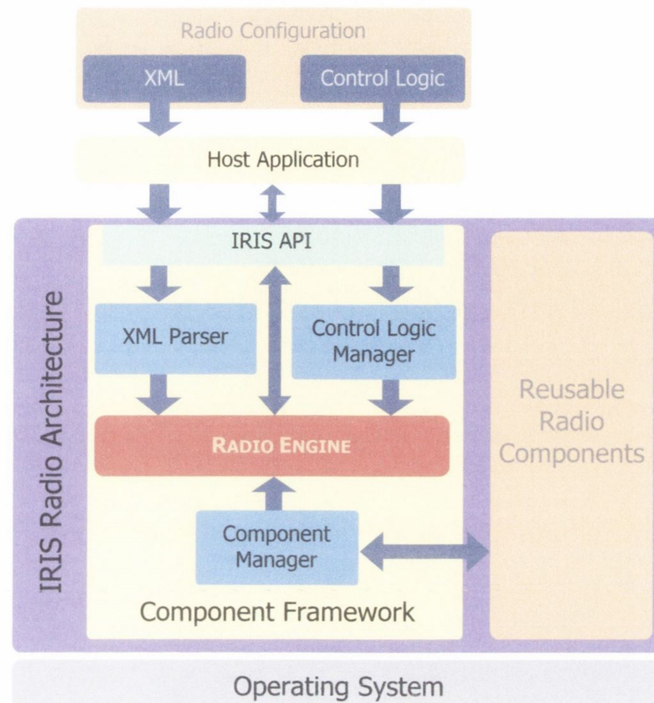


Figure 5.4 – The IRIS Radio Architecture

Each of the entities in the IRIS architecture has been designed to address the reconfigurability issues as addressed in Chapter 4. The following sections describe each of the entities in detail.

Note: In the remainder of the thesis capital letters will be used to denote the entities of the IRIS system, e.g. Radio Component, Component Framework and Control Logic

5.4 Radio Components

The fundamental unit for building reconfigurable radios in the IRIS Radio Architecture is the Radio Component. A user of IRIS creates a radio from existing Radio Components or by creating new components when necessary. The Component Framework (discussed in the next section) is used to chain Radio Components together to create the actual reconfigurable radio.

There were a number of challenges in designing the Radio Component. It was necessary to design the Radio Components to encapsulate radio functionality in a way that would facilitate their reuse among many applications. It was also necessary to develop a design that allowed ultimate flexibility throughout the system. Overall, it was necessary to create Radio Components that could

facilitate application, structural and parametric reconfiguration as discussed in the previous chapter. The resulting component design is described in the following sections.

5.4.1 Component Granularity and Component Types

Before looking at the actual structure of a Radio Component it is useful to describe component granularities and component types.

The granularity of Radio Components has been designed as discussed in Section 4.3.1; for example, each Radio Component implements operations at the granularity of FM modulators, QPSK symbol detectors and FIR Filters, i.e. the functional 'parts' of a radio system. It should be noted that although this is the approach taken in this work, the system itself does not constrain the user to a particular granularity. The designer is free to implement components in smaller or larger granularities if required; however, this thesis argues that these granularities are unsuitable for reconfigurable radio. This is because radio systems are inherently built in sub-sections that are easily identifiable. For example, the common Viterbi decoder is a reusable algorithm and therefore an ideal candidate for a single component. A Viterbi decoder is also made up of many adders and multipliers, yet it would not make sense to package these elements into individual components. If adder and multiplier components were built and subsequently connected together to form a Viterbi decoder, the algorithm would no longer be encapsulated in a component but would exist in the interconnection between these components. This approach would contradicy the component principles discussed in Chapter 3, Section 3.3 that require a component to be a self-contained independent unit of deployment.

Another important aspect of the Radio Component is visibility of its internal implementation. The Radio Component has been designed to use a black box abstraction. This means that all the internals of how the component works are hidden from the user of the component. The only way the component can be used is via the standardised interfaces it provides.

IRIS must support a multitude of different radio configurations. The majority of functions performed in a reconfigurable radio are DSP related, however there is other functionality that needs to be addressed such as how to input or output data, and how to interface and control hardware. In the IRIS architecture different radio functions are categorised by the following three types:

1. DSP components
2. Input/Output (IO) components
3. Standalone components

DSP components allow signal processing functionality to be encapsulated into a component. IO components are identical to DSP components but have extra constructs for supporting the input and

output of data into the signal processing chain. Standalone components satisfy the need to have additional functionality that is required by the radio, but separate from the signal processing chain; for example, controlling external hardware or implementing timers. The three basic types are formed by inheriting from the abstract class '`RadioComponent`' (see
Figure 5.5).

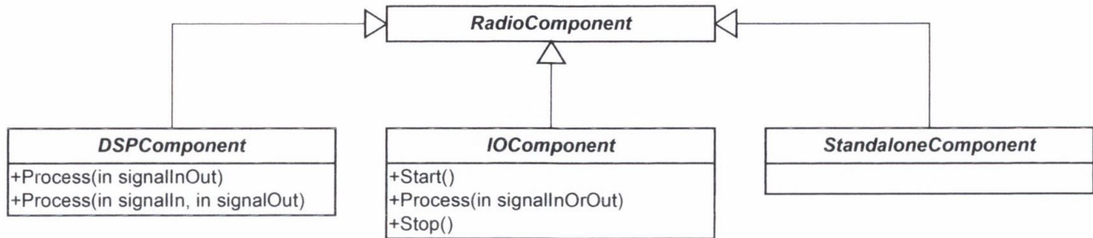


Figure 5.5 – Relationship of Component Types

5.4.2 Component Interfaces

An external view of the Radio Component helps to illustrate how the component is used in creating a reconfigurable radio. Externally a Radio Component can be viewed as shown in Figure 5.6. The Radio Component exposes a set of well-defined interfaces, which allow other entities in the IRIS architecture (such as the Radio Component Framework) to interact with each component using the same standardised pattern.

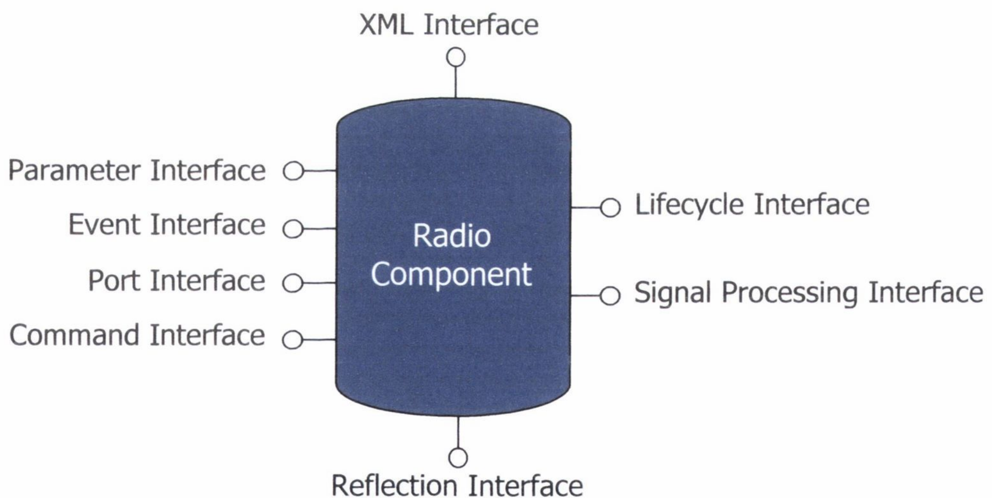


Figure 5.6 – External View of a Radio Component

Each Radio Component implements a set of interfaces each addressing a different requirement. Separate interfaces ensure that suitable cohesion is enforced in the component. The Radio Component interfaces have been carefully chosen to address the various ways in which components can be composed together in a component framework.

The basic interface supported by DSP and IO components is the signal processing interface. Figure 5.5 shows the `Process()` method that components must implement to consume and produce digital signals. The `Process()` method is discussed is detail with relevant examples in the next Chapter, Section 6.2.6. In addition to a signal processing interface, the Radio Component supports the seven different interfaces which the Component Framework uses to control and interact with the Radio Component, namely; Lifecycle Interface, Parameter Interface, Event Interface, Port Interface, Command Interface, Reflection Interface and Component Information Interface.

Lifecycle Interface (Figure 5.7): This interface exposes all the functionality for controlling the lifecycle of a Radio Component and its function is to allow for initialisation and cleanup of a component. While these methods represent the basic lifecycle of a component, different component types add additional steps to the lifecycle of a component.

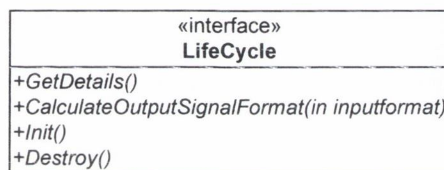


Figure 5.7 – Lifecycle Interface

Parameter Interface (Figure 5.8): The parameter interface allows the user of a component to configure and reconfigure the operation of the component throughout its lifecycle. Each component exposes a set of parameters that define its behaviour and parameters can have any data type. The parameter interface allows access to these parameters in a generic way.

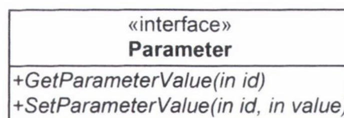


Figure 5.8 – Parameter Interface

Event Interface (Figure 5.9): Components can fire events to asynchronously inform external clients of occurrences during the lifecycle of the component. A component can support any number of events and various types of data can be passed with events. The event interface allows external clients to subscribe to any event that the component publishes.

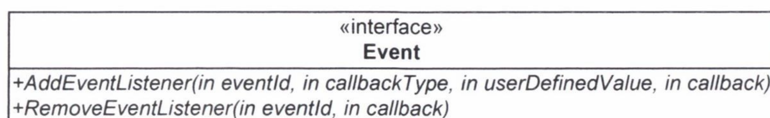


Figure 5.9 – Event Interface

Port Interface (Figure 5.10): Ports are inputs into a component. This interface allows external clients to asynchronously pass data to a component for processing. Ports are provided to differentiate the processing of data from that of digital signals.

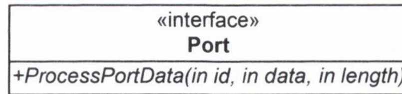


Figure 5.10 – Port Interface

Command Interface (Figure 5.11): Commands allow external clients to issue asynchronous commands to a component. This provides a generic mechanism for exposing common DSP functionality, examples being: 'reset synchroniser', 'recalculate lookup table' or 'cease carrier'. While parameters could be used to implement this type of functionality, commands provide a useful way to separate out more numerically based parameters from function-based operations.

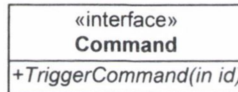


Figure 5.11 – Command Interface

Reflection Interfaces (Figure 5.12): The reflection interface allows external clients to query information about a component programmatically. External clients can query any information about the type of component, the parameters, events, ports and commands it supports and general-purpose information about the component such as author, version, etc.

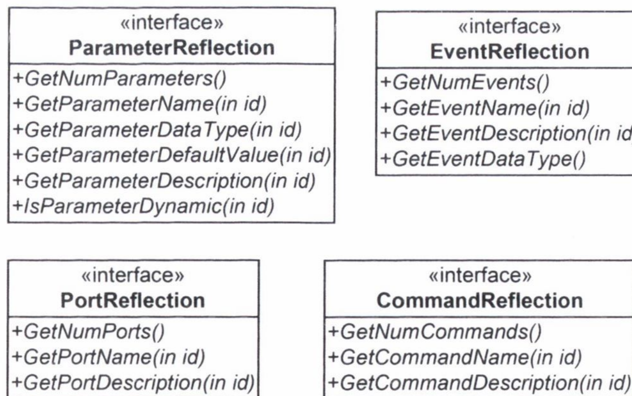


Figure 5.12 – Reflection Interfaces

Component Information Interface (Figure 5.13): This information provides information about the component itself and can be used for dynamic and automatic discovery of details about components. In addition to information such as name, author, version, etc, the component information exposes two methods offering XML descriptions of the component. The first method is

an XML interface which exposes an example configuration. This provides a sample of XML to a client wishing to know how the component can be configured. The second method provides XML indicating the capabilities of the component including all the information supported by the reflection information. This XML facility is of use both during system-design and during automatic reconfiguration in that systems can be built that can automatically use a component without user intervention.

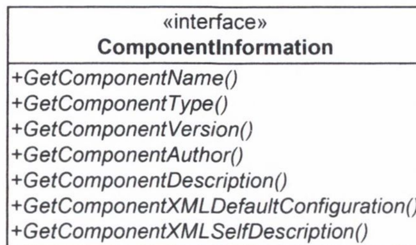


Figure 5.13 – Component Information Interface

Once a class implements all these interfaces, it can be used as a black box Radio Component within the Component Framework (see Figure 5.14).

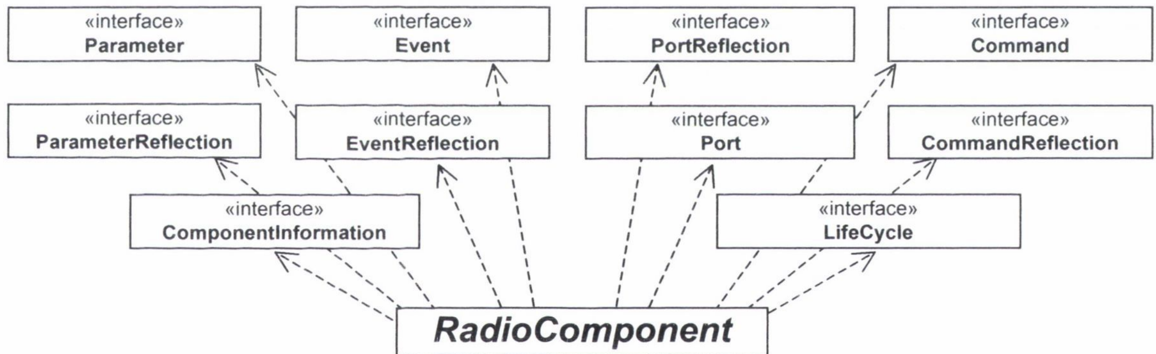


Figure 5.14 – Abstract RadioComponent class

5.4.3 Component Lifecycle

An important aspect of the Radio Component is its lifecycle, i.e. the pattern by which the component is used. The challenge in designing the lifecycle is to support enough functionality so that any aspect of a reconfigurable radio can be developed but exposed in a generic way. A suitable lifecycle has been designed for Radio Components and it consists of seven stages:

1. Loading
2. Initialisation
3. Starting
4. Processing

5. Stopping
6. Cleanup
7. Unloading

Loading: During this stage the component is loaded for use. This will include any retrieval of components and any instantiation of classes. Once this stage has completed the component must be available and ready for use.

Initialisation: During this stage the component is primed with all the information required for operation. The first step tells the component what parameters it should use and this is performed by repeated calls to `SetParameterValue()`. In the assembly of components into a working radio system, many different types of signal will be used so it is important that the framework can work out if a configuration is valid. The `GetDetails`() call provides the framework with the details required to work out if a component is suitable for inclusion in a radio design. Next, a call to `CalculateOutputSignalFormat()` tells the Radio Component what type of input signal it will be receiving and thus the component can work out what output signal it will produce for the given input. The final step in initialisation is a call to `Init()`. This allows the component to perform all other initialisation such as allocating memory, etc and also give the component an opportunity to reject the configuration it has been initialised with if it detects an error. For example, if the component does not support a particular data type or has been initialised with incorrect data the initialisation method can return false to indicate this error.

(IO only) Starting: Indicates to an IO component that input should commence.

(DSP and IO only) Processing: In this stage the framework repeatedly calls the `Process()` method causing the Radio Component to perform its actual processing. This can be any operation that the Radio Component supports but mostly DSP components will perform signal processing, and IO components will perform output/input data to/from hardware. During the process stage the component can fire events that occur in the course of processing. During this phase the component will also receive asynchronous method calls from the framework when values have been reconfigured, when commands have been issued and when data ports have received data. The developer of the component can decide how best to react to these asynchronous methods according to the context of a particular component.

(IO only) Stopping: Indicates to an IO component that input should cease.

Cleanup: At this stage `Destroy()` is called which allows the component to free resources.

Unloading: This stage involves the deletion of the component instance and unloading of component code.

Figure 5.15 shows a UML sequence diagram depicting the lifecycles of a DSP and an IO component. The differences between the two lifecycles can be seen in this diagram. IO components offer a `Start()` and `Stop()` method in addition to a `Process()` method. The `Start()` and `Stop()` methods tell a component to cease input or output. These are required as IO components usually have a great impact on the flow of signals between components. All signals are ultimately input and output via IO components therefore these methods allow the flow of signals to be controlled in the radio. The `Process()` method allows an IO component to either input or output data.

Unlike IO components, a DSP component has two `Process()` methods. The two methods differ in the way signals are processed by the component. One method is for processing signals in place, in that both the input and output of the component are read and written to the same memory location. The second type uses separate memory locations for both input and output. Memory conservation is an important factor in software radio design so the in-place method was designed to allow the developer to conserve the amount of memory used in the system. In some circumstances it can also reduce the amount of memory copying required in a component. The not-in-place method is provided so that memory copying can be reduced, as sometimes the in-place method requires data to be copied to a temporary location before processing. By providing separate inputs and outputs it is possible to avoid this copying. Overall these methods provide enough flexibility for the programmer to write efficient and simplified Radio Components.

The sequence diagram for the lifecycle of a standalone component is shown in Figure 5.16. Standalone components do not interact with signals and thus have less functionality. While all other components are processing signals, the standalone component can perform other functionality in response to changing parameters, commands or ports. Like all components, standalone components can fire events.

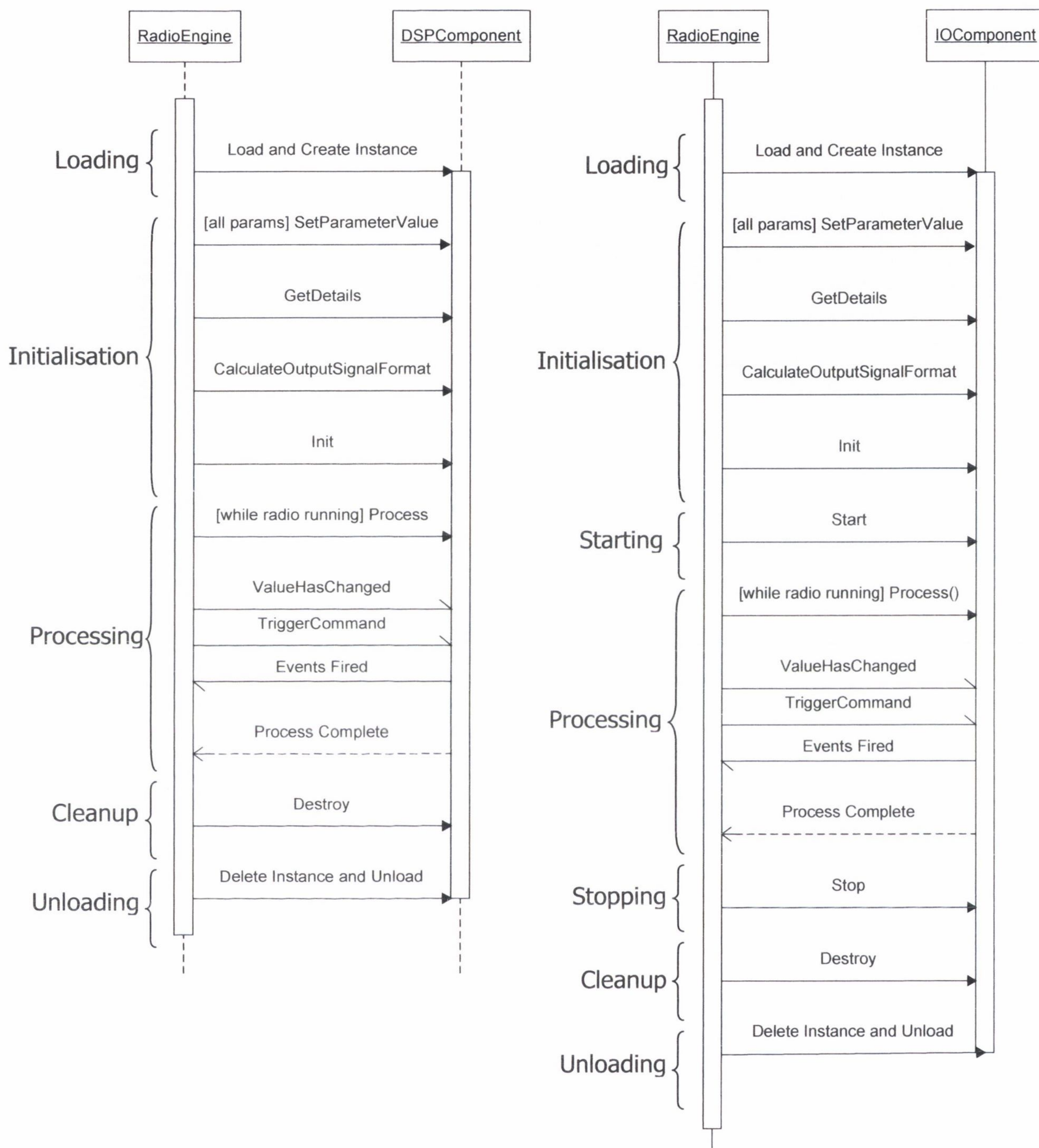


Figure 5.15 – Sequence Diagrams of DSP and IO Component Lifecycles

Standalone components do not offer any new lifecycle methods to the standard component and thus are the most basic type of component. Standalone components operate through the use of parameters, events, ports and commands.

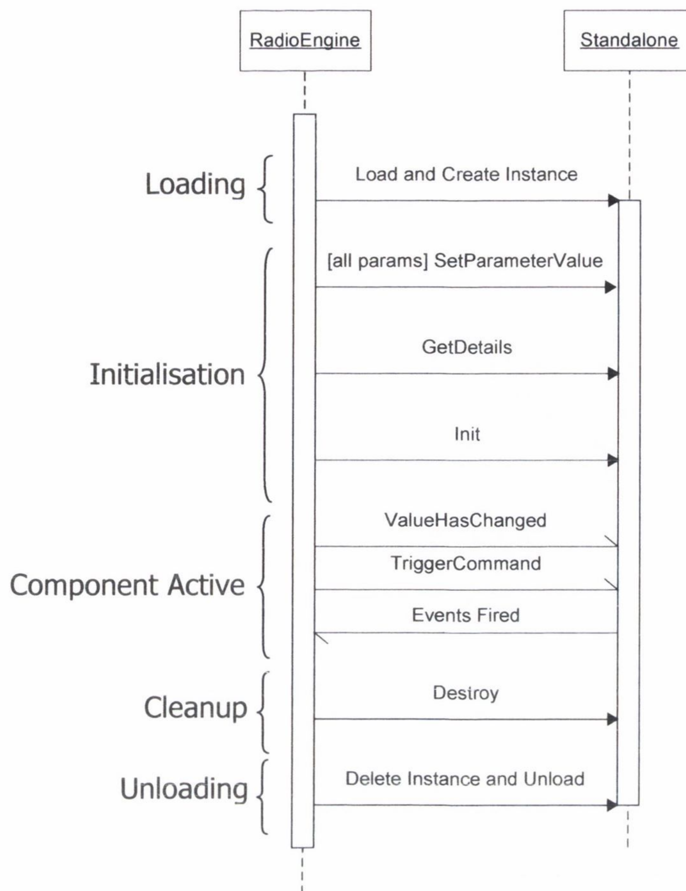


Figure 5.16 – Sequence Diagram of Standalone Component Lifecycle

5.4.4 Discussion

From the list of interfaces discussed in the previous section it becomes apparent that a significant amount of code must be written to implement each interface. The reason so many interfaces are required is to make up for the lack of reflection in some programming languages, in this case C++. As discussed in Section 3.3.3, reflection allows code to query the capabilities of other compiled code dynamically and facilitates meta-data. Languages like Java and C# inherently support reflection and thus a binary executable from these languages can be queried to find out what methods and member variables is exposed by the code. C# (and Microsoft's .NET platform in general) goes a step further than Java in that it inherently supports attributes (also known as declarative constructs), pieces of data that can be included within a programme itself [Liberty2001]. Using reflection a C# programme can read its own attributes.

Reflection information is required by the IRIS system because components have to be loaded dynamically and used at runtime. This reflection information allows the framework to query the component as to its capabilities, the data types it supports and provides access to additional meta-

data such as the component's name, version, description and documentation. For this to happen the framework has to be able to query information dynamically about the component, from simple information such as the component's name to more complicated information such as the parameters and events that a component supports. The latter is important when considering graphical applications and visualisation of radio systems. Using reflection a user interface can query information about all components in a generic away allowing the details of a component to be displayed. This allows a user to graphically build radio systems without having to write code.

To overcome this problem the process of writing most of these interface methods has been automated via a scripting language. This language will be covered in the next chapter (Section 6.2.3), but the overall effect of using this technique is that the programmer only has to implement a minimal amount of code to programme a Radio Component. This is illustrated in Figure 5.17. By using the scripting language the only interfaces the programmer has to implement are the lifecycle and signal processing interfaces. The remainder of the interfaces are automatically generated.

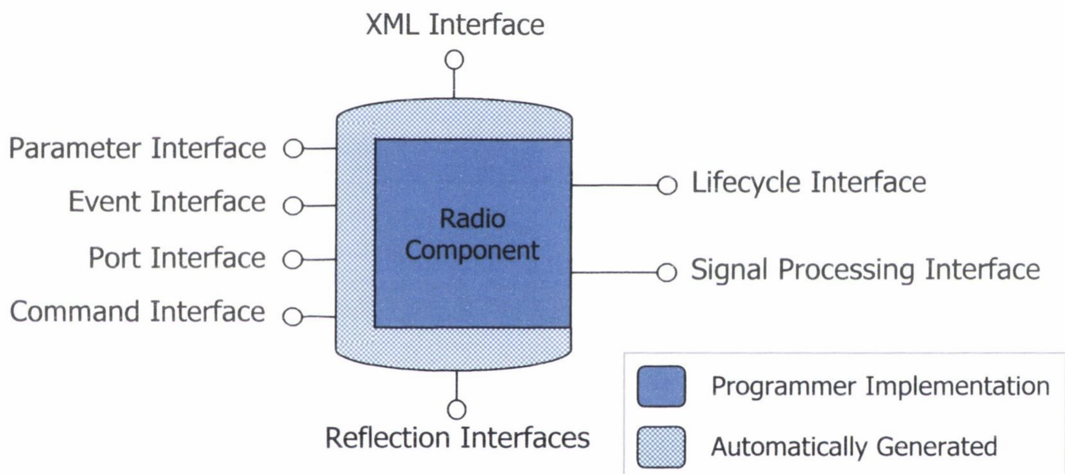


Figure 5.17 – Radio Component Showing Interfaces Implemented by Code Generation

In addition to implementing various interfaces, the abstract `RadioComponent` class provides several methods that allow the component implementation to interact generically with external users of the component (see Figure 5.18). For example, by overriding the `ValueChanged()` method, the programmer can be notified when a parameter has been changed externally. Likewise `CommandWasTriggered()` indicates to a component that external control logic issued a command to the component and `ProcessPortData()` indicates that data was received into a port of the component. These facilities maintain the black box abstraction that allows Radio Components to be used generically.

RadioComponent

#ValueHasChanged(in parameterId)
#CommandWasTriggered(in commandId)
#ActivateEvent(in eventId, in data)
#ProcessPortData(in portId, in data, in length)
#LogInfo(in text)
#LogError()
#LogWarning(in text)
#FatalStop()
+GetOutputSignalFormat()
+GetInputSignalFormat()

Figure 5.18 – RadioComponent

For firing events the `RadioComponent` class offers the method `ActivateEvent()` which allows a component to generically notify any number of external subscribers during processing. The scripting language automatically generates generic code that allows the Component Framework to interact with the Radio Component in this way. `RadioComponent` also supports a variety of support methods for logging information and errors, and for querying information about the signals it will be receiving from the Component Framework.

Implementing DSP and IO components requires the programmer to write signal processing code and this is written in the `Process()` method. The engine repeatedly calls a component's `Process()` method providing it with memory locations for reading and writing samples. This technique decouples components as each component does not require knowledge of the other components in the reconfigurable radio. Components are thus passive, only performing processing when called on to do so.

The other methods left for implementation by the programmer are mostly to satisfy the lifecycle of a particular component type. The programmer must implement `GetDetails()` and `CalculateOutputSignalFormat()` to allow external clients to query information about how a component plans to process data. The `GetDetails()` method allows a component to specify the data types it can accept and whether or not it processes data in-place. `CalculateOutputSignalFormat()` allows an external client to figure out what block size and sampling rate it can expect as an output from the component for a given input. The programmer can use the calls from `Init()` and `Destroy()` to perform pre and post steps to processing. IO components can avail of calls to `Start()` and `Stop()` to control the input and output of data.

5.5 Component Framework

The IRIS Radio Component Framework is an infrastructure that allows Radio Components to be composed together to form a reconfigurable radio. The diagram of the IRIS architecture is reproduced here in Figure 5.19. This diagram shows the role of the Component Framework in the architecture. The framework is the core of the architecture consisting of the sub-systems required to build a reconfigurable radio. It consists of the Radio Engine, Component Manager, Control Logic Manager, XML parser and the IRIS API:

- *Radio Engine:* The Radio Engine implements different radio configurations and is the core of the Component Framework. The Radio Engine brings together Radio Components and Control Logic to implement the radio design and controls their interaction.

- *XML Parser:* The XML parser reads XML radio configurations, verifies their content and converts them to an internal representation of a radio design that can be implemented by the Radio Engine.

- *Component Manager:* The Component Manager is responsible for loading and unloading Radio Components from the framework. The component manager can load components from a variety of locations (e.g. local file system or internet) and present them in a generic form for use by the Radio Engine.

- *Control Logic Manager:* The Control Logic Manager loads and unloads various types of Control Logic for use by the Radio Engine. Control logic can be implemented in potentially any language (currently C++ and Java are supported) so the manager must present each of these control logic types in a generic way for use by the engine.

- *IRIS API:* The IRIS API is provided to allow the Component Framework to be integrated into other applications. The API abstracts the particulars of the Component Framework, components and control logic from the user of the API providing a simple interface for the construction of reconfigurable radios

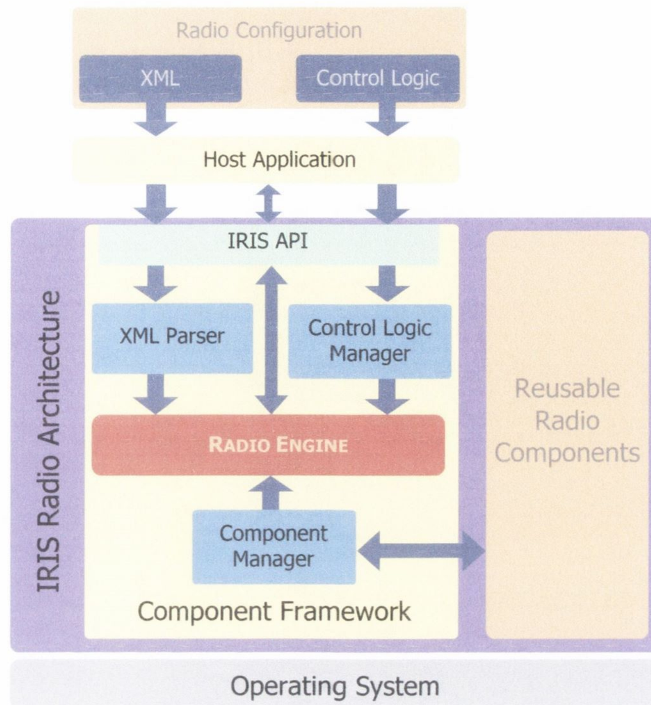


Figure 5.19 – The IRIS Radio Architecture

The operation of the Component Framework is illustrated by the flow diagram shown in Figure 5.20. This diagram shows that there are three main stages a radio system goes through; initialisation, running and cleanup. In the initialisation stage the XML configuration is read, verified and converted to an internal structure. The Radio Engine uses the internal structure in conjunction with the Component Manager and Control Logic Manager to build the radio system. During the running stage the engine controls the movement of signals through the components of the radio system. Also, Control Logic can respond to events from the components and reconfigure any aspect of the system. Finally, at the cleanup stage all the resources used by the system are released.

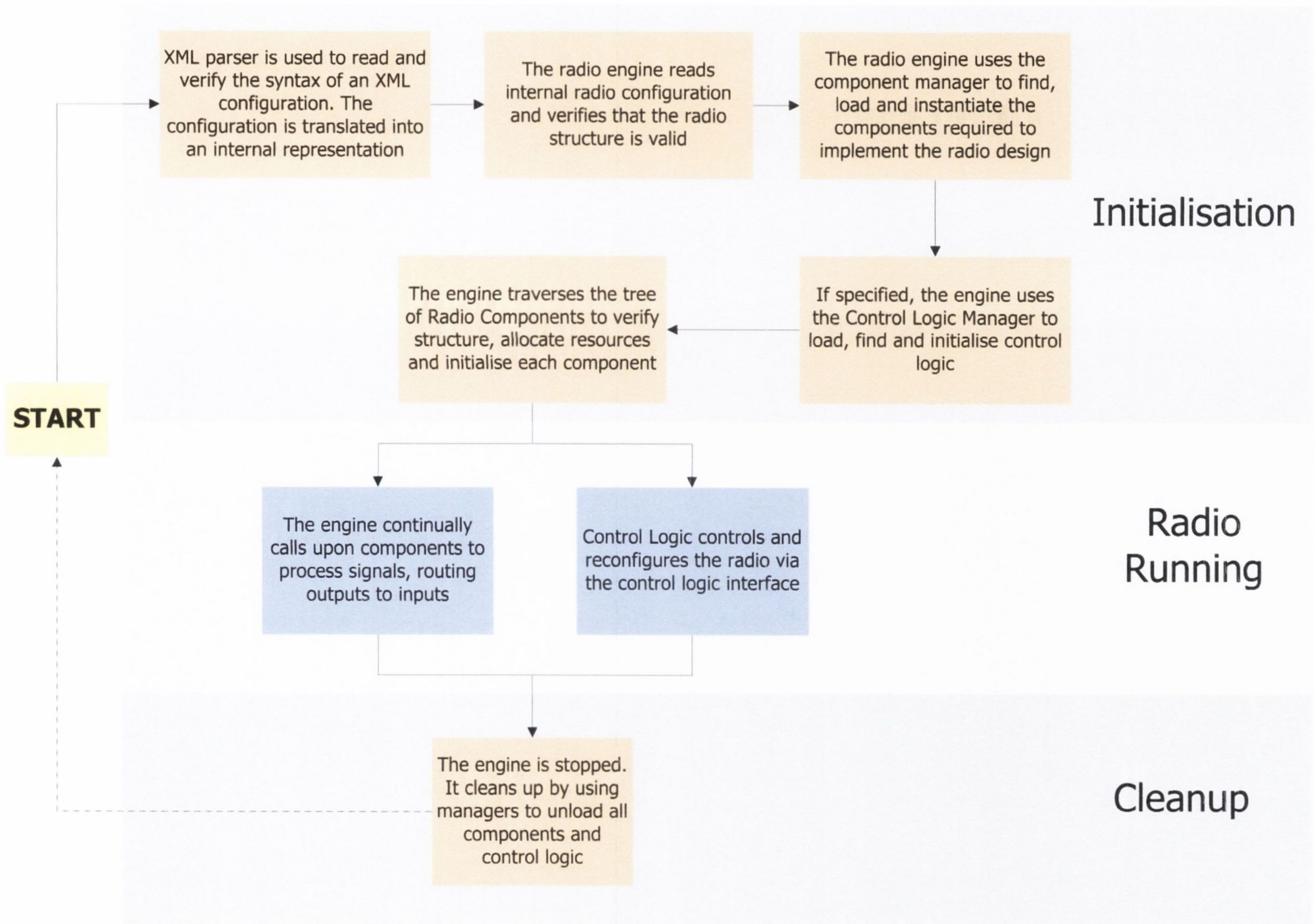


Figure 5.20 – Flow Diagram for Creating a Reconfigurable Radio

5.5.1 Radio Engine

The Radio Engine is the core of the Radio Component Framework. The engine is responsible for assembling a reconfigurable radio from a set of Radio Components. To achieve this it must be possible to define a radio configuration that shows how Radio Components can be fitted together thus allowing the Radio Engine to translate this into a working radio system. It was decided to use XML to define the radio configuration as it allows the representation of hierarchical data and thus was suitable for defining the structure of a radio system.

The XML file defines three things:

1. Components: The list of components required in the radio system along with values for configuring each component in a particular way.
2. Control Logic: Details about control logic. Control Logic is additional code written by the radio designer to control components and to provide a generic way to allow interaction among components. Control Logic is application specific in that each type of radio system will have a different Control Logic implementation. Control Logic is discussed in detail in Section 5.6.
3. Documentation: Details about the radio system being created, i.e. radio system name, description and version.

The interaction of components, XML configuration and Control Logic are illustrated in Figure 5.21.

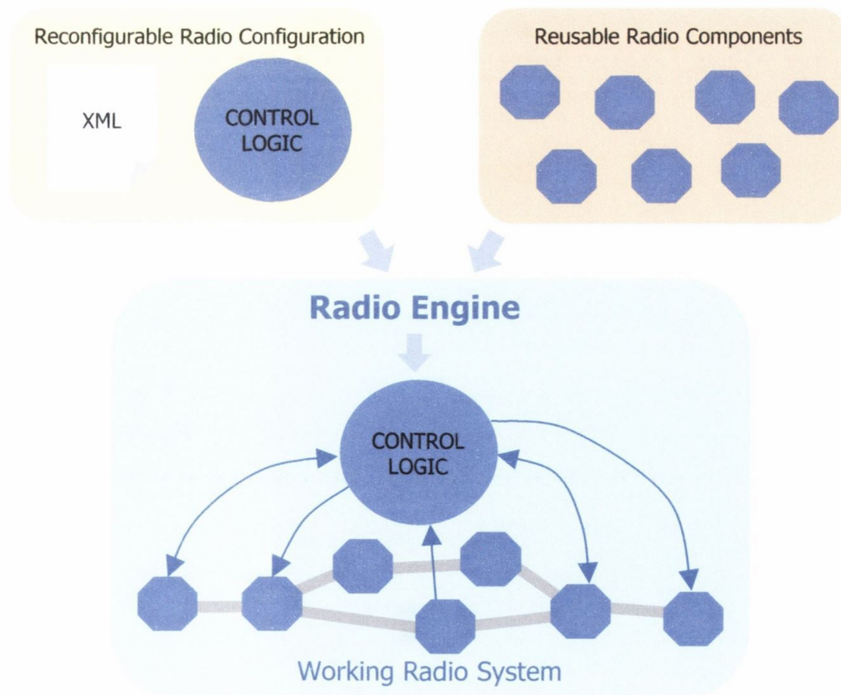


Figure 5.21 – Interaction of Radio Engine, Radio Components and Control Logic

5.5.2 *Basic XML Configurations*

The IRIS architecture defines its own XML configuration that allows a radio configuration to be described. The following example demonstrates how a basic configuration can be used to combine two Radio Components. In this example an FIR filter and a Decimator are being connected. A third component (i.e. a signal generator component) is included in the example as a means of supplying input to the two components of interest.

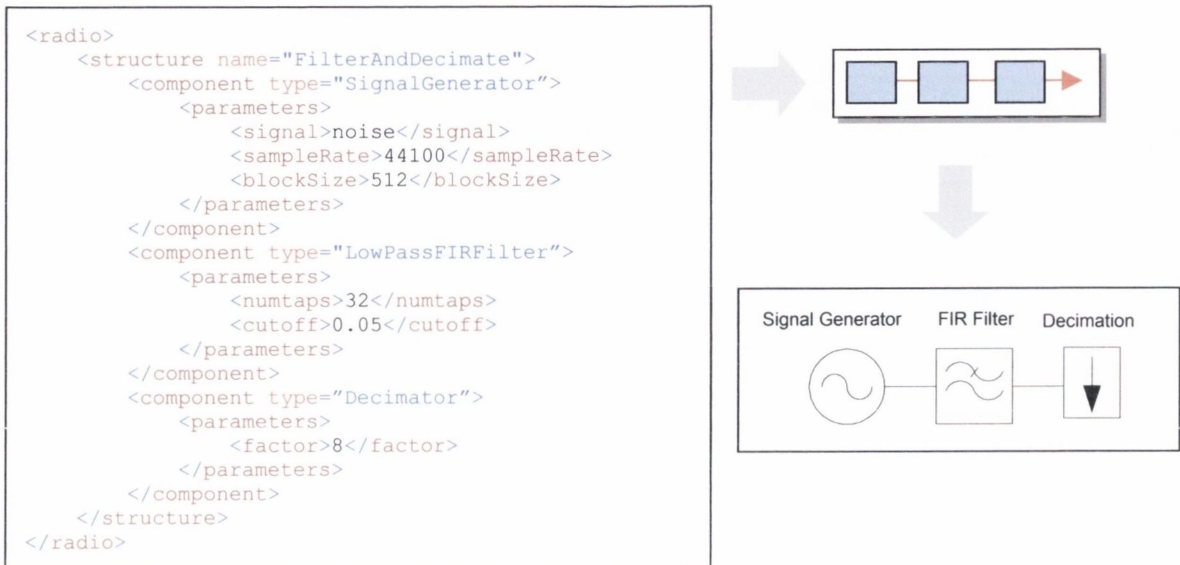


Figure 5.22 – Basic Series of Components

Configurations are built via structures, specified via the `<structure>` XML tag. Structures represent sequences of components and can be combined to form almost any radio configuration. Within a structure components are specified for inclusion via the `<component>` tag. The `<parameters>` tag within this allows the individual parameters for a component instance to be specified. From the simple example shown above, the engine will use the component manager to source and instantiate the '`SignalGenerator`', '`LowPassFIRFilter`' and '`Decimator`' components. Following initialisation and start-up, the Radio Engine will continually call `Process()` in each component passing outputs into inputs. All radio systems are constructed in this way.

There are a few points that should be noted from the example above. Firstly, the order of components as they appear in the `<structure>` XML tag is the same order the signal takes as it passes through the components. (The only exception to this is when parallel components are used. In this case the signal may be transferred to two structures or components, which appear one after another in the configuration.) This approach simplifies the configuration mechanism without the need for specialised structural languages.

Secondly, the `SignalGenerator` component demonstrates the automatic handling of sample rates and block sizes. The exact handling of sample rates and block sizes is discussed in detail in the next Chapter, Section 6.2.5. The `SignalGenerator` is the first component in the signal chain and thus this component determines the sample rate and block sizes used in the chain. In the initialisation phase the Radio Engine reads this signal format from the `SignalGenerator` and calls `CalculateOutputSignalFormat()` on each subsequent component thereby working out automatically the sample rates and block sizes to use between components.

Thirdly, the designer of the radio system is abstracted from the details of the platform, operating system and implementation languages of components. The engine maintains this abstraction and also automates other facilities such as memory allocation.

5.5.3 More Complex Radio Configurations

Not all radio configurations are linear. IRIS was therefore designed to facilitate more complex radio structures. Virtually any desired hierarchy of components can be created by combining the `<parallel>` tag in XML with any of the multiple structures available. The following discussion describes the IRIS structures and details how they are realised in IRIS.

Duplicated Signal Path

It is often necessary to pass a signal to two or more processing algorithms (see Figure 5.23). For example a design may require the filtering of two signals with a comparison of their result. IRIS supports the automatic duplication of signals. Figure 5.23 shows how the output of a sine wave generator can be passed to two FIR filters in parallel. The two filters are defined within one embedded structure, within the overall structure. IRIS recognises the embedded structure and will construct a signal path and include automatic duplication of the signal to both filter components.

In this case the two components in parallel must be configured to accept a signal of the same sample rate as the same signal from the signal generator is copied and passed to both components. During initialisation the Radio Engine will detect such inconsistencies, indicate an error and exit.

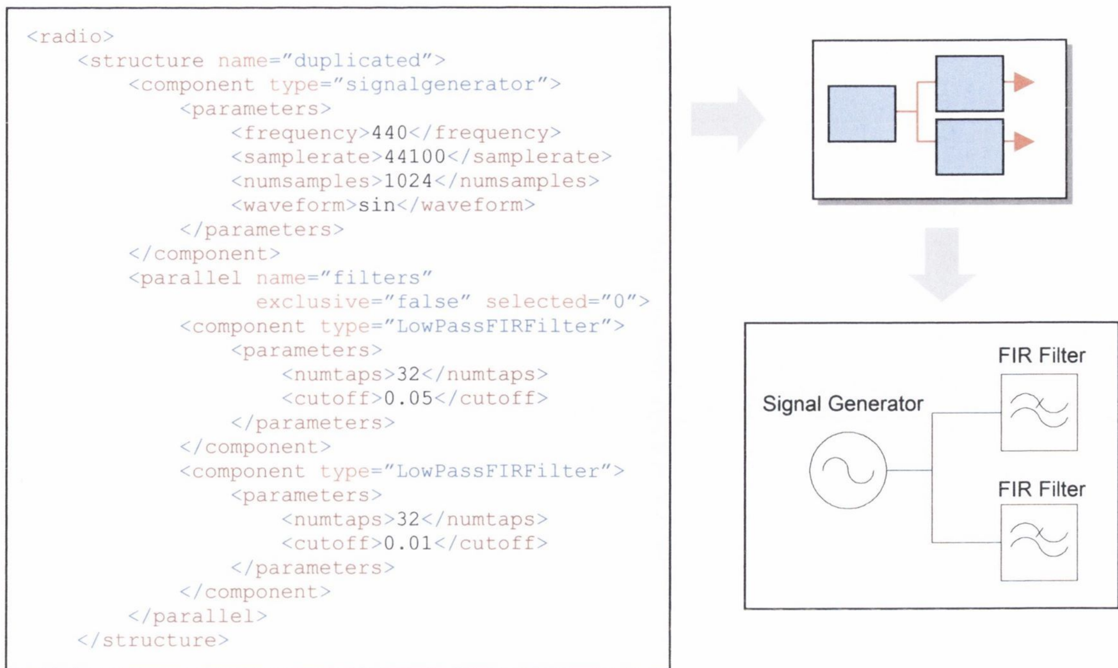


Figure 5.23 – A Duplicated Signal Path

For large amounts of data the duplication of signals can sometimes hinder performance, as a copy of the signal has to be created. For this reason IRIS only performs memory copying when absolutely necessary. This is achieved by analysing the component layout. Signal copying is avoided if IRIS detects that a particular component configuration will not corrupt a signal, thus allowing the same signal to act as an input to multiple components.

Synchronous and Asynchronous Signal Paths

DSP designs often require multiple signals to be processed simultaneously. This accommodates designs that use multiple input or output channels, or applications that require asynchronous processing as is the case in a transceiver which requires both a transmitter and receiver path. IRIS is capable of processing multiple signal paths as it supports the expression of multiple structures in XML. IRIS supports two types of signal paths, synchronous and asynchronous.

A synchronous signal path synchronises signals at particular points in the radio system. This facility is required in designs that process one signal through multiple paths, each path containing different numbers of components. The general problem is illustrated in Figure 5.24. One signal enters the system at X. This signal is copied, one copy applied to A, the other applied to D. The result that appears at Y must be the result of processing through A, B and C. Likewise the output at Z must be the result of the same signal processed by D and E. The order and technique used to process signals is essential in ensuring that the output receives the processed blocks of data at the

same time even though they have taken different paths through the system. Without synchronisation there would be a delay between the outputs of these multiple paths.

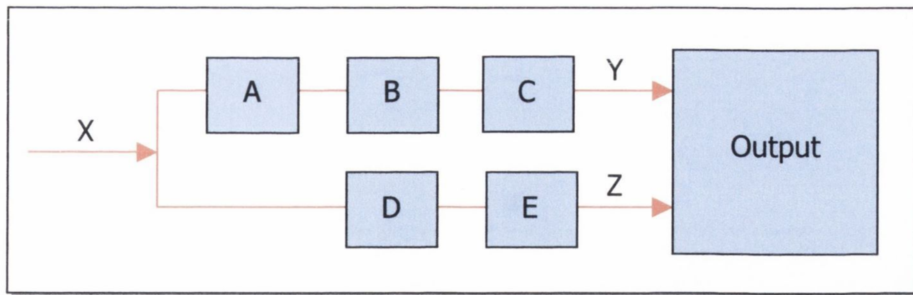


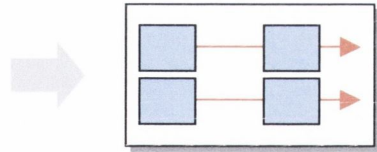
Figure 5.24 – Synchronisation in IRIS

Synchronisation is achieved in the IRIS system by grouping sets of components into a structure. Each structure is treated as a single component thus in the example in Figure 5.24, A, B and C would be grouped for processing as would D and E. A single thread is used for processing which eliminates the need for radio-wide synchronisation of memory. In the example discussed, A, B and C would be processed first with the result stored in memory, followed by copying the signal for processing by D and E. Only then are the two resulting blocks passed to the output.

In contrast, asynchronous structures are executed in different threads in the operating system. In the example above asynchronous operation would mean that (A, B, C) and (D, E) would be processed by different threads. To simplify memory synchronisation the blocks of memory used for communication between these components is allocated from different pools of memory, thus no OS-level synchronisation such as mutexes are required. This type of radio layout is more suitable when completely separate structures of components are required, an example being a transceiver. A transceiver requires both transmit and a receive paths thus it makes sense to separate these out into asynchronously. An advantage of this technique is that the use of multiple threads can also improve performance in implementations that use a lot of hardware I/O. A disadvantage of this technique is that memory cannot be shared between multiple paths therefore more memory may be required.

Figure 5.25 is an example of where synchronous parallel paths are created. The `<parallel>` tag in XML is used to indicate to IRIS that two synchronous parallel structures are to be created. Figure 5.26 is an example of where two independent asynchronous paths are created. The overall radio configuration contains two base structures, both operating independently of each other.

```
<radio>
    <structure name="FilterAndDecimate">
        <parallel name="first" exclusive="false">
            <component type="SignalGenerator">
                <parameters>
                    <signal>noise</signal>
                    <sampleRate>44100</sampleRate>
                    <blockSize>512</blockSize>
                </parameters>
            </component>
            <component type="BandPassFIRFilter">
                <parameters>
                    <numtaps>32</numtaps>
                    <cutoff>0.05</cutoff>
                </parameters>
            </component>
        </parallel>
        <parallel name="second" exclusive="false">
            <component type="SignalGenerator">
                <parameters>
                    <signal>noise</signal>
                    <sampleRate>44100</sampleRate>
                    <blockSize>512</blockSize>
                </parameters>
            </component>
            <component type="BandPassFIRFilter">
                <parameters>
                    <numtaps>32</numtaps>
                    <cutoff>0.05</cutoff>
                </parameters>
            </component>
        </parallel>
    </structure>
</radio>
```



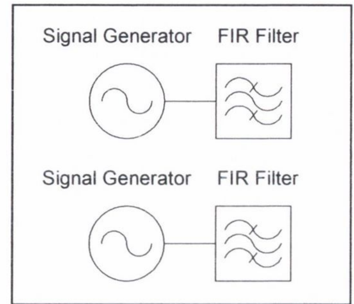
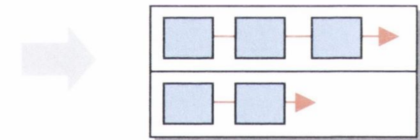
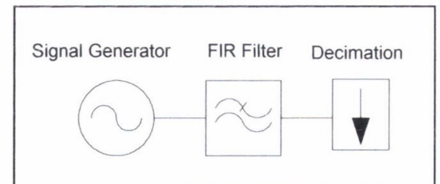


Figure 5.25 – Multiple Synchronous Signal Paths

```
<radio>
    <structure name="FilterAndDecimate1">
        <component type="SignalGenerator">
            <parameters>
                <signal>noise</signal>
                <sampleRate>44100</sampleRate>
                <blockSize>512</blockSize>
            </parameters>
        </component>
        <component type="LowPassFIRFilter">
            <parameters>
                <numtaps>32</numtaps>
                <cutoff>0.05</cutoff>
            </parameters>
        </component>
        <component type="Decimator">
            <parameters>
                <factor>8</factor>
            </parameters>
        </component>
    </structure>
    <structure name="FilterAndDecimate2">
        <component type="SignalGenerator">
            <parameters>
                <signal>noise</signal>
                <sampleRate>44100</sampleRate>
                <blockSize>512</blockSize>
            </parameters>
        </component>
        <component type="BandPassFIRFilter">
            <parameters>
                <numtaps>32</numtaps>
                <cutoffLow>0.10</cutoffLow>
                <cutoffHigh>0.20</cutoffHigh>
            </parameters>
        </component>
    </structure>
</radio>
```





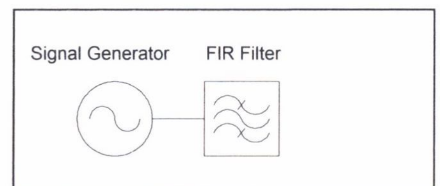


Figure 5.26 – Multiple Asynchronous Structures

Embedded Structure

At any point in the signal path IRIS supports embedded structures. This means that a group of components can be encapsulated to appear as just one component. In the example in Figure 5.27 one of the parallel paths contains an embedded structure.

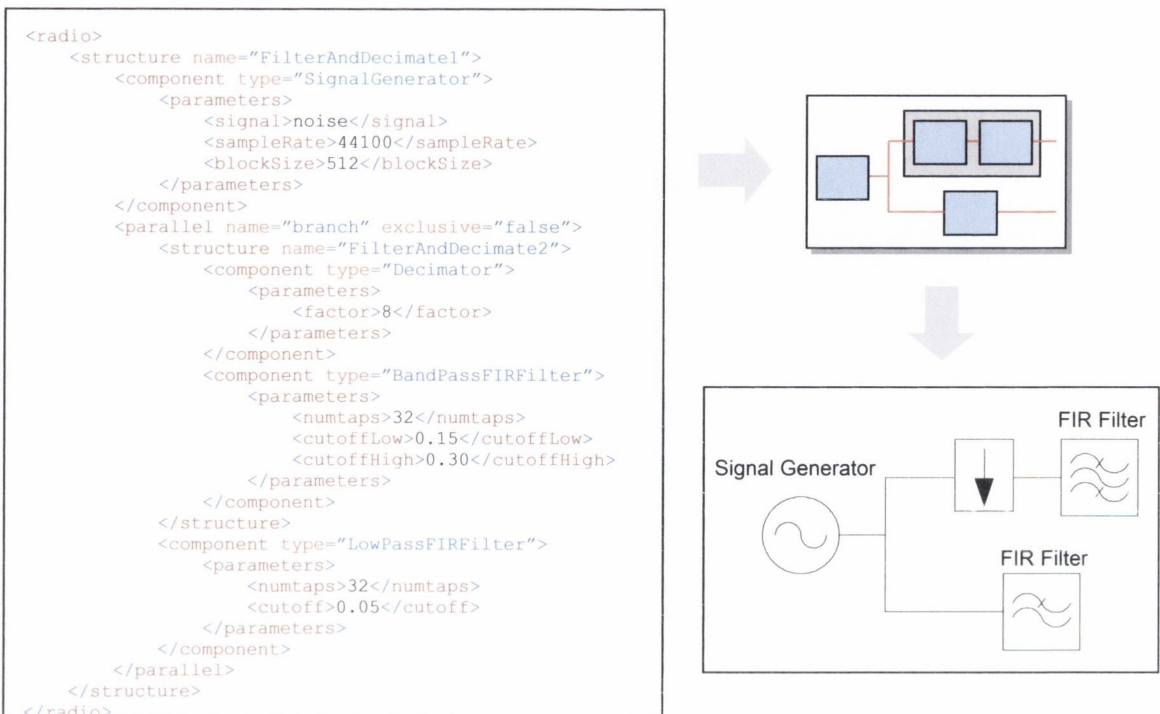


Figure 5.27 – An Embedded Structure

Signal Routing

IRIS supports signal routing in that it allows a signal to be optionally routed to a particular component. For example in Figure 5.28 the signal can be routed to one component or the other. The route that a signal takes can be adjusted at runtime thus allowing dynamic designs to be implemented. For example a modulation detection/classification component could cause the signal to be routed to the appropriate demodulation component. The XML description file allows for this by using the '`selected`' and '`exclusive`' options of the `<structure>` tag. The '`exclusive=true`' statement means that all routes are mutually exclusive thus only one route can be chosen. The '`selected="0"`' statement tells the engine which route to take and this value can be changed at runtime.

Multiple Inputs

Finally, IRIS allows signals from multiple components to be passed to one component. This is achieved via channels. Two signals from different components are combined as a multi-channel signal for input to another component (Figure 5.29). Inputs can also come from embedded structures. By combining all these constructs almost any DSP system can be achieved by the engine.

Figure 5.28 – Signal Routing

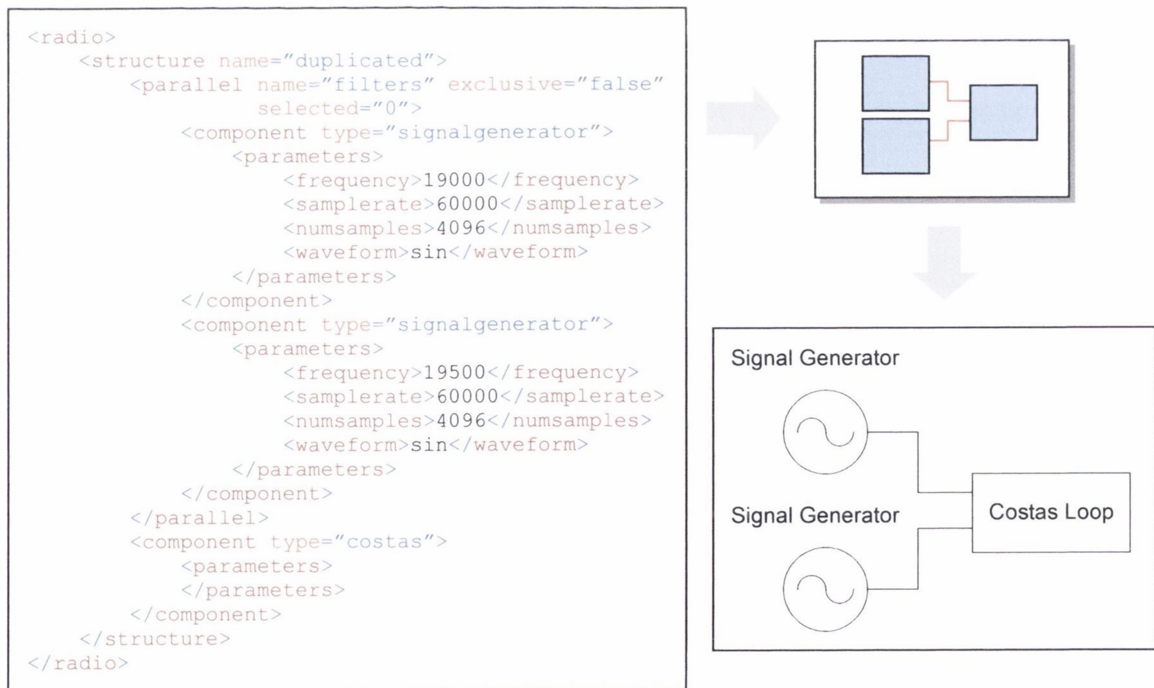


Figure 5.29 – A Component with 2 Input Channels

5.5.4 Internal Radio Representation

The XML parser must verify and convert the XML configuration into an internal representation for realisation by the Radio Engine. In early basic prototypes radio systems were built from series of components, in this case the only information that had to be stored internally was a list of components and their sequential order. However, as the design progressed it was recognised that a more hierarchical design was required to facilitate all of the constructs discussed in the previous section. A model was designed that allows components to be specified in a hierarchical order whilst simultaneously catering for the needs of signal processing.

The main problem that has to be addressed by the internal structure is synchronisation. The embedded structures discussed in the previous section introduce a problem in that they allow multiple components to be viewed as a single component. When single components are used in parallel with embedded components the engine must ensure that all processing has completed before the results are used. Figure 5.30 demonstrates this by showing synchronisation points. All processing of components must be completed before passing beyond this point.

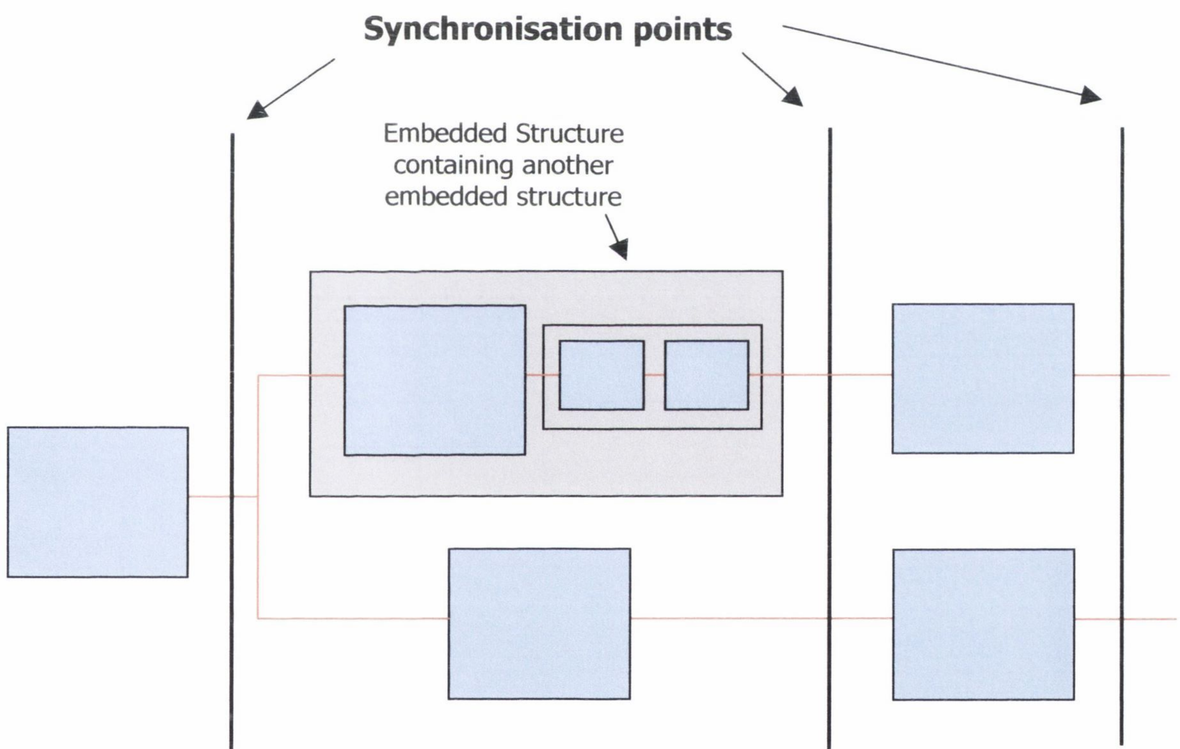


Figure 5.30 – Synchronisation of Processing

Internally the radio system is represented as shown in Figure 5.31. To take account of synchronisation in the internal representation of the radio, components and structures are stored in entities called Units. Each Unit can contain either a component or a structure, a structure being the equivalent of the embedded structure discussed above. Multiple units are stored in a Parallel, the

parallel being synchronised. Thus, during processing the Radio Engine goes through the hierarchy of the radio structure, processing each parallel in turn.

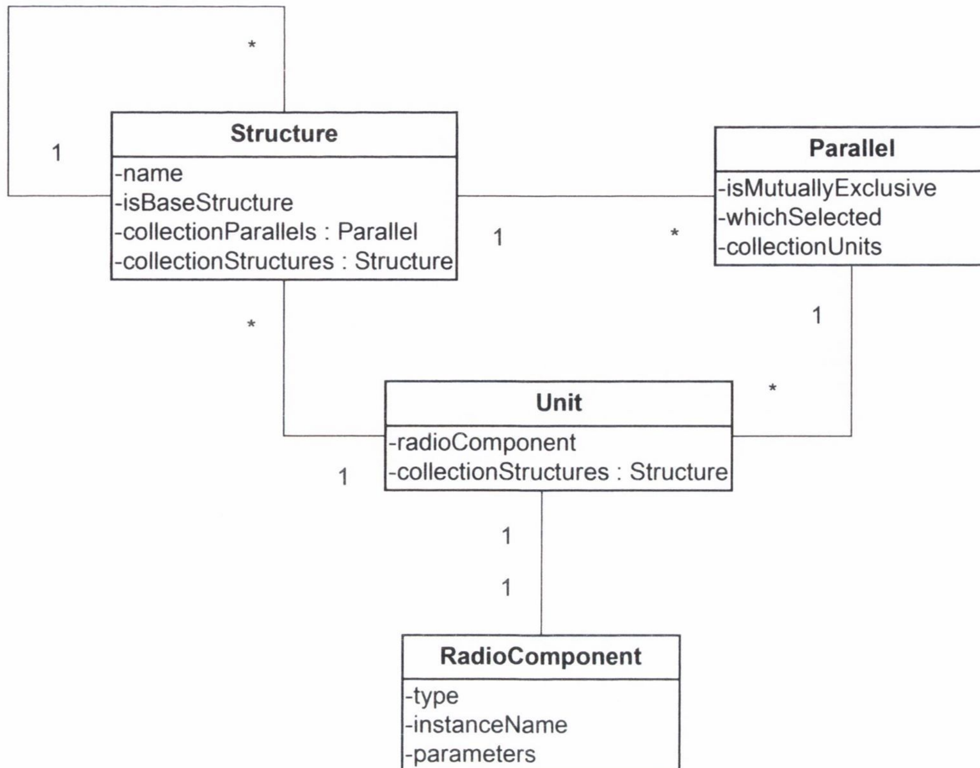


Figure 5.31 – Internal Representation of Radio System

5.5.5 IRIS API

As shown in the IRIS Radio Architecture diagram (see Figure 5.4), the architecture provides an API (Application Programming Interface) called the IRIS API. This API encapsulates all the functionality of the Component Framework into one API that can be used by other applications to create reconfigurable radio systems. The interface provided by the API is shown in Figure 5.32. In the next chapter, Section 6.4 will demonstrate how this interface can be used in practice.

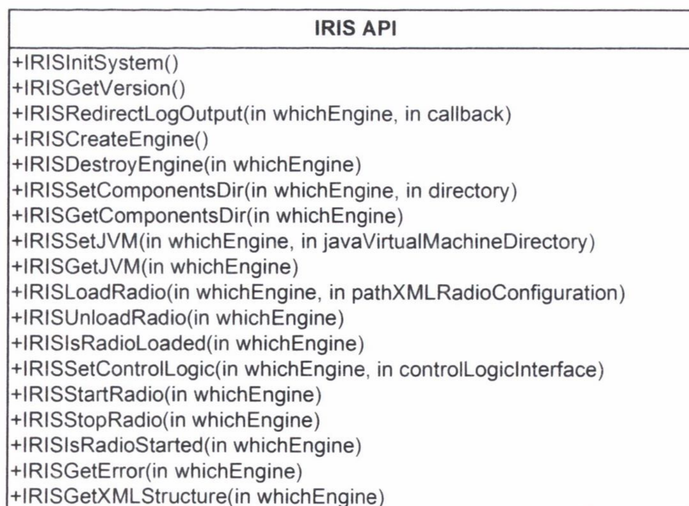


Figure 5.32 – Interface of the IRIS API

5.6 Control Logic

Although components allow functionality to be abstracted and encapsulated into reusable units, when components are combined together dependencies naturally occur (coupling and dependencies were discussed in Section 3.2.2). If for example (see Figure 5.33) a particular radio configuration requires that component *A* is dependent on symbol timing information provided by component *B*, then it follows that component *A* cannot operate without component *B*. If at a later date we want to reconfigure to a new radio implementation which uses a different source of symbol timing, then *A* will have to be changed to accommodate this new dependency.

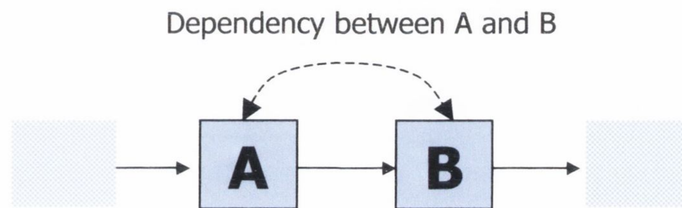


Figure 5.33 – Component Dependency

IRIS allows components to remain independent and decoupled by introducing the concept of Control Logic which allows component interaction to be specified by an implementation that exists outside the components themselves (see Figure 5.34). In addition, Control Logic is also abstracted from the overall structure of the radio implementation. This means that even if additional components are added into a structure (such as inserting a new component between *A* and *B*) the control logic will still function.

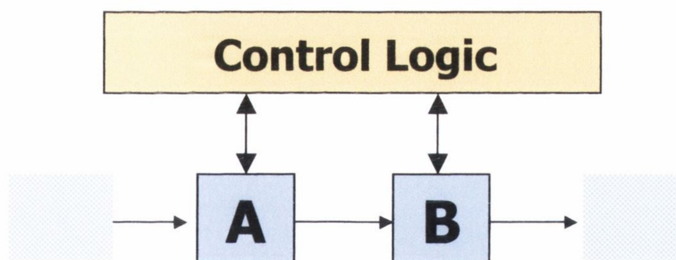


Figure 5.34 – Using Control Logic to Eliminate Component Dependencies

The Control Logic for IRIS can be implemented in either C++ or Java and is isolated from the particulars of components. Java in particular allows radio implementations to take advantage of Java's vast class library. A simple API is provided in both C++ and Java to allow the Control Logic to query and manipulate the radio through interaction with the Radio Engine. The interface definition of this API is shown in Figure 5.35. By making calls to this API the Control Logic can manipulate the components of a radio system.

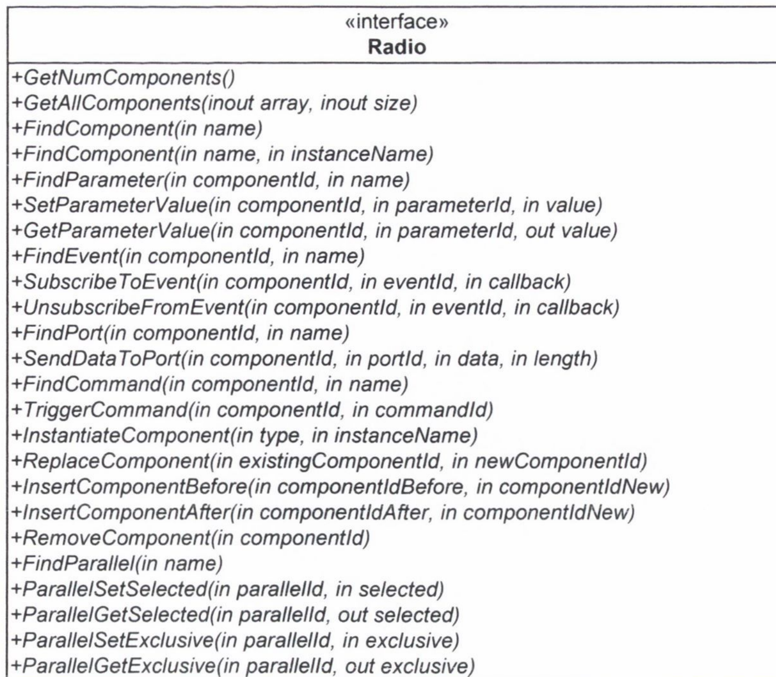


Figure 5.35 – Interface Control Logic uses to Control Radio

Like Radio Components, the Control Logic has a well-defined lifecycle and each controller must implement the same interface for use by the Component Framework. Figure 5.36 shows the interface definition for Control Logic. Three methods must be implemented:

Load(): This method is called by the Radio Engine to initialise the Control Logic. It provides a reference to the Radio interface (Figure 5.35), allowing the Control Logic to interact and reconfigure the radio system.

AttachToComponents(): This method is called to allow Control Logic to perform any initialisation of its own prior to the running of the radio.

Unload(): This method is called by the Radio Engine to unload the Control Logic allowing it to free its resources.

The lifecycle of the Control Logic is depicted in UML sequence diagram shown in Figure 5.37.

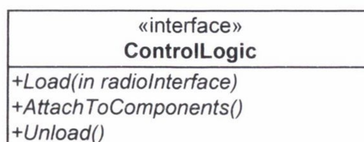


Figure 5.36 – Controller Interface

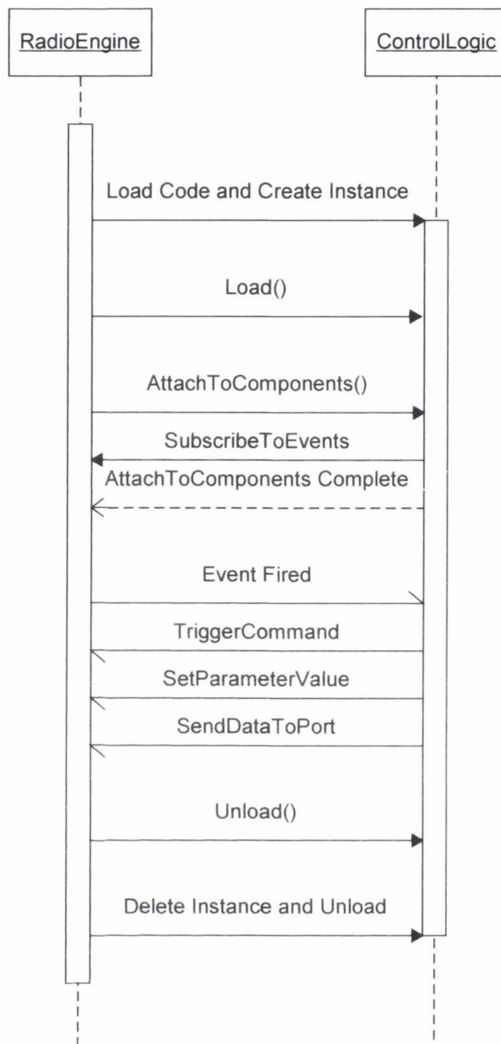


Figure 5.37 – Lifecycle of Control Logic

5.7 Worked Example

The overall operation of the IRIS framework is best illustrated by an example. While the IRIS system could be used for building many devices such as receivers, transmitters, test equipment and signal analysis tools, the transceiver is the best example to illustrate its many features. A transceiver incorporates both a receiver and transmitter and so is a good choice for demonstrating how multiple sub-systems can co-exist and interoperate within the Component Framework. For this example a digital FSK (Frequency Shift Keying) transceiver is considered.

5.7.1 An FSK Transceiver

The FSK transceiver considered will allow digital data to be transmitted and received using the IRIS system. This functionality could be used in many scenarios for example in communicating digital speech or any form of data. This raises the question as to what the operating parameters of the device might be, e.g. frequency, data rate, bandwidth, etc. Usually in traditional radio design these specifications determine the structure, hardware and capabilities of the final device. For example the data rates required to facilitate transmission of speech and video are very different and so will usually result in completely different hardware being used. Changing this rate if at all possible usually requires a significant change in hardware forcing changes from clock frequencies to firmware.

In the IRIS system however, all the specification of the application is independent of low-level hardware details. At this higher abstraction, system specifications are more decoupled from hardware, for example the hardware architecture is never changed in response to a change in transmission data rate. The one parameter that the system is dependent on is the overall processing power of the architecture and once the GPP system provides enough processing capability many radio types using different specifications can be implemented. This type of capability radically changes the radio design paradigm affecting how radio systems can be viewed and constructed. Instead of creating an FSK modulator for voice and another for data, one generic FSK software component is created and by changing the parameters of this component many different types of information can be accommodated.

With this approach in mind, designing the FSK transceiver becomes less about choosing operating parameters and more concerned with functional partitioning of the system via software components. Once the appropriate generic components have been built, the radio system can be configured via parameters to deliver the required specification.

5.7.2 Partitioning the System

In approaching the problem of building the FSK transceiver, the functional partitioning via components is thus first considered. A basic design for an FSK transceiver is shown in Figure 5.38. In the transmitter path a binary signal is modulated as an FSK signal using two local oscillators. Each oscillator generates a signal corresponding to a binary '1' or '0' producing a waveform as shown in Figure 5.39. This signal is then up converted, filtered and then converted to an analogue signal. Likewise in the receiver branch the received signal is digitised, down converted and filtered. The baseband signal is then demodulated by mixing it with two local oscillators at the same frequencies as those in the transmitter. Following filtering, the transmitted data is recovered via an 'integrate and dump' stage which outputs the stream of received bits. (For simplicity a non-coherent receiver is shown).

The actual air interface of the radio system is facilitated by an RF front end. This device is responsible for down-converting a signal of interest for demodulation and for up-converting the transmitting signal for transmission. As discussed in Section 2.4.3 up and down conversion can work in various ways either through an IF or by using direct conversion (zero-IF). This example assumes that an RF front-end exists that allows transmission and reception on a large range of frequencies with a suitable signal to noise ratio.

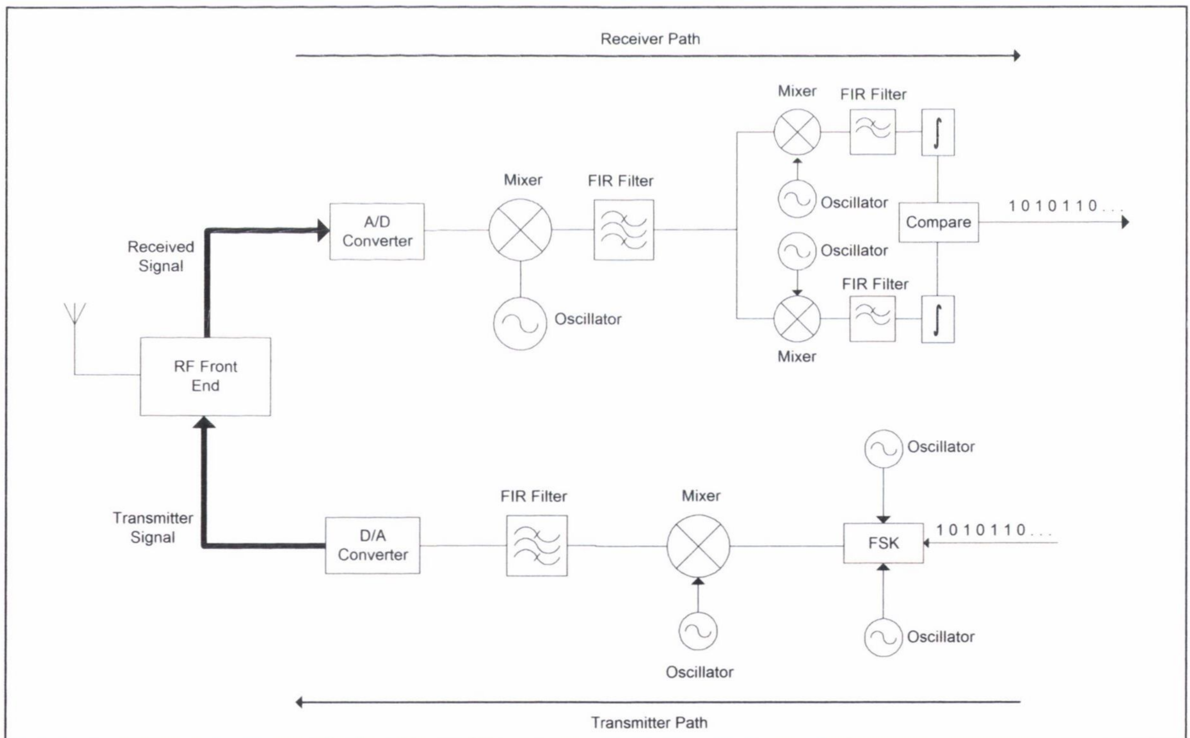


Figure 5.38 – FSK Transceiver Design

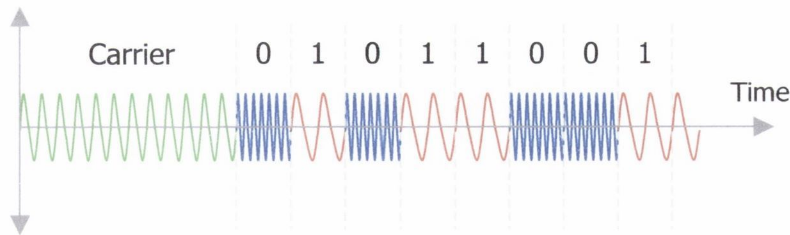


Figure 5.39 – FSK Waveform

There are various goals that must be addressed in partitioning such a system into components. Firstly, there is the goal of reuse. The aim here is to create components that can be reused in other applications. This requires an effective component granularity to be chosen. Secondly, reconfigurability is an issue. The boundary of components should facilitate structural reconfiguration so that components can be replaced or inserted at runtime. For this to happen the partitioning of the system should be related to function, for example it makes more sense to replace a reusable channel extraction component then it does to alter individual multipliers in a filter. Thirdly, the structure of components used should facilitate the operation of the radio. Components should be suitably identified as DSP, IO or Standalone components. Also, their composition should occur logically, for example separating out the signal paths for transmitter and receiver.

Figure 5.40 shows how such a design can be partitioned into software components using the IRIS system. There are many ways this partitioning can take place and various tradeoffs associated with its implementation. For example, consider the channel extraction component. This component takes a wideband signal, mixes it with a local oscillator, filters and then decimates the signal to a lower sample rate. While this component could be built from separate mixer, filter and down sampler components, it is sometimes better to encapsulate all this functionality into one component. Doing so can offer opportunities to optimise the performance of the component and is suited to radio functions such as channel extraction that are particularly data intensive. For example, Welborn [Welborn99b] describes a technique for implementing narrow band channel extraction from wideband receivers, a technique that reduces the processing requirement for channel extraction by combining and reordering stages of channel extraction.

On the other hand there are cases where optimisation may not be possible or required. The transmitter of the FSK transceiver is an example. Here separate components are used to implement the up conversion, up sampling and filtering of the signal before transmission. At this point the data rates may not be so intensive and reducing the ability to optimise may be an acceptable loss in the face of gaining more reusable components.

Another example is the FSK demodulator component. This contains a great deal of functionality that could be reused in other applications such as integrators, etc. However, the overall design must

be considered and implementing this and larger structures via hierarchies of components can over complicate a design making the radio design error prone and difficult to maintain. Also, if we want to later replace the FSK demodulator with another demodulator it is much easier to replace one component rather than a complex structure of interconnected components. For this reason the best approach is to partition the system via function, e.g. FSK modulator, down sampler, channel extractor.

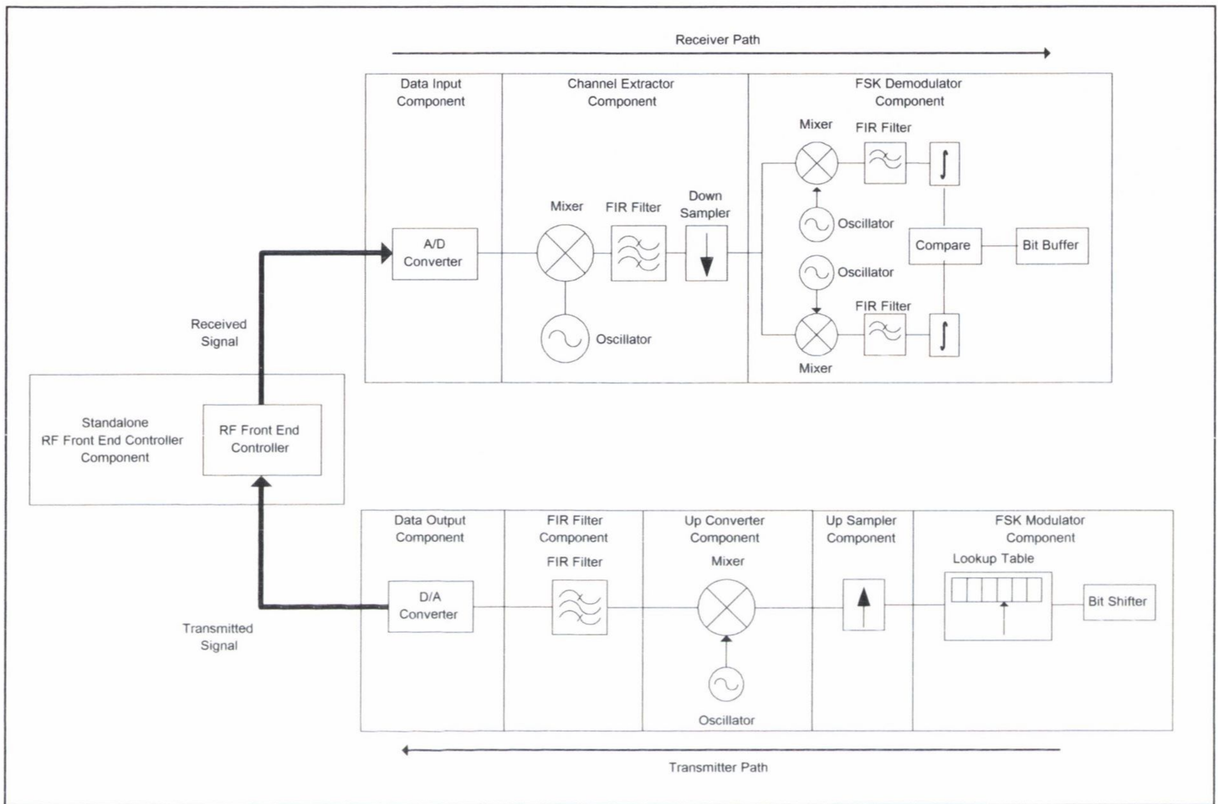


Figure 5.40 – Partitioning of FSK Transceiver into Software Components

5.7.3 Structures

With an appropriate component composition decided, an XML configuration can be written to describe the design. This XML configuration is used by the Component Framework to create the radio system. As discussed in Section 5.5.2 an XML file is used to define a set of structures, a structure being a set of components. Internally the Radio Engine uses these structures to identify how it should pass signals between components. Considering the FSK transceiver example, it contains three sub-systems, the receiver, the transmitter and the RF front-end controller. To separate out these sub-systems each is specified in a separate structure. Internally, the Radio Engine assigns one thread to each structure and this thread controls all the interaction among components. The use of a thread for structures and hence multi-threading in general is an important consideration for the Radio Engine. In general multi-threading offers advantages in that it can improve performance and is a useful technique in partitioning code into separate functional units.

However it can also introduce problems such as increased code complexity. Multi-threaded code often requires synchronisation by the use of mutexes and semaphores, which can be difficult to debug and maintain.

In terms of the IRIS system there were various ways multithreading could have been used in the processing of signals. One option was to assign a thread to each component; in this way components would be autonomous in their processing of signals (see Figure 5.41). Using this approach however, introduces some problems. Firstly synchronisation is required to pass signals between components and secondly, this synchronisation must occur independently of the radio structure. It is difficult to have a system that uses threads for each component and at the same time allow components to be developed without knowledge of their place in the radio system. One aim of the system was to reduce the burden on the programmer and to make the system easy to use. Therefore this approach was not used as overall it makes the radio system more complex.

Instead the approach used has been to use a single thread for each structure which has control over the whole radio system. This greatly simplifies the complexity of Radio Components. In terms of the FSK transceiver example, this approach results in the engine using three threads; one for each structure, the receiver, transmitter and RF front-end controller.

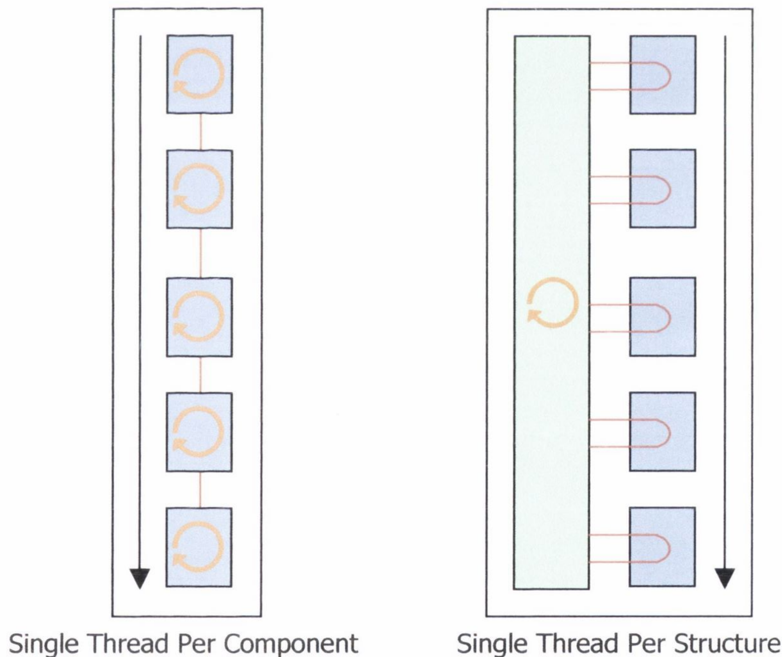


Figure 5.41 – Multithreading Approaches

The XML configuration to achieve this design is shown in Figure 5.42. This configuration shows how the various parameters for the transceiver can be set.

```
<?xml version="1.0" encoding="utf-8" ?>
<radio>
	<description>
		<name>FSK Transceiver</name> <comment>FSK Transceiver Example</comment>
	</description>
	<structure name="RFFrontEnd">
		<component type="HardwareController">
			<parameters>
				<serialport>COM1</serialport>
				<transmitfrequency>100000000</transmitfrequency>
				<receivefrequency>100000000</receivefrequency>
			</parameters>
		</component>
	</structure>
	<structure name="Receiver">
		<component type="a2dpci4020" instance="DataInput">
			<parameters>
				<samplingRate>4000000</samplingRate>
				<outputBlockSize>524288</outputBlockSize>
				<channel>1</channel>
				<useExternalClock>off</useExternalClock>
				<voltage>5</voltage>
			</parameters>
		</component>
		<component type="ChannelExtractor">
			<parameters>
				<MixerFrequency>2694700</MixerFrequency>
				<NumberTaps>8</NumberTaps>
				<FilterCutoff>0.07</FilterCutoff>
				<Decimation>16</Decimation>
			</parameters>
		</component>
		<component type="FSKDemodulator"">
			<parameters>
				<BlockSize>40960</BlockSize>
				<SampleRate>250000</SampleRate>
				<SignalFrequency1>10000</SignalFrequency1>
				<SignalFrequency2>30000</SignalFrequency2>
				<CarrierFrequency>20000</CarrierFrequency>
				<SymbolLength>30</SymbolLength>
			</parameters>
		</component>
	</structure>
	<structure name="Transmitter">
		<component type="FSKModulator"">
			<parameters>
				<BlockSize>40960</BlockSize>
				<SampleRate>250000</SampleRate>
				<CarrierFrequency>20000</CarrierFrequency>
				<SignalFrequency1>10000</SignalFrequency1>
				<SignalFrequency2>30000</SignalFrequency2>
				<SymbolLength>30</SymbolLength>
			</parameters>
		</component>
		<component type="UpSampler">
			<parameters>
				<ratio>4</ratio>
			</parameters>
		</component>
		<component type="UpConverter">
			<parameters>
				<MixerFrequency>190000</MixerFrequency>
			</parameters>
		</component>
		<component type="FIRFilter">
			<parameters>
				<NumberTaps>32</NumberTaps>
				<CutoffFrequency>0.05</CutoffFrequency>
			</parameters>
		</component>
		<component type="DataOutput">
			<parameters>
				<OutputChannel>0</OutputChannel>
				<ScaleFactor>2.0</ScaleFactor>
			</parameters>
		</component>
	</structure>
</radio>
```

Figure 5.42 – XML Configuration for FSK Transceiver

5.7.4 Control Logic

The last aspect of the system that needs to be addressed in how data can be input and output from a device like an FSK transceiver. As introduced in Section 5.6, control logic is a general-purpose mechanism for providing this type of functionality. Control Logic provides a generic way to perform interaction with components without requiring specific knowledge of the internal workings of a component or of the structure of the radio itself.

Generic interaction with components is provided by the control facilities all components provide, namely, properties, events, ports and commands. Properties allow control logic to change a value in a component, for example a frequency setting. A piece of Control Logic can use a component's events by providing callback functions that are triggered when a component fires an event. Ports allow Control Logic to send data to a component and commands allow the logic to trigger the running of routines in a component. By using combinations of these, control logic can control any aspect of the radio system.

Control Logic mimics the type of control functionality that might be found in the microprocessor of a software-defined radio system or in the infrastructure of the JTRS SCA discussed in Section 2.5.2. Its similarities being that it controls and monitors the overall operation of the device. However, apart from this, Control Logic in the IRIS system is distinctly different. Unlike these other systems the IRIS control logic is only loosely coupled to the components of the radio system. It has been specifically designed this way for reconfigurability. Components can be added, removed and replaced from the radio system and the Control Logic will still function. This loose coupling is maintained by the Radio Engine which abstracts the control logic from the engine and by the APIs it provides. Instead of providing direct access to components the Radio Engine acts as a proxy to all control logic/component interaction, effectively maintaining separation between control logic and components. Instead of manipulating the parameters of a component directly it uses the standard facilities for component interaction provided by the framework (as demonstrated in Section 5.6). By using calls such as `SetParameter()` and `ReplaceComponent()` the radio system can be manipulated to achieve parametric and structural reconfiguration.

Returning to the FSK transceiver example, the requirement here is straightforward; the control logic must be able to send data to the FSK modulator component for transmission and be able to receive data from the FSK demodulator component following reception. This is illustrated in Figure 5.43. For transmission the control logic accesses the '`SendData`' port of the FSK modulator component it can then use the controller API to send data to the component. Internally the component implements a handler function that allows it to react to the instruction to send data. For reception the control logic subscribes to the '`DataReceived`' event that the FSK demodulator component provides. Every time data is received the Radio Engine calls a method in the Control

Logic thereby transferring the data from the component through the proxy of the engine and into the controller.

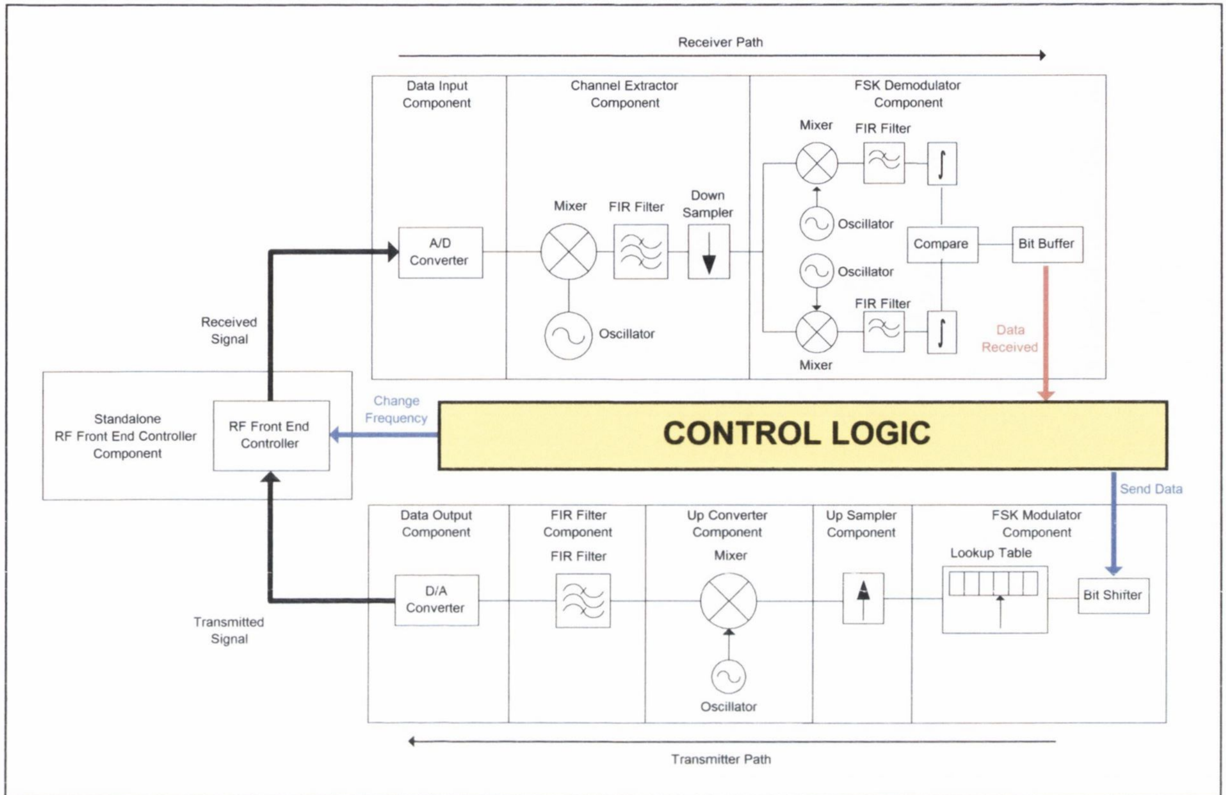


Figure 5.43 – FSK Transceiver Using Control Logic

Figure 5.44 shows example control logic for the FSK transceiver (error reporting code has been removed for brevity). This controller continuously listens for a signal on a specified frequency. Once a signal is received it transmits 'Hello' and moves to the next frequency. During the calls to `Load()` and `AttachToComponents()` the control logic finds all the references it requires to the components of the radio system. It finds the '`SendData`' port for transmission and subscribes to the '`DataReceived`' event from the demodulator component. When a signal is received the frequency is incremented and the RF-Front End is instructed to move to a new frequency.

```
void Controller::Load(EngineInterface *engineInstance)
{
	engine = engineInstance;
}

bool Controller::AttachToComponents()
{
	//Find the FSK modulator component
	hComponentModulator = engine->FindComponent("FSKModulator");

	//Find the ModulateData port of the FSK modulator
	hPortModulate = engine->FindPort(hComponentModulator, "ModulateData");

	//Find the FSK Demodulator component
	hComponentDemodulator = engine->FindComponent("FSKDemodulator");

	//Subscribe to the SignalReceived event of the demodulator
	engine->SubscribeToComponentEvent(hComponentDemodulator,
	                                  "SignalReceived",
	                                  (int)this,
	                                  SignalReceived);

	//Find the FrontEnd component
	hComponentFrontEnd = engine->FindComponent("FrontEnd");

	//Find the parameter for setting the frequency
	hParameterFrequency = engine->FindParameter(hComponentFrontEnd, "Frequency");

	//Set the frequency to an initial value
	currentFrequency = 100000000;
	engine->SetParameterValue(hParameterFrequency, currentFrequency);

	return true;
}

void Controller::Unload()
{

}

void Controller::SignalReceived(int identifier, void* data, unsigned int length)
{
	Controller *instance = (Controller*)identifier;

	//Modulate/Transmit the message 'Hello'
	char *message = "Hello";
	instance->engine->SendToPort(instance->hComponentModulator,
	                             instance->hPortModulate,
	                             (unsigned char*)message,
	                             strlen(message)+1);

	//Change the operating frequency of the RF Front End
	currentFrequency += 1000000;
	engine->SetParameterValue(hParameterFrequency, currentFrequency);
}
```

Figure 5.44 – Sample Control Logic Source Code

5.7.5 Reconfiguration

Section 4.2 discussed reconfigurability and how this can be broken down into parametric, structural and application reconfiguration. This section demonstrates how the IRIS system achieves reconfigurability in the context of the FSK transceiver example.

The example of the FSK transceiver already features parametric reconfiguration. When changing the frequency of the RF Front-End the control logic is changing a parameter of the radio system

and thus reconfiguring its functionality. Any parameter of a component in the radio can be changed in this way thus allowing any aspect of the system to be reconfigured dynamically at runtime.

Internally inside components, the component must react to a parameter change. Each component can optionally implement the `ValueChanged()` method and thus perform any recalculation or reconfiguration required to react to a parameter change. For example, the FSK modulator component may have to recalculate its lookup table to respond to a change in operating frequencies.

Structural reconfiguration allows Control Logic to manipulate the structure of the radio system by adding, removing or replacing components at runtime. Also, when components are in parallel signals can be routed to one or other components. To demonstrate how this works in the context of the FSK transceiver, Figure 5.45 shows how the FSK modulator component of the transceiver can be replaced at runtime.

```
void Controller::SignalReceived(int identifier, void* data, unsigned int length)
{
        Controller *instance = (Controller*)identifier;

        //Extract the name of the modulator
        char newModulationScheme = (char*)data;

        //Create a new component
        HANDLE_COMPONENT hComponentNewModulator =
                engine->InstantiateComponent(newModulationScheme);
        if(hComponentNewModulator == INVALID_HANDLE_VALUE)
        {
                printf("Unknown modulation scheme or component %s\n", newModulationScheme);
                return;
        }

        //Replace the existing modulator with the new one
        if(engine->ReplaceComponent(hComponentFSKModulator, hComponentNewModulator) == false)
        {
                printf("Error replacing component, incompatible with this configuration");
                return;
        }

        //Release the existing modulator
        engine->DestroyComponent(hComponentFSKModulator);
}
```

Figure 5.45 – Code for Replacing a Component at Runtime

In this example the data of the received signal contains the name of the new component that should replace the existing modulator. Using this name the new component is instantiated and used to replace the existing component with a call to `ReplaceComponent()`. This example demonstrates that functionality of the radio can be replaced at runtime. This shows that it is possible for radio systems to alter their structure based on information received from other systems. This facility also allows the FSK transceiver to become a general-purpose generic transceiver requiring only changes in modulators and demodulators to enable new modulation schemes. (Chapter 7 will demonstrate case studies showing how this type of reconfiguration can be used in other scenarios).

The final type of reconfiguration considered is application reconfiguration. This would require the IRIS system to be reconfigured at runtime to implement a system using a different set of components, parameters and control logic, thus in the focused context of the FSK transceiver this facility holds less consequence. However, instead of changing the radio application completely, application reconfiguration can be used as a means to upgrade the software of a reconfigurable radio. With new components, parameters and control logic, a working radio system can be reconfigured to a newer version possibly fixing bugs or enabling new capabilities. For example, in the FSK transceiver example this could be used to upgrade the device by introducing a new configuration with a new set of parameters that allow for better data throughput. Application reconfiguration can be programmed via the IRIS API which is discussed in the next chapter.

5.8 Summary

The focus of this chapter has been the basic design and capabilities of IRIS, a means of building a reconfigurable radio system using a component framework. The approach taken makes it extremely simple to express the structure of a reconfigurable radio system. Part of this approach has been to factor out as many common radio functions as possible into the framework so that radio system design is simple and straightforward. This allows the system developer to concentrate on the design of radio systems without having to deal with recurring problems and issues surrounding platforms and hardware. As demonstrated in the worked example, the system features application, structural and parametric reconfigurability through a cohesive API. IRIS is thus a system that allows the development of highly reconfigurable radio systems.

6 Implementation and Analysis

6.1 Introduction

This chapter provides further insight into the reconfigurable radio concept by discussing the practical implementation of a real-life reconfigurable radio system. The IRIS Architecture described in the previous chapter has been implemented as a fully functioning system and runs on Windows 2000/XP. Implementation issues surrounding the Radio Component, the fundamental unit for building reconfigurable radios in IRIS, are discussed in Section 6.2 and are followed by practical examples in Section 6.3. Section 6.4 provides details of the IRIS API and supporting tools. The use of external hardware is discussed in Section 6.5 with results of scalability and memory analysis of the system presented in Section 6.6.

6.2 Implementing Radio Components

This section gives insight into the development of Radio Components on Windows. It starts by discussing some operating system issues of relevance when considering component-based reconfigurability. It then goes on to provide detailed technical information on how Radio Components deal with different sampling rates and data types, how they carry out signal processing, and how the Component Framework combines components together to create a radio system.

6.2.1 Choice of Operating System

There were no specific requirements of the IRIS system that demanded a particular operating system. All modern operating systems provide basic system services such as virtual memory, multi-threading, networking and therefore any of them would have been suitable for building the core IRIS system. However, integrating the IRIS system with hardware posed a significant challenge. During the course of this research it was difficult to source A/D/A converters of appropriate specification and most of them suitable for the task required Windows to operate therefore Windows was a natural choice. Also, using Windows offered the opportunity to integrate the IRIS system into the DAWN networking system, also a Windows-based system, the result of which is presented in Chapter 7.

6.2.2 Radio Components on Windows

The IRIS system was targeted for development in an object-oriented language such as C++, but the C++ language itself does not have direct support for creating software components. In practice, it is the operating system that dictates how code can be encapsulated and reused in this way. To implement IRIS Radio Components for practical use the following questions had to be addressed for the windows operating system:

- What form would a component take?
- How would components be loaded, unloaded and instances created of a component at runtime?
- How would components expose their functionality?

The unit of code reuse inherent to the Windows operating system is the DLL (Dynamic Link Library). DLLs are libraries of executable code that can be dynamically loaded from disk by applications. It should be noted that while it would be possible to develop a completely proprietary method for encapsulating code in the same way as a DLL, the DLL approach has a distinct advantage. DLLs are highly integrated into the Windows operating system as it uses DLLs to improve system performance. For example, if multiple threads on the same Windows computer use the same DLL, only one copy of the DLL code will be loaded and shared seamlessly between the threads. This optimisation provides better overall system performance and is particularly important for a software radio system that re-uses multiple components in a radio design. DLLs thus provide an efficient mechanism for reuse.

DLLs make their functionality available or expressed in software terms, 'expose their functionality' through an export table. This table describes the functions that the DLL contains, functions that other applications can make use of by loading the DLL. Most windows development tools allow the creation of DLLs and functions written in languages such as C and C++ can be exported in this way.

IRIS Radio Components have been realised using DLLs and each component is written as a separate DLL. Each Radio Component DLL exposes two functions that the IRIS Component Framework can use to create and destroy instances of a component, `CreateRadioComponent()` and `ReleaseRadioComponent()`.

Figure 6.1 shows how these functions are exported from the DLL for a Radio Component. The component in the example is the `SignalGenerator` component. As detailed in the previous chapter, this component is used to generate a signal that can be fed to other components. When `CreateRadioComponent()` is called an instance of the specified component is created. Likewise

a call to ReleaseRadioComponent() deletes the instance of the component. It should be noted that while C++ has been used for this work, any language that supports the exporting of code in this way (i.e. virtual function pointer tables) could be used to build components.

```
extern "C" __declspec(dllexport) Component* CreateRadioComponent()
{
	return new SignalGeneratorComponent();
}

extern "C" __declspec(dllexport) void ReleaseRadioComponent(Component* comp)
{
	if(comp != NULL)delete comp;
}
```

Figure 6.1 – Exporting a Component from a DLL

DLLs meet all the requirements of the Radio Component, and are particularity suitable because they contain native code. DLLs can be dynamically loaded and unloaded from the system using calls to the Windows platform API functions LoadLibrary() and FreeLibrary(). Once loaded, any number of instances of a component can be created using calls to CreateRadioComponent(). These component instances can then be used by the Component Framework to realise the radio design. The overhead of doing this is negligible as DLLs are an integral and thus highly optimised aspect of the Windows operating system.

6.2.3 Programming Radio Components

As shown in the previous section, the functions exported by the DLL is straightforward requiring only two functions, however the actual implementation of a Radio Component itself is more involved. As shown in Chapter 5, Section 5.4, IRIS Radio Components are rich in functionality and as a result require the implementation of many interfaces. While the IRIS system was designed to facilitate rapid development and experimentation of radio systems, having to implement numerous interfaces to create a component can be tedious. To solve this problem a scripting language and code generator has been developed to automate the process.

The scripting language is written as part of the C++ header file of a component. The programmer writes attributes alongside C++ code. These attributes expose information about the component and identify the properties of the component. (Attributes are hidden in C++ comments to avoid compiler errors). Properties include parameters, events, ports and commands as discussed in the previous chapter. Before compilation a Java-based parser reads these attributes and generates seventy methods offering all the functionality required by the Component Framework. The reason so many methods are required is to facilitate function overloading so components can support

multiple data types for parameters and events. A full list of these methods are included in Appendix 10.1.

Figure 6.2 shows the C++ header file of a `SignalStrength` component. This simple component has been designed to fire events indicating whether the signal received is above or below a threshold value.

```
//@component analyses the signal strength of the incoming block
//@version 1.1
//@author Philip Mackenzie
//@event SignalAboveThreshold float fired when the signal level is greater than threshold
//@event SignalBelowThreshold float fired when the signal is less or equal than threshold
class SignalStrengthComponent : public DSPComponent
{
private:
        //@param the threshold in dB at which a signal exists
        //@default -144
        //@dynamic
        int threshold;

public:
        virtual void GetDetails(ComponentDetails *details);
        virtual void CalculateOutputSignalFormat();
        virtual bool Init();
        virtual void Process(Signal signal);
        virtual void Destroy();
};
```

Figure 6.2 – Header File of a Signal Strength Component

Each line starting with '`//@`' indicates an element of the scripting language. For example, the line:

```
//@event SignalAboveThreshold float fired when the signal level is greater than threshold
```

indicates that this component exposes an event called '`SignalAboveThreshold`' and every time this event is fired it supplies a floating point value. The remainder of the line allows the programmer to provide information about the event.

The declaration:

```
        //@param the threshold in dB at which a signal exists
        //@default -144
        //@dynamic
        int threshold;
```

exposes one of the member variables of the class as a parameter of the component. The '`//@param`' statement must be included and indicates that `threshold` will be exposed as a parameter. '`//@default`' provides a default value for the threshold. The engine automatically uses this value if none is supplied. '`//@dynamic`' indicates that this component can be changed dynamically at runtime and the engine will inform the component when this value has changed

through the `ValueHasChanged()` method. A full list of the commands supported is included in Appendix 10.2.

Overall, the code generator automates the process of creating a Radio Component allowing the programmer to concentrate on the implementation of radio functionality. This greatly reduces the time required to build and test components.

6.2.4 Dealing with Signals

An important issue when designing the Radio Component was how it would deal with signals, more specifically:

- How should numeric samples be stored?
- How should signals (multiple samples) be manipulated by components?

Samples must be stored using a data type that suits the particular application. The data type used to store samples must offer enough dynamic range to allow the full range of digital sample values to be represented in the radio system. Dynamic range is the ratio between the largest and smallest numbers that can be represented. For example, 16-bit integers offer the ability to represent numbers from –32768 to 32767 which corresponds to a dynamic range of approximately 96 dB. Devices such as DSPs and in particular FPGAs can be limited in the number of data types available. However the flexibility of the GPP offers a variety of data types, ranging from both signed and unsigned integers to floating point representations (see Figure 6.3).

Name	Bits	Range	Dynamic Range
Signed Integers	8	-128 to 127	48.1dB
	16	-32768 to 32767	96.3dB
	32	- 2147483648 to 2147483647	192.7dB
Unsigned Integers	8	0 to 255	48.1dB
	16	0 to 65535	96.3dB
	32	0 to 4294967295	192.7dB
Floating Point (Single Precision)	32	1.4×10^{-45} to 3.4×10^{38}	1668dB
Floating Point (Double Precision)	64	4.9×10^{-324} to 1.8×10^{308}	12630dB
And also complex number combinations of each, for example two 16-bit signed integers could be used to represent a complex number, thus resulting in 32-bits being used.			

Figure 6.3 – Data Types Supported by IRIS

The choice of data type influences the implementation of the overall application. For example, choice of data type can have a dramatic effect on the amount of memory used in the system with a move from 8-bit to 32-bit representation causing a quadrupling in memory requirements. Data types can also affect performance with differences occurring between calculations performed using

integer and floating point arithmetic. The performance hit may be due to capabilities of the underlying processor or to the fact that increased amounts of data need to be transferred to and from RAM.

IRIS components can support multiple data types, meaning that they can consume and produce signals of any of the supported data types. Internally, no restrictions are placed on the use of data types. Programmers are free to use techniques such as templates to develop generic algorithms that work with many different data types. However, care should be taken when doing this as moving between data types can introduce subtle errors caused by loss of precision. For example, if an algorithm is implemented using a double precision floating point number then moving to a signal precision floating point or even an integer data type will change the precision of the calculation. This can change the accuracy of the calculation and have an overall effect on the output of the algorithm.

For the IRIS system, a primary aim was flexibility and thus it was necessary to be able to inherently support multiple data types in the system. This raises problems however as the desire is to create components that are highly compatible, yet incompatible data types can break a system. To overcome this problem the IRIS components were designed so that they can accept and produce multiple data types but in a well defined way. A component exports a method called `GetDetails()` which the Component Framework uses to obtain information about the signal formats a component can produce and consume. The framework uses this information to verify the validity of a radio design by checking that the input and outputs between components are compatible.

Another flexibility issue is how signals or blocks of multiple samples are handled by the system. IRIS uses the common signal processing approach of treating signals as blocks of data. Blocks are stored in memory as a series of sequential samples. For DSP applications it is also useful to be able to represent signals using complex numbers. Complex numbers require the use of two numbers to correspond to the real and imaginary values of a complex number. IRIS inherently supports the data types and their corresponding complex combination of all the data types shown in Figure 6.3.

Another concern is multiple signals, as it is common for signal processing algorithms to produce or consume multiple signals. The problem is that IRIS must allow multiple signals to be represented and at the same time be able to ensure the validity of a radio configuration. For simple cases this is not a problem, for example an I (In-phase) and Q (Quadrature) signal is often represented as a complex signal therefore a complex data type can be used. But for implementations requiring multiple arbitrary channels, this method is not suitable. For example, a narrow-band channel extraction component may output eight channels of data from only one input.

One possible solution to support this is to use bigger block sizes (or a larger data type) to allow multiple signals to be combined together. However, this requires the programmer to implement code to combine and extract the multiple signals upon input or output, something that causes additional processing overhead and can lead to error-prone code. For this reason IRIS inherently supports channels. Channels allow a component to input or output multiple simultaneous channels of data. In memory IRIS stores samples and channels sequentially as depicted in Figure 6.4. By inherently supporting channels in the architecture, the programmer does not have to resort to a personal means of passing multiple signals between components.

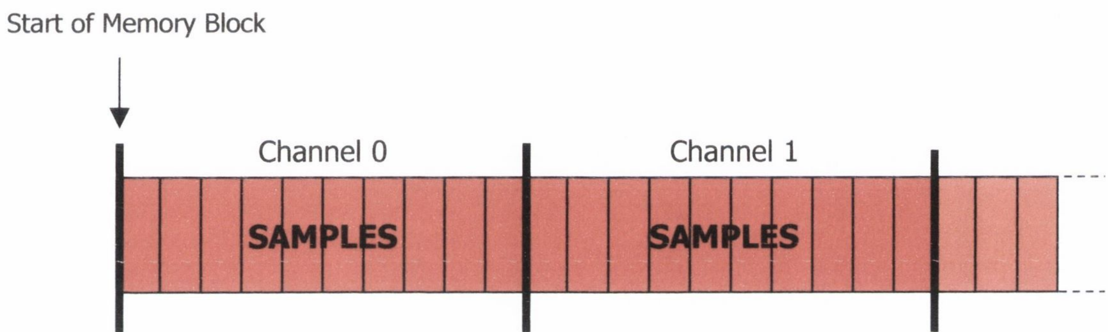


Figure 6.4 – Sequential Layout of Samples and Channels in Memory

6.2.5 Block Size and Sample Rate

The term 'block size' refers to the size of data used to transfer a portion of a signal between components. From early prototypes it became evident that multiple block sizes were required for the IRIS system for various reasons. Firstly, a fixed block size limits the ability to reuse a component in different scenarios, as it is difficult to combine code that requires different block sizes in a generic way. For example, an algorithm requiring a fixed block size of 200 samples will require alteration to deal with a block size of 201 samples. Secondly, block size is directly proportional to the latency of the system. Block size can be an important parameter in tuning the system to both application and performance requirements. Finally, some hardware devices (or their device driver implementations) can often specify a set range of block size values. By accommodating a varying block size it is possible to work with various different types of hardware input/output devices without having to alter components. For this reason, all IRIS components support variable input and output block sizes.

Developing algorithms to be variable in block size can come as quite a change to existing DSP developers. In many existing applications block sizes are fixed, especially in applications where the sample rate of the signal being processed does not change. For example, many audio DSP implementations always use the same sample rate of 44.1kHz, therefore static block sizes and

hence static latencies are common. DSP processor implementations mostly use fixed block sizes and the code implementations are fixed to a particular block size/latency.

In software-based radio applications however, the block size can change throughout the signal processing chain. This is because the sample rate of the signal often changes many times as it makes its way through the path of the receiver or transmitter. The receiver example in Chapter 5, Figure 5.3 (Section 5.8) was an example of a radio system in which the sample rate changes. In that case the signal of interest was down-converted and then down-sampled to a lower frequency. Down-sampling reduces the sample rate of the signal. Down-sampling (and up-sampling) occur often in radio applications as high sample rates require large amounts of data to represent a signal.

Manipulating the sampling rate of the signal path can have a dramatic effect on the performance of the system reducing the processing requirements by many orders of magnitude. In general, in the receiver the aim is to reduce the sample rate as soon as possible after reception. In the transmitter the aim is to increase the sample rate as late as possible before transmission. Each different radio scheme will have different signal characteristics and therefore there is no generic way to dictate how the sample rate can be manipulated. To address the problem of multiple sample rates and hence varying block sizes, the IRIS architecture inherently supports variability of these parameters. Where possible IRIS components are built to work at any sample rate, with any block size and with a variety of data types.

The Component Framework automatically handles all calculations involving block sizes, sampling rates and data types. This is illustrated in Figure 6.5. This diagram shows a sequence of components that produce and consume different numbers of samples. The blue boxes in the middle describe the function of each component and what it is configured to do. The green and yellow boxes on the right show how the data type, sample rate and number of samples are used in calculating the memory required to store the output of a component for a given input. For example, the Down Sampler component is configured to decimate the incoming signal by a factor of eight. Thus, the signal entering this component with a sample rate of 160kHz (160,000 samples per second) and a block size of 3200 samples will require a memory block of 6400 bytes. After decimation the output block size required is reduced, as the number of samples produced is 400 requiring a block size of 800 bytes. Similarly the 'Scale and Convert' component causes a different block size to be output as it changes the data type of the input signal from a 16-bit integer to a 32-bit floating point number.

These calculations are carried out in the initialisation phase of each radio system. The IRIS system starts at the beginning of the signal processing chain by looking at the output produced by the first component. Each component is given the sample rate, number of samples and data type it will be

rceiving. A component must then calculate the output it will produce for that given input. This means that the person designing the radio system only has to specify the sample rate and data type fc the first component. The engine automatically calculates the values required for the rest of the sstem based on this first component. If any mismatch occurs during this operation the IRIS system idicates an error and exits. This can occur when the data type produced by one component is not spported by the next. Another example is when a hardware device requires a fixed block size and a component either produces too many or too few samples. In this case a buffering component can b used or the designer can change the sample rate of the first component so that the correct block sie is produced where required.

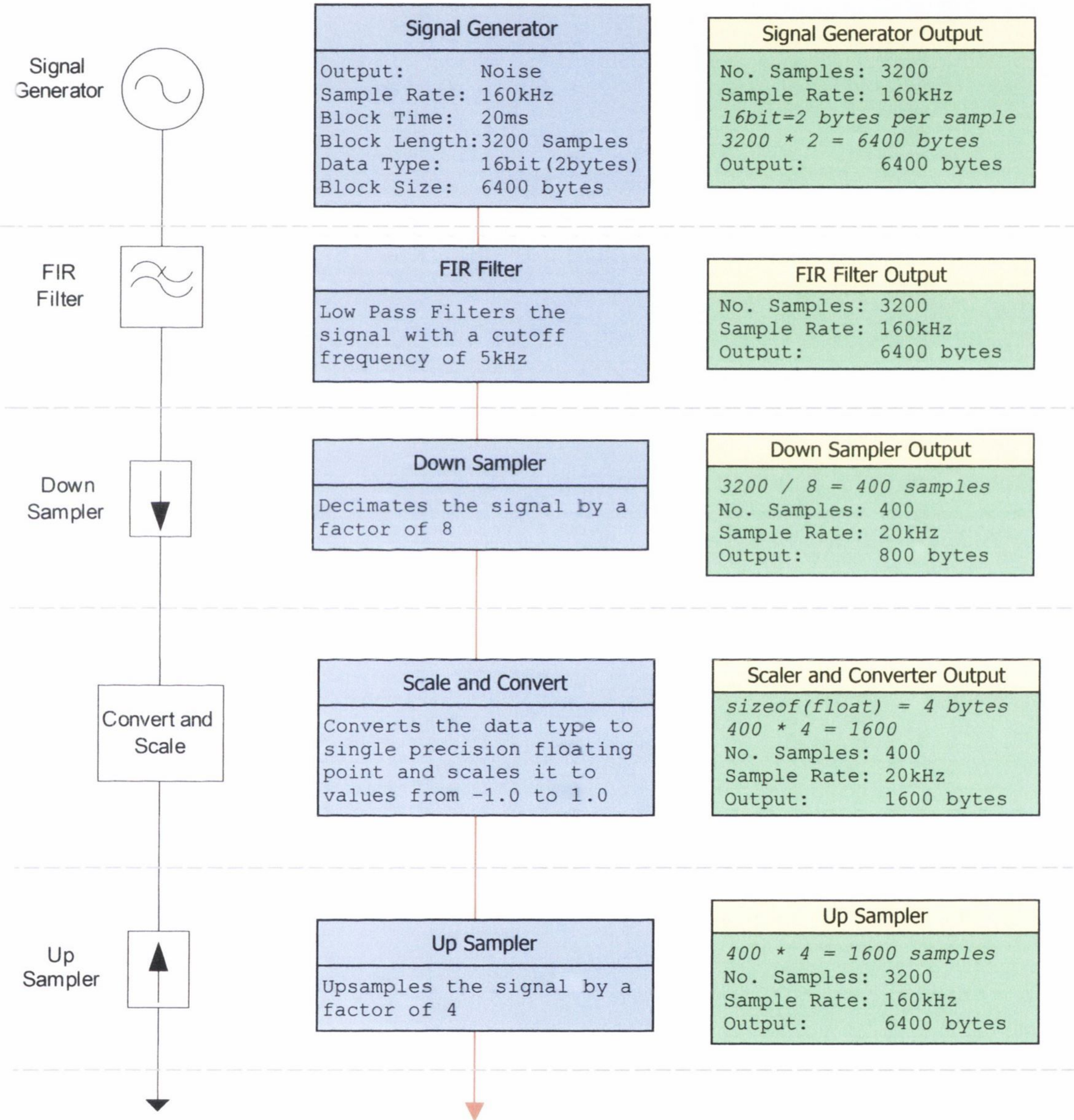


Figure 6.5 – Automatic Calculations Performed by the Framework

6.2.6 *Implementing Process()*

By implementing the `Process()` method a component makes available or exposes its functionality. Two types of `Process()` calls are available, in-place and not in-place. Signals are passed to a component via a data structure that contains pointers to memory that can be used to input or output data (see Figure 6.6). This data structure also provides access to the various input/output channels of the component and to numeric values which indicate timestamps and sample counts. These values can be used to implement specific timing logic in components.

```
struct Signal
{
	void *data;
	void *channel[MAX_NUM_CHANNELS];
	long timestamp;
	long samplestamp;
};
```

Figure 6.6 – Struct Definition used by Process()

To demonstrate a typical scenario, Figure 6.7 shows an simple `Process()` method of an in-place component that doubles the amplitude of the incoming signal.

```
void MyComponent::Process(Signal inout)
{
	float *sig = (float*)inout.data;
	for(int i=0; i<SignalFormatInput.blockSize; i++)
	{
		sig *= 2.0f;
	}
}
```

Figure 6.7 – Example Process() Method

During the initialisation phase of the radio each component is configured with a particular signal format which can be accessed by the `SignalFormat` data structure as shown in Figure 6.8. This data structure provides the developer with essential configuration information about block sizes, sample rates, channel information and data types. This information is used continuously by a component in its calculations.

```
struct SignalFormat
{
	int blockSize;
	int numChannels;
	int samplingRate;
	DataType dataType;
};
```

Figure 6.8 – Signal Format Struct

6.3 Radio Component Examples

6.3.1 Worked Example

To demonstrate how a practical Radio Component is created, the design of an FSK (Frequency Shift Keying) modulator is considered. The function of this Radio Component is to take data and to generate an FSK signal for further transmission by other components. This example demonstrates how the component can use the facilities available in the IRIS architecture to expose the functionality of a component in a generic way.

As discussed, each Radio Component in IRIS is defined via its parameters, events, ports and commands. Figure 6.9 below shows the properties that can be used to define an FSK component.

Property Type	**Description**
Parameters	

`SampleRate`
`BlockSize`
`CarrierFrequency`
`SignalFrequency1`
`SignalFrequency2`
`SymbolLength`
 	
Events
	The sample rate of the output FSK signal
	The number of samples to output from the component
	The frequency of the carrier signal
	The frequency of the first signal
	The frequency of the second signal
	The length in samples for one symbol

`DataModulated` Ports	Fired when the data has been modulated
`ModulateData` Commands	When data is sent to this port it is modulated using FSK
`Reset`	Resets the modulator aborting all transmissions

Figure 6.9 – Properties of FSK Component

To expose this information from the component the code in Figure 6.10 is written. From the header file in the figure, the code generator generates the XML definition of the component as shown in Figure 6.11. The code generator can then use this XML description to generate all the code required for the component to be used by the Component Framework (as discussed in Section 6.2.3).

```
//@component modulates data using Frequency Shift Keying
//@version 1.1
//@author Philip Mackenzie
//@port ModulateData modulates the data sent through the port
//@event DataModulated int fired when the data has been modulated
//@command Reset resets the modulator
class FSKModulatorComponent : public IOComponent
{
private:
        //@param whether to output debugging information
        //@default false
        bool debug;

        //@param what block size to produce
        //@default 2048
        int BlockSize;

        //@param the sampling rate in samples per second
        //@default 44100
        int SamplingRate;

        //@param the first frequency in Hz
        //@default 600
        //@dynamic
        int SignalFrequency1;

        //@param the second frequency in Hz
        //@default 1200
        //@dynamic
        int SignalFrequency2;

        //@param the symbol length of a bit in Hz
        //@default 300
        //@dynamic
        int SymbolLength;

        //@param the frequency of the training carrier in Hz
        //@default 900
        //@dynamic
        int CarrierFrequency;

. . . . Truncated
```

Figure 6.10 – C++ Header File Definition of FSK Modulator Component

To implement the actual signal processing code of the FSK component the developer must implement the lifecycle of the component as discussed in Section 5.4.3 and the methods to receive data. For example, Figure 6.12 shows the code used to respond to data sent to its port. When data is received the component allocates memory for the data and copies it to this location. It then signals an event to indicate that modulation should occur and waits for this to complete. This is required as information can arrive into ports asynchronously and thus this code has to wait until calls to `Process()` have occurred to modulate the data. When completed it fires an event using `ActivateEvent()` to notify externally subscribed code that processing has completed.

```xml
<component type="fskmodulator">
	<description>
		<name>FSKModulator</name>
		<author>Philip Mackenzie</author>
		<version>1.1</version>
		<information>modulates data using Frequency Shift Keying</information>
	</description>
	<parameters>
		<parameter name="BlockSize" type="int" id="VALUE_BLOCKSIZE">
			<description>what block size to produce</description>
			<dynamic>no</dynamic>
			<bytes>4</bytes>
			<array>no</array>
			<default>2048</default>
		</parameter>
		<parameter name="SamplingRate" type="int" id="VALUE_SAMPLINGRATE">
			<description>the sampling rate in samples per second</description>
			<dynamic>no</dynamic>
			<bytes>4</bytes>
			<array>no</array>
			<default>44100</default>
		</parameter>
		<parameter name="SignalFrequency1" type="int" id="VALUE_SIGNALFREQUENCY1">
			<description>the first frequency in Hz</description>
			<dynamic>no</dynamic>
			<bytes>4</bytes>
			<array>no</array>
			<default>600</default>
		</parameter>
		<parameter name="SignalFrequency2" type="int" id="VALUE_SIGNALFREQUENCY2">
			<description>the second frequency in Hz</description>
			<dynamic>no</dynamic>
			<bytes>4</bytes>
			<array>no</array>
			<default>1200</default>
		</parameter>
		<parameter name="SymbolLength" type="int" id="VALUE_SYMBOLLENGTH">
			<description>the symbol length of a bit in Hz</description>
			<dynamic>no</dynamic>
			<bytes>4</bytes>
			<array>no</array>
			<default>300</default>
		</parameter>
		<parameter name="CarrierFrequency" type="int" id="VALUE_CARRIERFREQUENCY">
			<description>the frequency of the training carrier in Hz</description>
			<dynamic>no</dynamic>
			<bytes>4</bytes>
			<array>no</array>
			<default>900</default>
		</parameter>
	</parameters>

	<events>
		<event name="DataModulated" type="int" id="EVENT_DATAMODULATED">
			<id>EVENT_DATAMODULATED</id>
			<description>fired when the data has been modulated</description>
		</event>
	</events>

	<ports>
		<port name="ModulateData" id="PORT_MODULATEDATA">
			<id>PORT_MODULATEDATA</id>
			<description>modulates the data sent through the port</description>
		</port>
	</ports>

	<commands>
		<command name="Reset" id="COMMAND_RESET">
			<id>COMMAND_RESET</id>
			<description>resets the modulator</description>
		</command>

	</commands>

</component>
```

Figure 6.11 – XML Generated to Describe the FSK Modulator Component

```
bool FSKModulatorComponent::ProcessPortData(int portId, unsigned char* data, int length)
{
	if(portId == PORT_MODULATEDATA)
	{
		//Allocate memory and copy in received data
		dataToTransmit = new unsigned char[length];
		dataToTransmitLength = length;
		memcpy(dataToTransmit, data, length);

		//Signal that the data should be modulated and wait
		//until it has been completed
		SignalObjectAndWait(hEventWait, hEventComplete, INFINITE, FALSE);

		//Deallocate memory
		delete [] dataToTransmit;
		dataToTransmit = NULL;
		dataToTransmitLength = 0;

		//Fire event to indicate to external subscribers that
		//the data has been modulated
		ActivateEvent(EVENT_DATAMODULATED, length);

		return true;
	}

	return false;
}
```

Figure 6.12 – Code to Implement Data Received Port

When `Process()` is called it must generate an FSK waveform as shown in Figure 6.13. This may require multiple calls to `Process()` as the signal may span multiple blocks of data. The component can make use of the lifecycle of the component to prepare itself for processing. For example, a component can use the `Init()` method to pre-calculate lookup tables which can be used to generate the waveform. This reduces the amount of processing required in the `Process()` method to generate the FSK waveform.

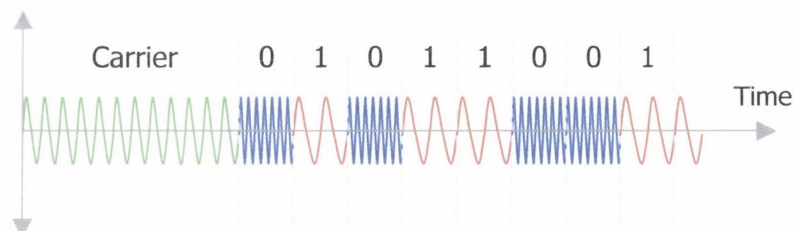


Figure 6.13 – FSK Waveform

Once the FSK component has been compiled as a DLL it can be used in any radio configuration using the XML shown in Figure 6.14.

```xml
<component type="FSKModulator" instance="MyModulator">
	<parameters>
		<BlockSize>40960</BlockSize>
		<SampleRate>250000</SampleRate>
		<SignalFrequency1>10000</SignalFrequency1>
		<SignalFrequency2>30000</SignalFrequency2>
		<CarrierFrequency>20000</CarrierFrequency>
		<SymbolLength>30</SymbolLength>
	</parameters>
</component>
```

Figure 6.14 – XML for Configuring an FSK Modulator Component

This example has demonstrated the development of a single component, however over fifty components have been developed for use in the IRIS system for use in this and other related research. The next sections briefly outline some of the more interesting of these that provide insight into the reconfigurable radio concept.

6.3.2 Signal Processing Components

A variety of components have been written implementing the standard functions required in radio systems. Examples being:

`ChannelExtractor`: This component extracts a channel of interest from a wideband source

`FIRFilter`: Filters a signal using an FIR filter

`DownSampler`: Down samples (decimates) a signal to a lower sample rate

`UpSampler`: Up samples a signal to a higher sample rate

`SignalScaler`: Scales (amplifies or attenuates) a signal by a specified amount

`SignalDetector`: Detects when a signal is present

These components can be used as fundamental building blocks when designing many radio systems.

Modulation and demodulation are also catered for with a variety of analogue and digital schemes. The simplest of these allow operation with AM and FM analogue signals, a variety of digital schemes such as the FSK example presented, but also other researchers have concentrated on using IRIS for more complex modulation schemes such as OFDM (Orthogonal Frequency Division Multiplexing) [Nolan2003a, Nolan2003b, Nolan2003c, Nolan2003d]. This work has produced promising results demonstrating that a generic OFDM component can be written which can be reconfigured to work with a variety of operating parameters.

Also in related research to this work, techniques for performing automatic modulation detection have been developed. These allow a radio system to reconfigure itself dynamically according to the detected modulation scheme of the incoming signal. The IRIS system has been used as the basis for

this work and further information can be found in [Nolan2001, Nolan2002a, Nolan2002b, Nolan2002c].

Specific applications have been catered for too; for example components have been built to handle specific 2-way radio systems and another allows decoding of a proprietary data communications module. Another researcher has used IRIS to build an RDS (Radio Data Signal) [Flood2003]. This receiver reuses some of the standard signal processing components mentioned above in addition to a 'Costas Loop' component and 'RDS Decoder' component.

6.3.3 IO Components

A variety of components have been written to allow both the input and output of signals using external hardware. Section 6.5 will discuss the use of IRIS with hardware in more detail, but in terms of the components involved there are a new points worth noting.

IO components specifically cater for the input and output of signals. The idea is that any signal source or signal output can be encapsulated as a standard component. This means that IO components can be used interchangeably in a system to process signals in different ways. While most radio systems will be built for one particular piece of input/output hardware (or RF front-end), during the testing phase it is advantageous to be able to route signals to different hardware. This can be achieved in the IRIS system by replacing the IO component to route the signal to another piece of hardware to even to write it to a file.

The components currently written for the IRIS system allow the input and output of signals to a variety of PC hardware. Examples are:

`A2DPCI4020`:	A component allowing input from a 20MHz PCI A/D Converter
`DAC0412HS`:	A component allowing output to a 250kHz D/A Converter
`WaveOut`:	Allows audio output using Windows audio API
`WaveIn`:	Allows audio input using Windows audio API
`DirectXOut`:	Allows audio output using DirectX API
`DirectXIn`:	Allows audio input using DirectX API
`ASIOOut`:	Low latency audio output using the ASIO standard
`ASIOIn`:	Low latency audio input using the ASIO standard

In addition to higher frequency DAC and ADC converters, this list shows that components have been written to accommodate a variety of audio standards. Although this type of hardware does not allow operation at RF frequencies, they are useful in testing the basic functions of the system. ASIO (Audio Streaming Input Output) has been particularly useful [Steinberg99]. ASIO is a

standard for low latency audio allowing deterministic input and output of digitised audio signals in real-time. This provided a valuable test environment for verifying the functionality of the IRIS system as its low latency operation better mimics an RF front-end even though it operates at a lower sampling frequency. A practical radio system using RF frequencies is discussed in Section 6.5.

6.3.4 Testing Components

A series of components have been developed for testing purposes. Some of these are particularity interesting, as they do not have counterparts in the analogue world. For example, of great importance has been the ability to write and read digitised RF waveforms directly to/from the hard disk. This effectively allows the recording of RF signals that can be processed later offline by re-reading in the waveform from a file. This makes it very easy to test the implementation of Radio Components and complete systems as a receiver can be tested against real test signals offline without having to use external RF test equipment. For example, the ADC card allows the digitisation of a large bandwidth up to 10MHz, and using a `FileWriter` component this signal can be written to a file. In this raw state the signal of interest can be analysed later to assess its frequency content and to perform tuning in software by extracting different signals from the wideband source. This changes the radio design paradigm in that radio signals become much more accessible and facilitates the development of new types of radio systems in creative ways.

Testing components include:

`FileReader:`	Reads a waveform from the hard disk
`FileWriter:`	Writes a waveform to the hard disk
`NumericAnalyser:`	Performs analysis on the numerical content of signals
`SignalGenerator:`	Allows the output of a variety of signals at any frequency
`SignalAboveThreshold:`	Fires events when a signal is above a threshold
`Delay:`	Introduces a delay between blocks
`SystemStatistics:`	Provides information about the CPU time being used by a radio

6.3.5 Visualisation Components

Visualisation components have been written allowing the signal to be viewed and analysed at any point in the radio system. By simply moving the component through the radio structure (or by using multiple visualisation components), the user can inspect the signal at any point in the radio. As an example, Figure 6.15 shows the output of a spectrum analyser component plotting the power spectrum of an FM signal after demodulation. This diagram clearly shows the constituent parts of a

broadcast FM signal, namely the 19kHz pilot tone and 38kHz DSB-SC (Double Side Band-Suppressed Carrier) stereo signal.

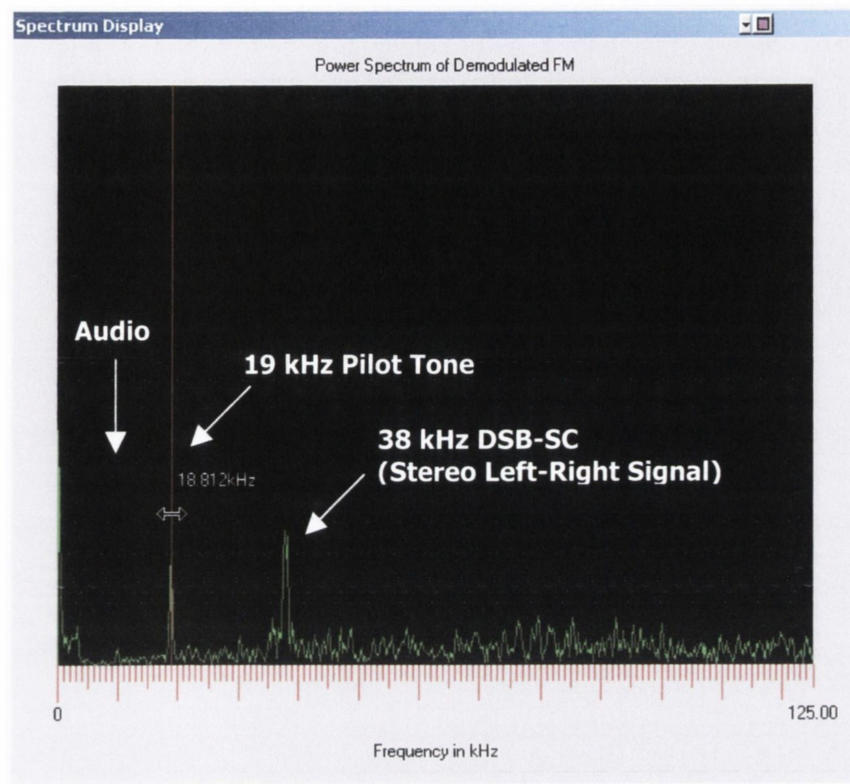


Figure 6.15 – IRIS Screenshot of Received FM Signal

Visualisation components include:

`SpectrumDisplay:`	Performs an FFT displaying the power spectrum of a signal in real-time
`Oscilliscope:`	Traces the waveform of a signal in real-time
`PeakMeter:`	Displays the amplitude and peak value of a signal in real-time

6.4 Using the Component Framework

6.4.1 IRIS API

While the Component Framework could have been implemented as a standalone entity, it is of better use when combined with other software. For example, it may be necessary to integrate a reconfigurable radio into another system that requires wireless communication. The IRIS system caters for this through the IRIS API. As introduced in Chapter 5, Section 5.5.5, the IRIS API abstracts the programmer from all the details of the underlying system effectively encapsulating the

framework into a reusable sub-system. This allows someone to use IRIS without requiring specific knowledge of radio technology.

Figure 6.16 shows sample code for creating a new reconfigurable radio using the IRIS API. This example shows how to configure the framework, for example setting the components directory. A call to `IRISLoad()` loads the radio configuration into the framework. At this point the framework verifies the radio design and brings together the Radio Components and Control Logic to form the system. A subsequent call to `IRISStartRadio()` starts flow of signals through the radio system.

At this high level no knowledge of radio systems is required as the description of the radio and associated Control Logic is contained within the radio configuration.

```
bool CreateReconfigurableRadioExample()
{
	//Initialise the IRIS sub-system
	IRISInitSystem();

	//Create a radio engine
	HANDLE_IRIS_ENGINE hRadio = IRISCreateEngine();

	//Redirect the log output to receive logging messages
	IRISRedirectLogOutput(hRadio, IRISLogOutput);

	//Tell the framework where the components are
	IRISSetComponentsDir(hRadio, "c:\IRIS\components");

	//Load the radio
	if(IRISLoadRadio(hRadio, "MyRadio.xml") == false)
	{
		char *error = IRISGetError(hRadio);
		printf("An error occured loading the radio: %s\n", error);
		return false;
	}

	//Start the radio
	if(IRISStartRadio(hRadio) == false)
	{
		char *error = IRISGetError(hRadio);
		printf("An error occured starting the radio: %s\n", error);
		return false;
	}

	return true;
}
```

Figure 6.16 – Code to Create a Reconfigurable Radio

While the IRIS API allows implementations to be abstracted from the underlying system, there are occasions when an application using IRIS as a sub-system may require full interaction with particular components in the radio system. For example, when used in a communications stack an application may need to transfer packets to and from the radio system. As another example a graphical-based radio system may need to control a component to change a frequency setting or alter the properties of a filter.

To facilitate this an application that uses the IRIS sub-system may create its own control logic in addition to the control logic of the radio system itself. Figure 6.17 illustrates this. The diagram on the left shows an application that uses the IRIS sub-system, fully abstracted from the internal operation of the radio. The diagram on the right shows how additional control logic can be specified by an application allowing it to control components and receive information from the radio system.

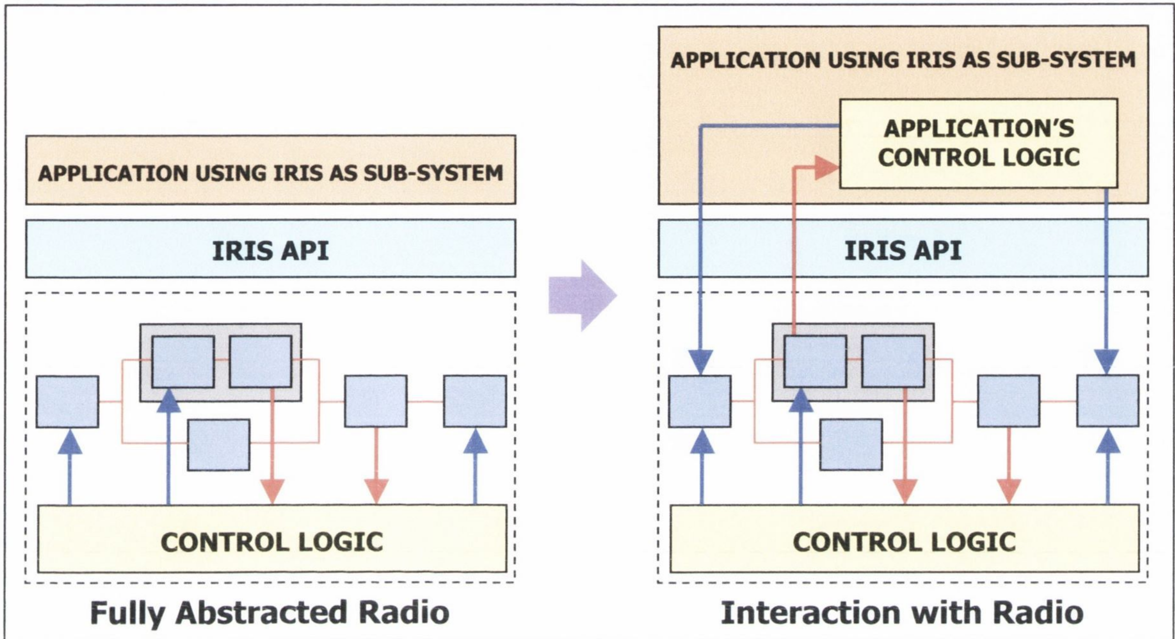


Figure 6.17 – Application Specified Control Logic

Additional control logic can be attached to a radio system simply by using the IRIS API. Figure 6.18 shows a code sample of a control logic controller and how the IRIS API can be used to attach this control logic to a radio system. This technique is very useful for graphical applications as it allows the internals of a radio system to be viewed and changed dynamically at runtime.

Figure 6.19 shows a screenshot of an application that uses this facility to expose the parameters of a radio system, in this case displaying the parameters exposed from an FM receiver. This application creates additional control logic as described above and attaches it to the receiver. It then uses the reflection interface of each component (see Section 5.4.2) to query information about its parameters. Using this information it constructs a user interface and creates separate graphical controls for each parameter of the radio. This allows the user to dynamically change any parameter of the radio, a useful tool for experimentation and development.

```
//Application specified control logic
class ApplicationController : public ControllerInterface
{
private:
        EngineInterface *engine;
public:
        //Called when the radio is being loaded
        virtual void Load(EngineInterface *eng)
        {
                engine = eng;
        }

        //Called to initialise control logic
        virtual bool AttachToComponents()
        {
                //Find the channel extraction component and
                //set its frequency to 2MHz
                HANDLE_COMPONENT hComponent = engine->FindComponent("ChannelExtractor");
                HANDLE_PARAMETER hParameter = engine->FindParameter("MixerFrequency");
                engine->SetParameterValue(hComponent, hParameter, 2000000);
        }

        //Called during unload
        virtual void Unload()
        {
        }
};

. . . Truncated

//Create instance of controller
ApplicationController *controller = new ApplicationController();

//Apply the control logic to the current radio configuration
IRISSetControlLogic(hRadio, controller);
```

Figure 6.18 – Sample Code for Creating Application-Defined Control Logic

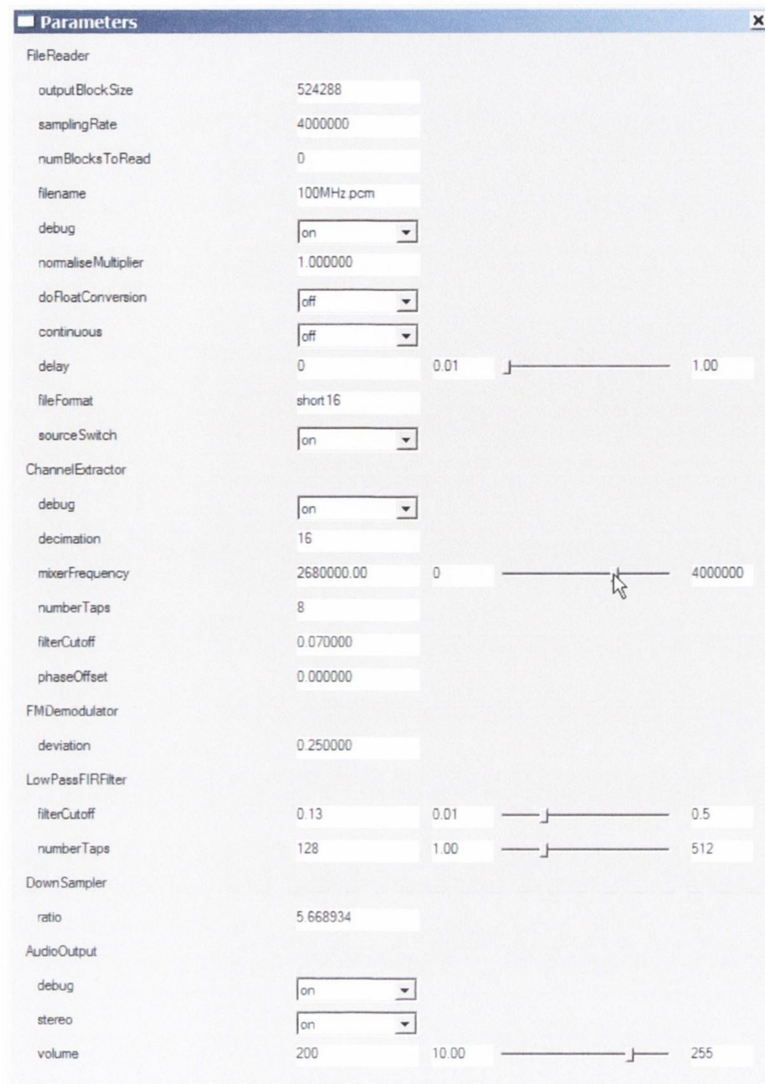


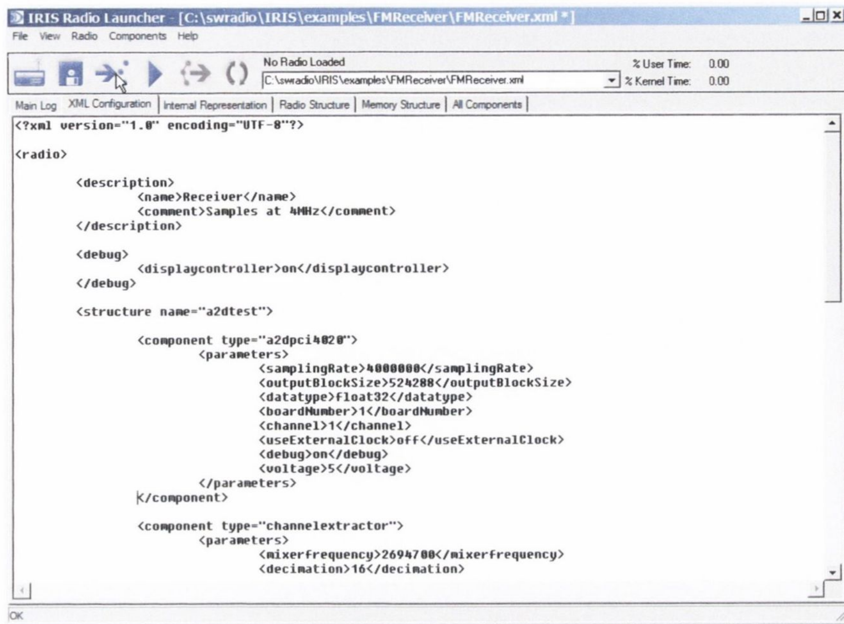
Figure 6.19 – Screenshot of Parameter Controller

6.4.2 Tools

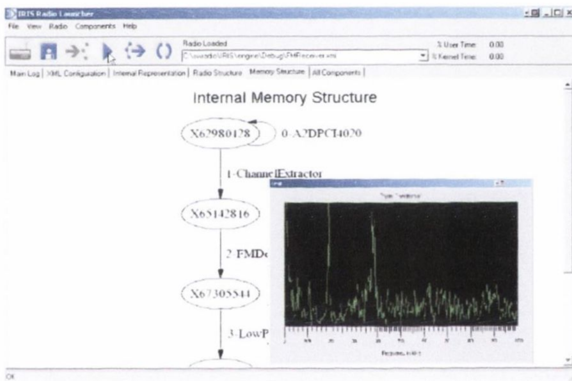
Using the IRIS API two tools have been written which automate many of the procedures in loading and running a radio system. The launcher is a standalone executable that encapsulates the Component Framework. This command line application uses the IRIS API to load and control radio systems. This tool provides various facilities for debugging and testing individual Radio Components and complete radio systems.

One of the advantages of using GPPs is the ability to have rich graphical user interfaces allowing interaction with the internals of the radio system. To demonstrate this the IRIS Radio Designer was written, a graphical user interface built also with the IRIS API that allows users to design and test radio systems interactively. Screenshots of the system are shown in Figure 6.20. The screenshots show various functions of the radio designer and how it can be used to edit radio configurations,

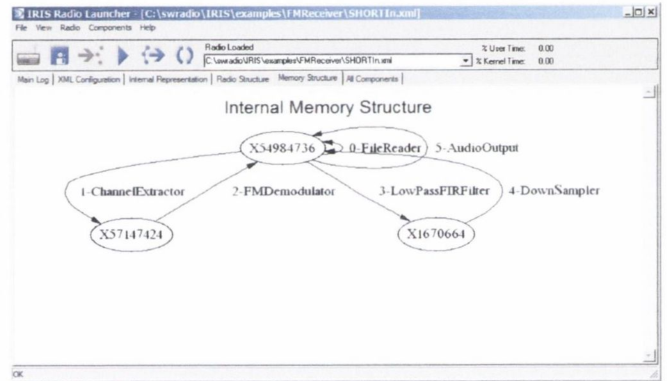
graphically visualise signals in real-time and view various graphical representations of a working radio system.



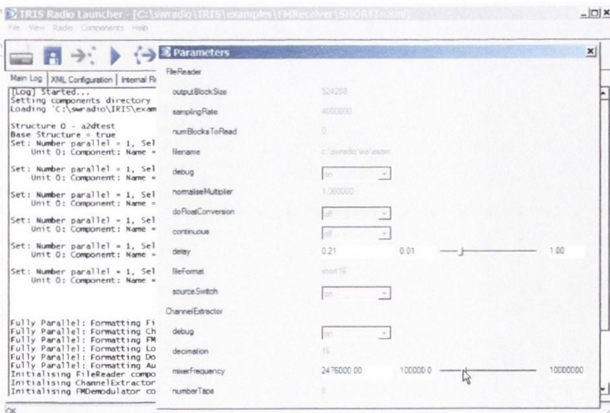
Radio Designer Showing XML Configuration



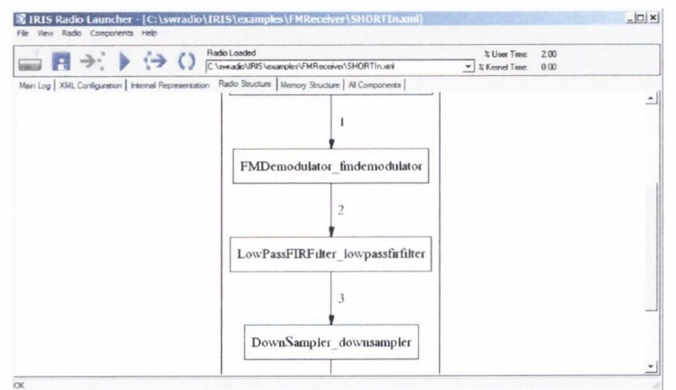
Displaying Real-time Spectrum



Viewing Internal Memory Structure



Reconfiguring Parameters in Realtime



Viewing Structure of Radio System

Figure 6.20 – Radio Designer User Interface Screenshot

6.5 External Hardware

The IRIS system as implemented provides the software infrastructure for building radio systems, however to effectively test and experiment with real signals it was necessary to integrate the system with a real RF front-end. Various hardware setups and associated Radio Components have been developed enabling the input and output of RF and audio signals.

Radio Components have been developed allowing signals to be input from hardware and output to hardware. For example, a data acquisition component allows the input of digitised signals from an ADC (Analogue to Digital Converter) PCI (Peripheral Component Interconnect) card. This component is implemented as an IRIS IO component with parameters used to control variables such as the sampling rate of the converter and input voltage settings. Each type of hardware component encapsulates the hardware interface through the standard IRIS configuration mechanism (as discussed in Section 5.5). For example, the XML configuration file may be used to indicate the sample rate of an ADC. The component uses the value received from the framework to initialise the hardware using the programming library provided by the original manufacturer of the board. The advantage of encapsulating this functionality into an IRIS component is that it can be re-used in many different designs.

In the experimental prototype developed for this work, a hardware setup allowing the reception of RF signals has been developed (see Figure 6.21) and this forms a basic RF front-end. A commercial wideband receiver is used to tune to a frequency of interest. The 10.7MHz IF signal is available from the receiver and this signal is amplified, digitised, filtered and fed to the input of the ADC card. Using band pass sampling at the appropriate rate (typically a 4MHz sampling rate), the IF signal is digitised allowing the remainder of receiver functionality to be implemented in software.

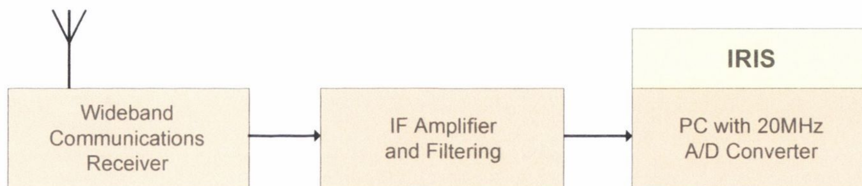


Figure 6.21 – Receiver Hardware Setup

To demonstrate how the hardware works, Figure 6.22 shows a photograph of the hardware consisting of the wideband receiver, IF amplifier and ADC PCI card. Superimposed on this figure is the component structure for typical receiver architecture. This picture illustrates how IRIS forms the infrastructure between hardware, software and also the user of the software.

In terms of performance, for example using a 2GHz Pentium IV processor, the IRIS system can digitise a signal at 4MHz using band pass sampling, extract a channel of interest, and perform FM

demodulation and audio playback. Using un-optimised code within components this consumes approximately 60% of processor time. Performance and related issues are discussed in the next section.

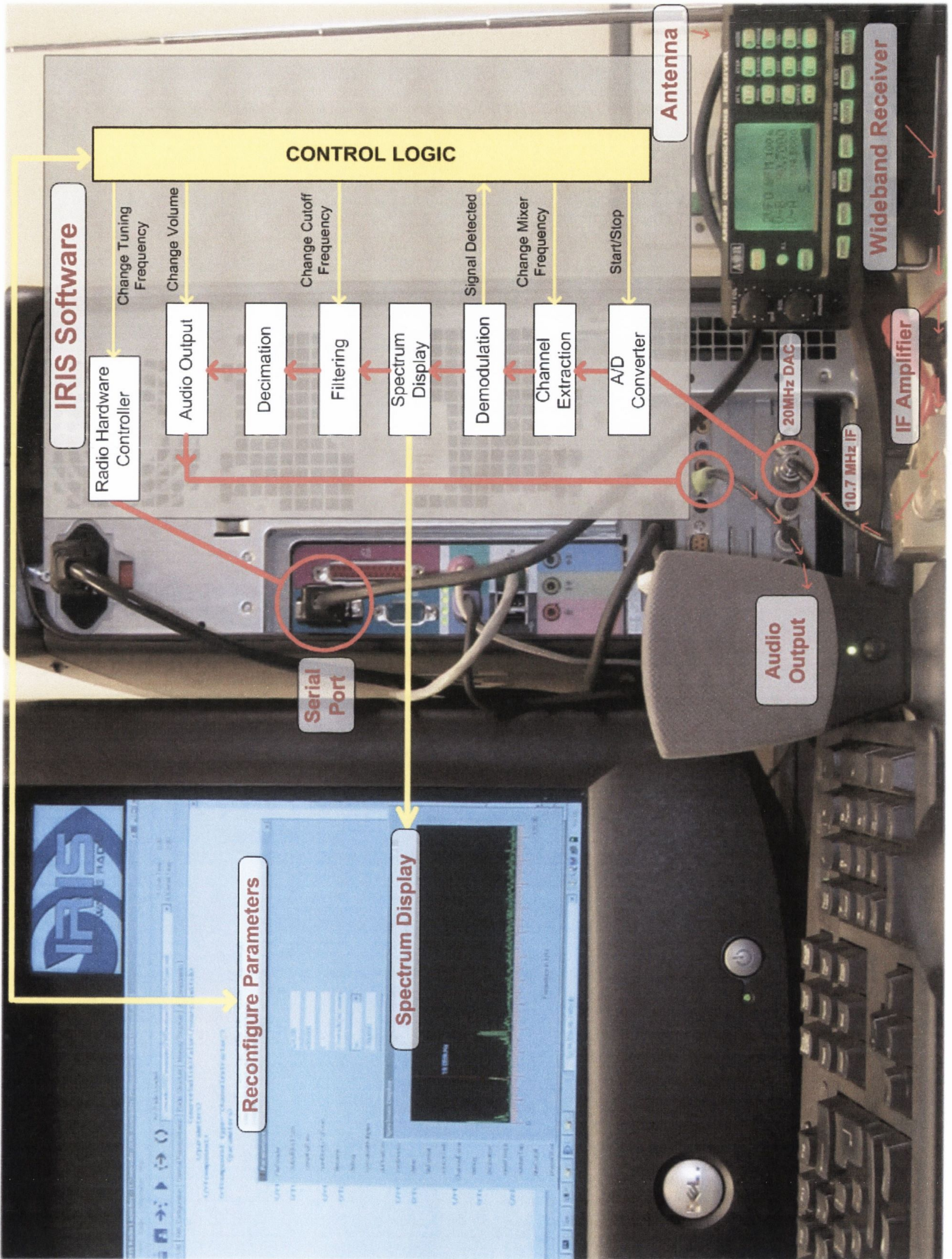


Figure 6.22 – IRIS Test Hardware

6.6 Analysis

The purpose of the IRIS system has been to demonstrate the concepts of reconfigurability through the use of the IRIS Component Framework. Performance of individual signal processing algorithms has not been a primary concern as the onus is on developers to write high-performance Radio Components. (Performance in this context refers to how efficiently code executes on a processor). While IRIS does not specifically dictate policies for writing highly efficient DSP code, as a framework it should not hinder the overall performance of a radio system. Highly efficient components are of no use without an efficient and scalable technology for combining these components together.

To address this issue the IRIS implementation was developed in such a way that it does not hinder the performance of the overall system and attempts to keep the overhead of the framework to a minimum. The following sections analyse various aspects of the framework with respect to performance and scalability.

6.6.1 Scalability

Scalability is the measure of how well a system performs as it grows in size or as more demand is placed on the system. The aim is to create systems that scale linearly so that as system size or load increases the processing power required increases linearly. A system that features bad scalability will thus have an exponential or even unpredictable response to increasing demand. For example, for a web server, scalability is measured by load, i.e. the number of requests being received for web pages. As more users request web pages the processing time required to deal with these requests must increase linearly. Also, as more processing power (e.g. additional CPUs) is added to a server its load capacity should increase linearly.

In implementing the IRIS architecture it was important that scalability was considered. The aim was to ensure that as more increasingly complex radio systems are developed, involving the use of more Radio Components, the processing time of the system should not hinder the overall performance of the system. In the IRIS architecture it is the Radio Engine that has the potential to hinder scalability as it provides the interconnection between components. The engine naturally introduces a processing overhead and thus as radio systems grow more interconnections are required between components. It was thus important to implement the engine in such a way that this overhead is kept to a minimum and where possible to make this overhead scale linearly.

In the IRIS system basic configurations of components scale linearly. This is achieved by storing the structure and interconnection between components in a highly efficient and scalable data

structure. In the object-orientated approach of storing the radio structure (see Section 5.5.4) the structure of the radio is stored in a high-level representation that can represent a complex hierarchy of components. In this form the radio layout is stored in memory just as it appears in the XML file. C++ classes are used to represent each entity, examples being the `<structure>`, `<component>` and `<parallel>` tags. In this form finding a particular component and its interconnections takes a non-deterministic amount of time. The hierarchy must be searched to determine interconnections between components. It is not scalable to require such a search each time a component produces an output.

Instead, the Radio Engine pre-processes and converts the object-oriented structure into a highly efficient linked list data structure and in the process pre-determines the interconnection among components. When this process is complete the engine has a linked list in memory, the nodes of which correspond to the path of the signals rather than the visual hierarchy of the radio. This means that when passing signals between components the engine only has to walk a basic linked list rather than having to perform un-deterministic searches of data. The advantage of this technique is that more interconnections can be added between components without hindering scalability, as the process of traversing the linked list is always linearly scalable. The disadvantage is the increased code and complexity required to implement this scheme.

A practical experiment was carried out to test the scalability of the IRIS implementation. The aim of the experiment was to plot the CPU processing time required to implement radio systems of various sizes to determine if the processing time scales linearly. A simple DSP configuration was constructed using a signal generator followed by multiple FIR filter components (Figure 6.23). The FIR filter component was used, as filtering is the most commonly used signal processing algorithm. Each separate test involved the generation of a 40kHz sine wave (sampled at 250000 samples per second) that was passed through multiple FIR filters. Five FIR filters were added at a time to the XML configuration until the full processing power of the CPU was reached. The percentage CPU time was measured using calls to the Win32 method `GetProcessTimes()`. In this test CPU percentage refers to the percentage of user mode CPU processing time spent processing 254952 samples during a one second interval. The test was performed on a 2.8GHz Pentium 4, 266MHz DDR 1GB SDRAM running Windows XP. All tests involved code that was compiled with full optimisations with the Microsoft Visual Studio .NET C++ compiler.

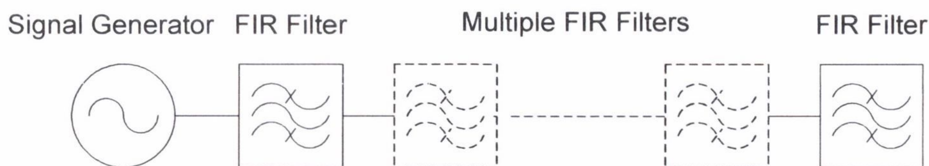


Figure 6.23 – Scalability Test Scenario

The results of the scalability test are shown in Figure 6.24. The red line indicates the percentage of CPU time required to process increasing numbers of FIR filters. The linearity of the red line indiates that the IRIS system achieves linear scalability for this basic set of components.

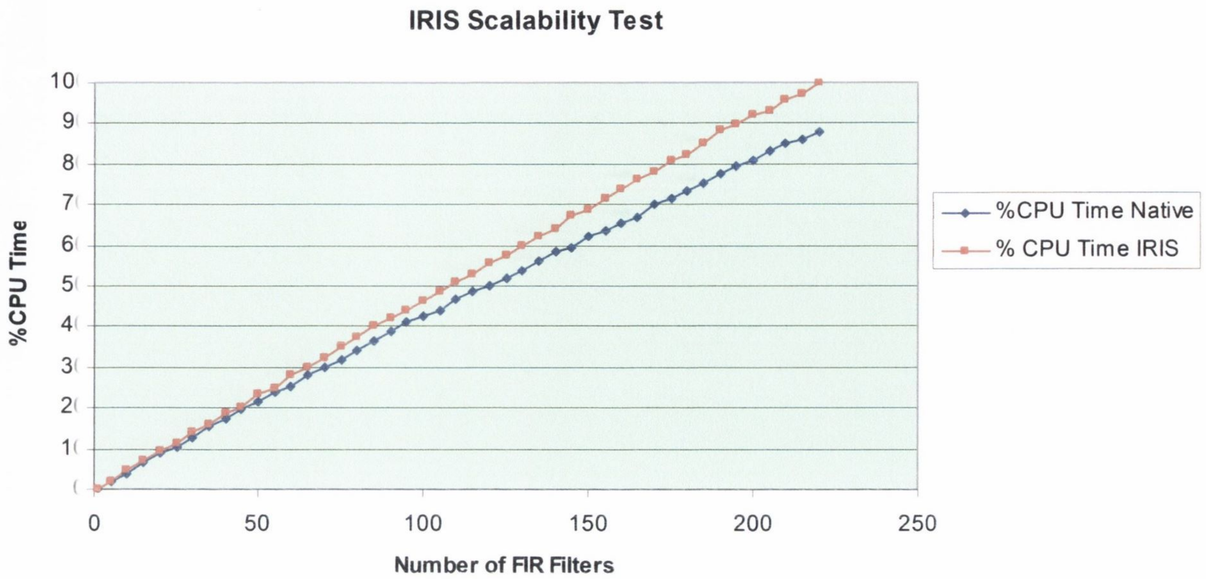


Figure 6.24 – IRIS Scalability Test Results

To investigate the overhead associated with using the Radio Engine a second scalability test was performed. The aim of this test was to quantify the overhead of using a Component Framework as opposed to implementing code as native executables. For this test the code of the signal generator and FIR filter were extracted from the components and implemented as a standalone native executable. Functionally the two systems produced identical results but one was built using the general-purpose Component Framework and the other was a purpose built executable. In the native implementation there is no overhead associated with decoupling components as there is in the IRIS system and the result of this can be seen by the blue line in Figure 6.24. The difference in CPU time between these two approaches in graphed in Figure 6.25. This graph demonstrates that the price of using IRIS is a processing overhead that increases linearly as more sequential components are added to the configuration.

Although the difference plot shown in Figure 6.25 demonstrates a 10 to 12% CPU overhead for 200 components or more, in reality most radio implementations will require much less and will rarely go beyond 20 or 30 components. The graph indicates that up to 50 components the overhead associated with using IRIS is on the order of 0 to 2% for the test system used, a figure which is mininal and in the vast majority of cases an acceptable price to pay for the facilities the archiecture provides.

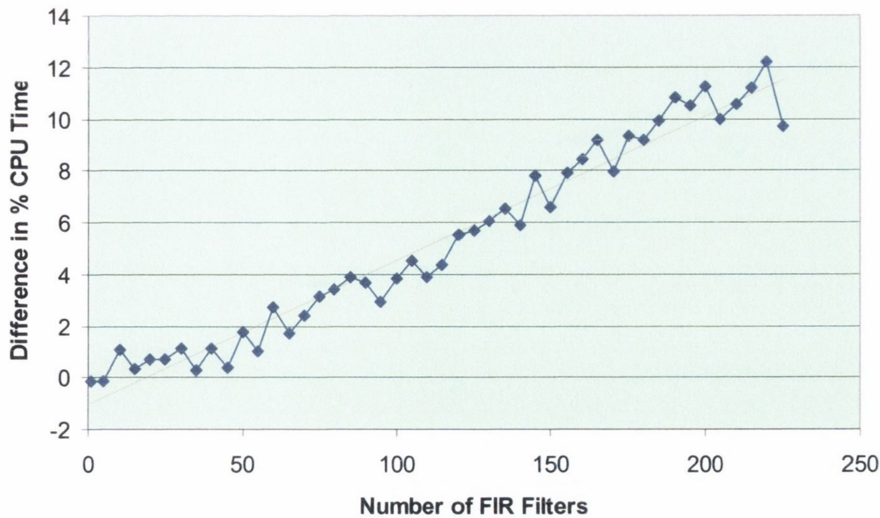


Figure 6.25 – Difference between IRIS and Native Implementation

These tests have analysed only sequential chains of components for scalability. It should be noted that more complex examples involving signal duplication and branching would not necessarily demonstrate linear scalability, as these features would require further processing overhead between components. As scalability in itself is not a focus of this thesis, these basic examples should serve as a basis for future work into analyzing more complex sets of components.

6.6.2 Memory Consumption

An important consideration for IRIS was the memory consumption of the system. The use of memory in a software radio can have a direct impact on the performance of the radio system. In the IRIS design the allocation and designation of memory for use by components is controlled by the Radio Engine, therefore it was important to implement a memory policy that worked well across multiple types of radio applications.

The considerations for memory were twofold, firstly to reduce memory copying and secondly to reduce the amount of memory used. The copying of memory can be expensive in processing time due to the large amounts of data inherent to software radio. Since IRIS allows the use of any number of components in a variety of hierarchies and structures, it was important the IRIS system did not rely on copying of signals between components during operation. The amount of memory used had to be kept to a minimum. In a GPP-based operating system virtual memory is used to increase the amount of memory available to applications. This works by swapping data from RAM to disk when memory is low. When the swapped memory is requested by an application it must be transferred back into memory making the overall time to access memory much slower. With excessive amounts of memory typically used by software radio applications it was important to develop a radio infrastructure that used memory in an efficient way.

The basic approach developed in managing memory for the implementation of IRIS was to allocate memory in such a way that the output of one component is written directly to the input of another component. For simple radio configurations this approach is simplistic, however the problem can become more complex for the following reasons:

- Size: Components in a radio can consume and produce different block sizes therefore different sized memory blocks are required throughout the radio.
- Multiple Signals: The hierarchical approach used for expressing radio systems allows multiple signals to exist simultaneously in the radio system, therefore the radio system must facilitate the use of multiple memory allocations.

In addition IRIS allows complex hierarchies of components to be implemented and thus the output of one component must be mapped to the input of one or more other components while trying to make best use of memory.

To address these issues a specific memory algorithm was developed within the Radio Engine. This algorithm attempts to efficiently reuse memory throughout a hierarchy of components while keeping memory copying to a minimum. The algorithm works by predetermining the path of signals through a component hierarchy before radio operation begins. With this knowledge the engine can allocate the same memory block to multiple components without having to worry about signals being overwritten. To achieve this the engine contains a memory manager, a sub-system that allows memory to be allocated, locked and released. During the construction phase of the radio, the engine traverses the structure of the radio locking memory where needed and subsequently unlocking it when it identifies that it is no longer required by a component. Internally, the memory manager maintains a pool of memory, which is reused according to the current lock/release status of the memory it maintains. Using this technique only the minimum amount of memory required by a radio is used.

The basic concept of this approach is illustrated in Figure 6.26. In this simple example only two memory blocks are required to allow a signal to be passed through seven components as blocks get reused. The result of using this technique for a practical scenario is shown in Figure 6.27. Here the memory required is analysed for the signal generator, FIR filter example of the previous section (Figure 6.23). The blue line shows the total amount of memory the engine requests from the memory manager and the red line shows the amount of memory that is actually used. In simple scenarios like this the engine can reduce the amount of memory required to a constant amount for any number of Radio Components.

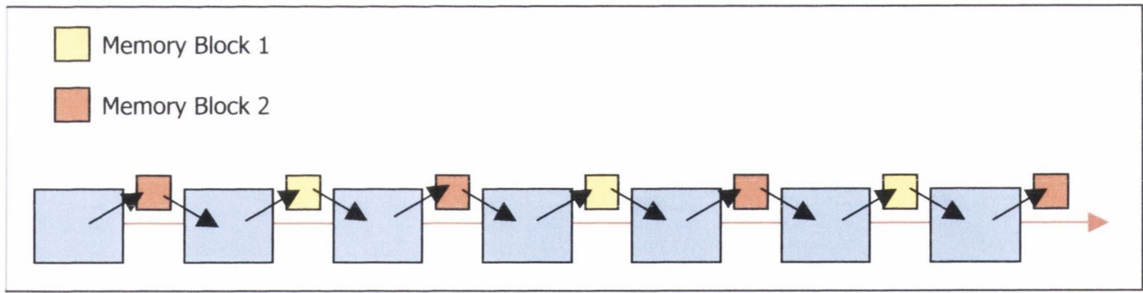


Figure 6.26 – Memory Allocation Technique

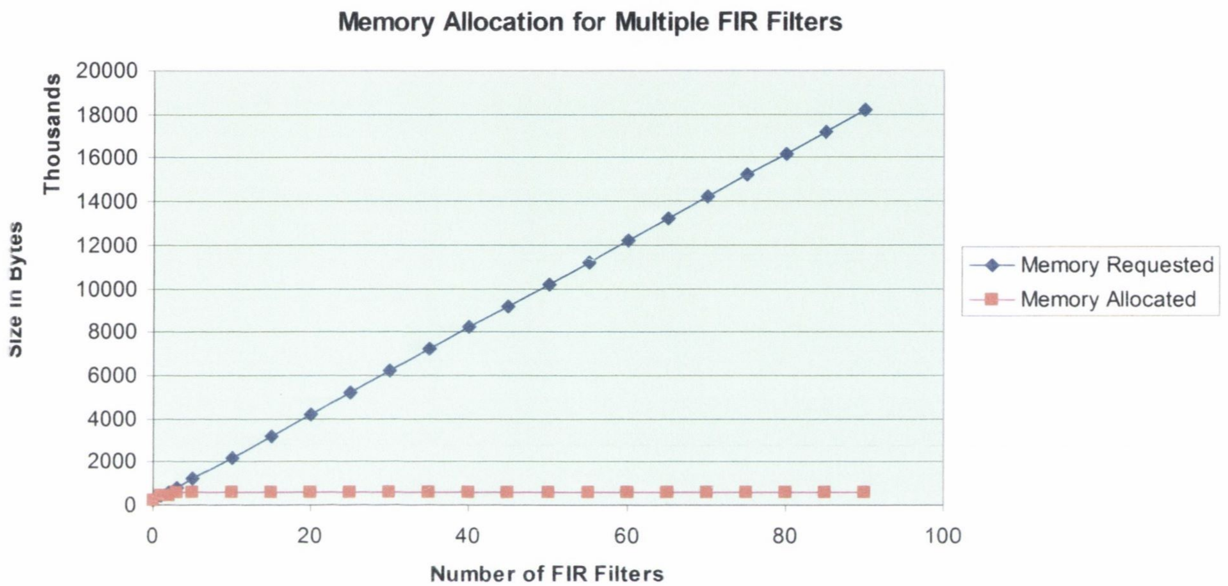


Figure 6.27 – Memory Allocation for Multiple FIR Filters

The reason the previous example can allocate so little memory is because the structure is simple. There is only one signal in the path, block sizes are fixed and thus maximum reuse can take place. However, in a practical software radio system, block sizes vary due to differing sampling rates and data types used in the system. For example, a transmitter will up convert a signal resulting in much more memory being required at the end of the signal processing chain than the beginning. In this scenario although the memory manager may contain memory blocks to service the needs of the engine, these blocks may not be big enough and thus additional memory has to be allocated.

To analyse the effectiveness of the memory manager in the face of growing and shrinking block sizes, two test scenarios were developed. In the first scenario, a high data rate signal is consistently down sampled by half using multiple '`DownSampler`' components (see Figure 6.28). This effectively reduces the sampling rate and the block size by half with each extra component.

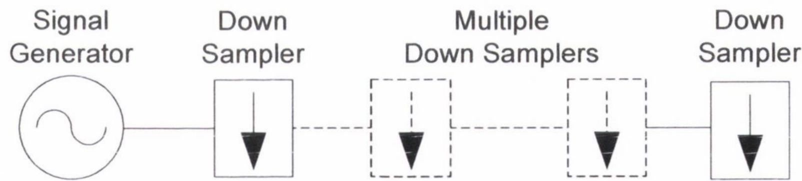


Figure 6.28 – Memory Test for Down Sampler Scenario

The results of this test are graphed in Figure 6.29. The test involved analysing the memory requirements for down sampling a 100MSPS (Million Samplers Per Second) signal split into block sizes of 23592960 samples (90MB). This graph shows that the memory required for a system involving a constantly decreasing block size can becomes constant, i.e. after a few stages of down sampling enough memory has been allocated overall to service the needs of the full system. The reason this happens is that larger memory blocks are allocated first (for higher sample rates) and these can be reused to service the needs of components requiring smaller memory blocks.

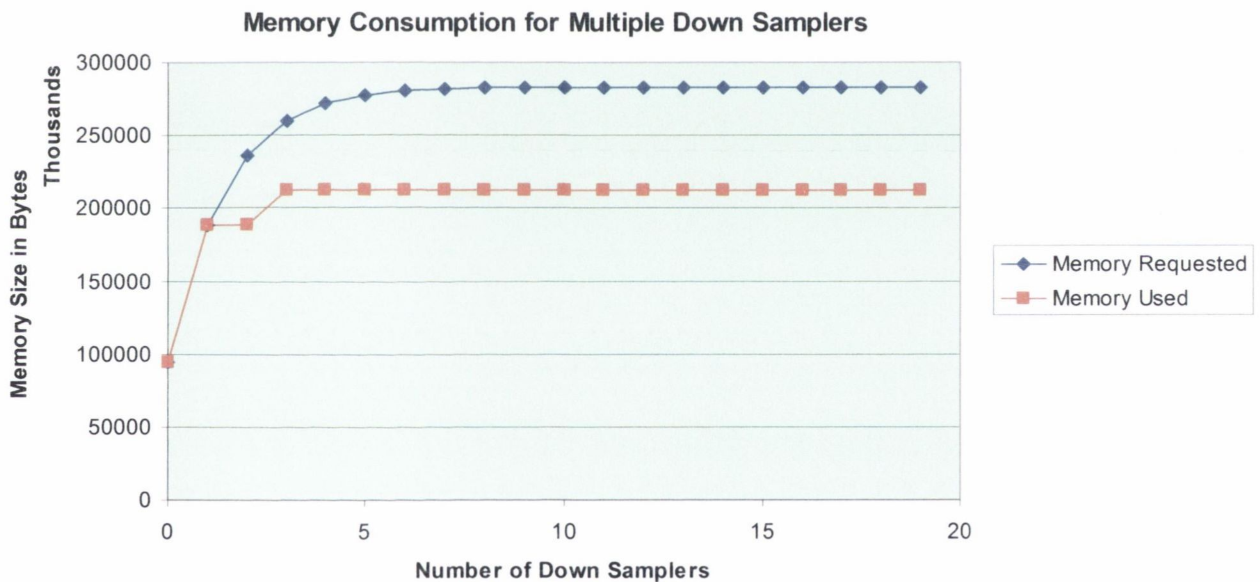


Figure 6.29 – Memory Consumption for Multiple Down Samplers

Figure 6.30 shows the second scenario in which a signal is up sampled by two, effectively doubling the sample rate and hence memory requirements for each additional component. The results of this test are shown in Figure 6.31. This graph demonstrates that although the memory manager can reduce the amount of memory used in the up sampler scenario, memory usage cannot converge to a set memory amount as block sizes constantly increase.

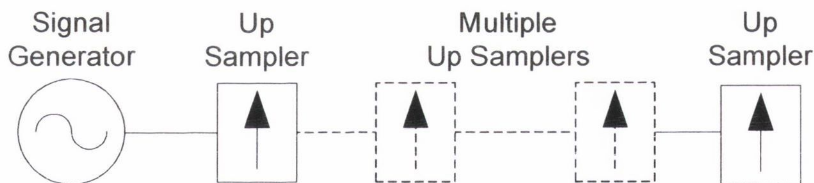


Figure 6.30 – Memory Test for Up Sampler Scenario

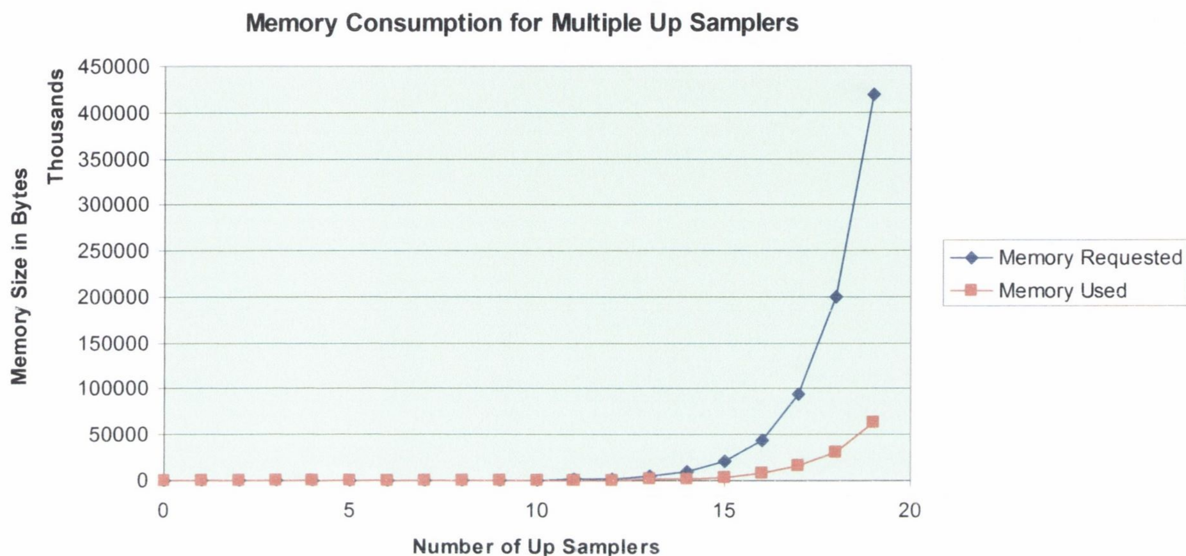


Figure 6.31 – Memory Consumption for Multiple Up Samplers

The three memory tests presented here have demonstrated that it is possible to constantly reduce the amount of memory required by the system in a generic way. By observing memory use the engine can reduce the overall memory requirement of a radio by reusing memory among components. The down and up sampler scenarios have tested these limits to extremes. In reality less sample rate changes will probably take place meaning more memory reuse can occur.

These tests also highlight the dominance of large memory allocations in memory consumption. For example, the down sampler test showed memory usage convergence at 202.5MB (53.08 million samples). The reason this memory requirement is so high is due to the first few stages of down sampling which require the representation of a 100MSPS signal with 90MB block sizes whereas subsequent stages require much less memory. Thus, analysing the sampling rate and memory requirements of a radio application can be an important tool in identifying the requirements of the overall system.

6.7 Summary

This chapter had demonstrated the implementation of IRIS providing a practical insight into the development of reconfigurable radio systems. Through a series of examples this chapter has demonstrated how the IRIS system is flexible enough to handle the requirements of almost any radio system and to allow the specification and construction of these systems in a generic way. The results of analysis show that IRIS can be implemented in a scalable and memory efficient way.

7 Case Studies

7.1 Introduction

The two previous chapters discussed the IRIS system and how it can be used to develop common radio systems such as receivers, transmitters and transceivers. However, although IRIS facilitates the development of these devices, its reconfigurable nature enables much more. To demonstrate the flexibility of IRIS, this chapter shows how it can be applied to some newer emerging wireless technologies. This is shown by way of three case studies.

'Over the Air Reconfiguration' is presented first. This case study shows how the IRIS system, in particular its support for reconfigurability, facilitates the development of radio systems that can be reconfigured remotely by downloading new software. The next study discusses Wireless Networking and how the IRIS system can work as the physical layer in a communications stack, bringing enhanced reconfigurability to ad hoc wireless communications. The final study discusses the growing need for spectrum management technology and describes how IRIS can be used to build systems to address this need.

7.2 Over the Air Reconfiguration

7.2.1 Overview of Over the Air Reconfiguration

Chapter 2 (Section 2.5) touched on the topic of software download or Over-the-Air Reconfiguration (OTAR). OTAR allows a mobile wireless terminal (such as a mobile phone) to be reconfigured by downloading new software to the terminal over the wireless connection [Noblet98, Cummings99c, Bucknell2002]. This software can be used to reconfigure the wireless terminal thus changing the capabilities of the radio device.

For the mobile communications industry, OTAR has moved from being an ancillary aspect of software radio to become a primary motivating factor driving the adoption of software radio. In this field manufacturers have long recognised the advantages of being able to use software to perform upgrades, fix bugs or add new features to a mobile phone. Software download occurred first with the ability to upgrade the firmware (essentially the operating system) of a mobile phone. This gave manufacturers the ability to correct errors or add new features. Subsequently with more data connectivity appearing in handsets, the download paradigm has extended into downloading over-

the-air and the downloading of Java applications and games to phones has become commonplace. Many believe the next stage in this evolution is that the radio system of the mobile phone will be upgradeable over-the-air, facilitated by software radio.

There are wider implications of this technology that could change the economics and usage models for wireless devices. Just as software download has become ubiquitous on the Internet, this type of download scenario could in time begin to emerge in wireless devices. Instead of radio devices being sold with fixed operating parameters, software download could allow devices to be sold as general-purpose units that are later configured for a particular application, much in the same way computers are sold today. For example, when someone owning a general-purpose communications device enters a new city, their mobile wireless terminal could automatically download the local popular communications standard. This would consist of the software required to configure the radio system, possibly including information such as local frequency plans, modulation schemes, etc. This type of reconfiguration could allow the general-purpose radio terminal to cut across the standards boundaries of today offering true ubiquitous connectivity.

Technically, OTAR allows two capabilities. Firstly, OTAR allows a terminal to download new software that changes the capabilities of the device, possibly introducing new features or changing the standards by which the radio device communicates. For example, a mobile phone could download a wireless standard such as GSM and reconfigure itself to work with this new standard. Secondly, OTAR facilitates the upgrading of software. Using OTAR, a mobile device can download new software that possibly fixes bugs, adds new features or improves the performance of the device. For example, a terminal could download a more CPU efficient filtering algorithm thus improving its battery performance. The technical capabilities and potential advantages of OTAR are evident, but there is no consistent methodology for enabling OTAR in radio systems. The next section shows how the IRIS system can address this need.

7.2.2 Applying IRIS to OTAR

To enable OTAR, two main issues must be addressed; how to download software and how to reconfigure a device once the download has completed. In terms of the work in this thesis, the latter is facilitated by reconfigurability. The built-in reconfigurability of the IRIS system allows a radio to be reconfigured once a software download has been completed. To actually download software, the main issue that has to be addressed is what actually gets downloaded, as different degrees of reconfiguration will require different types of downloads. By looking at the degrees of reconfigurability discussed in this thesis, we can identify what needs to be downloaded via OTAR to enable reconfiguration.

Parametric reconfiguration allows the parameters of signal processing or any radio functionality to be altered. In terms of software download, parametric reconfiguration would involve the transfer of a new parameter or set of parameters that change the configuration of components in the radio system. This could be as simple as transferring the new operating frequency of the radio system to more complex configurations that redefine the signals in components by changing sample rates, data types, etc. Downloaded parameters could be transferred via XML thus providing a standardised method for OTAR at this level.

Structural reconfiguration allows the actual structure of components in the radio system to be changed, altered or replaced. Structural reconfiguration could be very important for software download, especially as it facilitates the replacement of components. A new software component could be downloaded and used to upgrade an existing component (the issues surrounding software upgrading for software radio were discussed in Section 4.3.8). For example, a new speech encoder component could be downloaded possibly offering improved speech quality. The software component and associated structural information can therefore be another downloadable item.

Application reconfiguration allows the whole radio to be changed, consequently altering the complete function of a radio system. Application reconfiguration is the most comprehensive form of software download, as by downloading a new configuration, a radio should reconfigure itself to work as a completely different device. This could involve the download of new components or be achieved by reusing existing ones. For application reconfiguration to occur with OTAR, components, XML configurations and control logic would have to be downloaded.

These types of reconfiguration are summarised in the table shown here in Figure 7.1:

Reconfiguration Type	Items Required for Download
Parametric	Parameters *Examples: Frequency settings, sample rates, filter characteristics, modulation settings.*
Structural	Components and Parameters *Examples: GMSK modulator, FIR Filter, Speech Encoder.*
Application	Components, Parameters and Control Logic *Examples: GSM transceiver, QPSK transceiver, GPS receiver, location transmitter.*

Figure 7.1 – Items for Download with OTAR

While these types of reconfiguration directly dictate the type of downloads that should take place, a method is still required to perform the actual download. A set protocol or procedure is required to

initiate and perform the actual reconfiguration. There are various forms such a protocol can take but this will largely depend on the final application, and who has control over the radio terminal. For mobile telephony the network operator may want to have the ability to remotely reconfigure the mobile phones using its network. This would allow the operator to ensure the stability and reliability of the network by controlling the communications standards used. In this scenario the operator may 'push' new software to mobile phones. This would have an impact on roaming allowing people to use their phone in any country. Network operators could push a download to a users phone that provides the software required to operate in that network.

On the other hand in some situations it may be the owner of the terminal who initiates a reconfiguration. For example, in an emergency, police officers may want to reconfigure their 2-way radios to allow communication with other emergency services. In this scenario the police officer's radio device would initiate the reconfiguration by 'pulling' new software. Whatever form OTAR takes, the software radio system must provide sufficient functionality to facilitate these processes.

In terms of the IRIS system there are two ways in which software download is catered for, depending on the levels of reconfiguration required. Figure 7.2 shows how software download could be performed using the control logic of a reconfigurable radio. In this scenario the control logic would use a protocol for sending and receiving data through the radio system. This protocol could be used to initiate software download between two radio systems. As described in Figure 7.1, the actual information received could be XML data for parametric reconfiguration or the actual binary code of a component with associated parameters for structural reconfiguration.

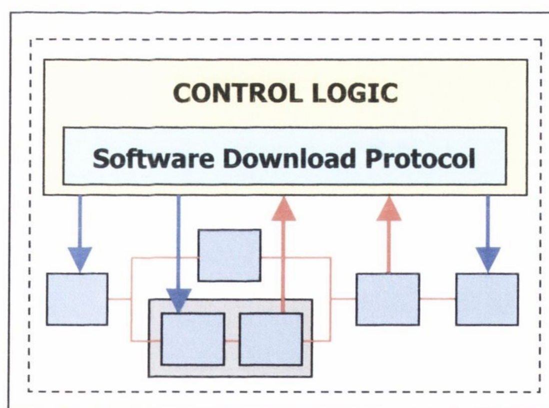


Figure 7.2 – Software Download with Control Logic

While control logic facilitates the reconfiguration of parameters and structure, application reconfiguration is the full upgrade of a whole radio system possibly including the control logic itself. In this scenario the software download mechanism must exist outside the radio system itself

that has to be replaced. This is facilitated using the IRIS API (see Section 6.4.1). Figure 7.3 shows a diagram of how this would work.

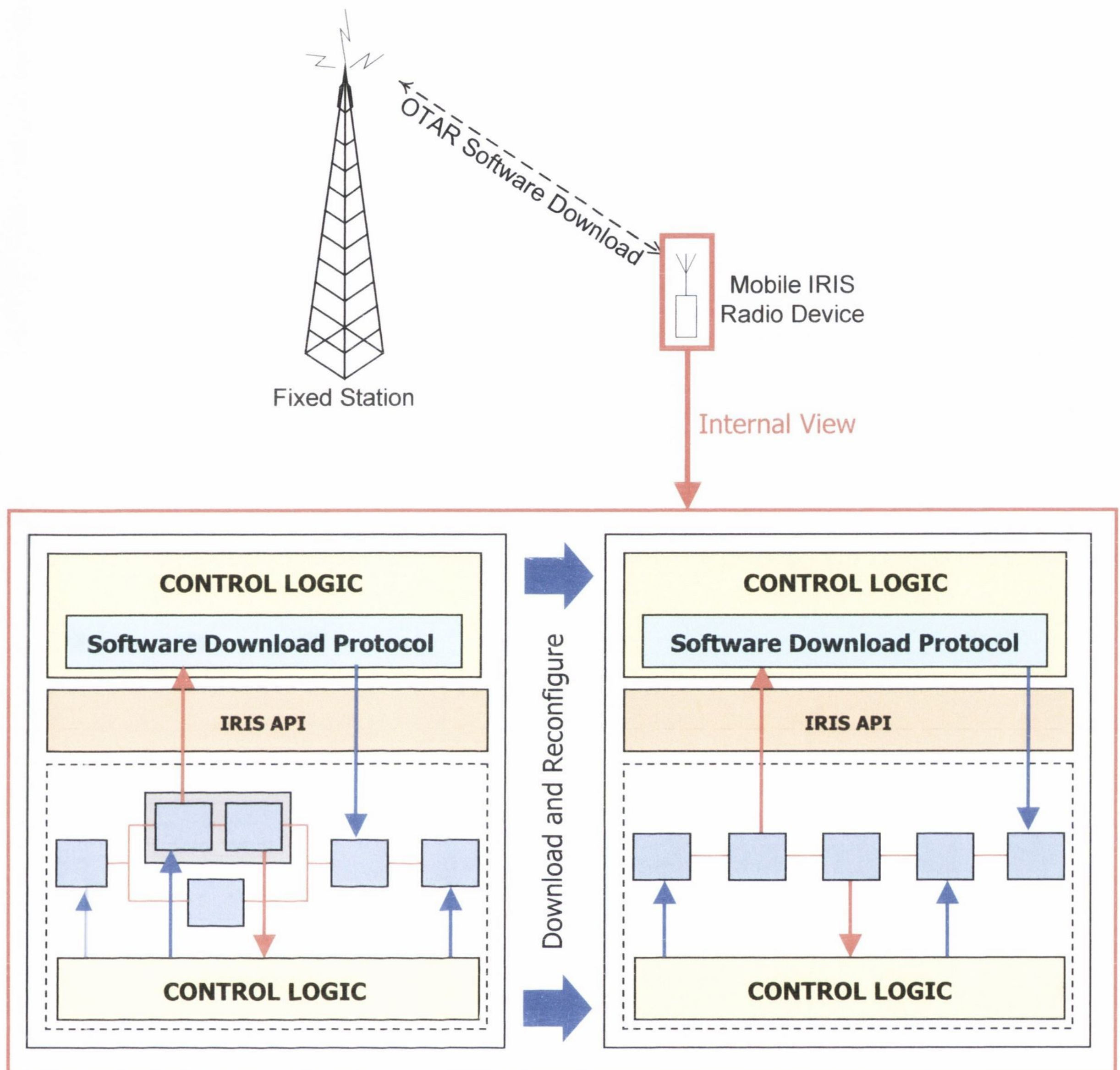


Figure 7.3 – Software Download Using the IRIS API

In this scenario extra control logic is written at the level of the IRIS API (as discussed in Section 6.4.1). This control logic can implement the download protocol for receiving the software, but it is also separated from the particular radio itself in such a way that it can undo the existing radio system and reconfigure it to the new downloaded specification.

There are several forms the actual protocol used for software download could take. Reliable communication is required to reconfigure the radio system itself. Reconfiguration at this low level is often beneath the logic that implements error correction and reliability protocols. For this reason

protocols implemented for software download may require their own protocol stacks to ensure reliable transfer of data.

There are other problems that have been addressed in the literature which deserve consideration in the context of the IRIS system. Security is a common concern as the adverse reconfiguration of a terminal could have dire consequences in a radio network as discussed in Section 4.3.5. Michael [Michael2002] and Bucknell [Bucknell2000] have both suggested schemes for secure download for software radio systems. The consensus is that security incorporating suitable encryption needs to be built into the download protocol itself. In the IRIS system, either a control logic or IRIS API level protocol can be written to enable this, the specifics of which will depend on the final application.

The last problem to discuss is software versioning. As mentioned in Chapter 4, Section 4.3.8, conflicts in the versioning of software can lead to a malfunctioning system or even result in more subtle communications errors due to an unexpected configuration. This problem can be further complicated by the fact that a radio system may not be able to re-establish communications to fix the problem. For this reason, when performing software download, the radio system must be able to validate a particular radio configuration. IRIS supports two methods for validating configurations. Firstly, the IRIS system automatically verifies the structure of a radio design by checking its configuration. This checks the validity of the XML itself and also the semantics of the radio design. As discussed in Section Chapter 6, 6.2.5, invalid block sizes or sample rates cause an error to be raised. This error checking ensures that incompatible component combinations and parameter values are avoided. Secondly, IRIS inherently supports versioning of components. Each component has versioning information associated with it. Using this information the control logic of a radio system can predetermine whether particular components will work together.

7.2.3 Conclusions

This case study has discussed OTAR and how it is possible using the IRIS system. This review has shown that the degrees of reconfiguration supported by the IRIS system can be directly mapped onto the software elements that are downloaded by an OTAR-capable reconfigurable radio. The inherent reconfigurability of the IRIS framework ensures a suitable platform for developing OTAR radio devices.

7.3 Wireless Networking

7.3.1 Overview of Wireless Networking using DAWN

In related research to this work, the NTRG (Networks and Telecommunications Research Group) in Trinity College has developed a test bed for developing wireless networking applications called DAWN (Dublin Ad hoc Wireless Network) [O'Mahony2002]. This test bed is a software-based environment for dynamically creating network communication stacks and it allows for

experimentation with a wide range of networking technologies. DAWN has been a host to a variety of research related to wireless networking [O'Mahony2002b, Doyle2002b, Doyle2002c, Doyle2001, Forde2000].

DAWN is a useful test bed for developing wireless applications. Figure 7.4 illustrates a typical DAWN network topology. In such a topology many devices can participate in the network including fixed computers with wireless connections and mobile devices for example PDAs (Personal Digital Assistants) and laptops. Each device hosts a DAWN stack that provides all the communications infrastructure required to enable communications with multiple devices.

Internally, the core of the DAWN test bed is a 'generic layer' interface that allows the dynamic assembly of a network communication stack. Individual layers are written to address the various functional requirements of networking, for example security, routing and medium access control (MAC). Information is passed through the stack using messages, each containing a data payload and additional descriptive information. This descriptive information can be used to transfer information between layers in a stack.

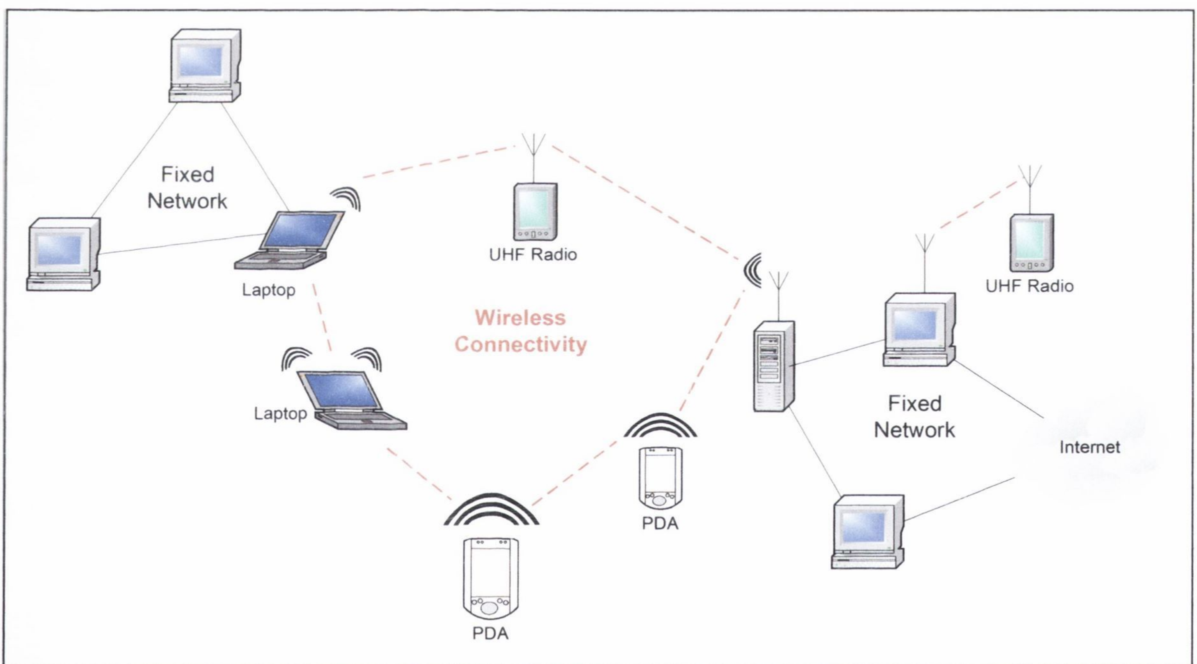


Figure 7.4 – Typical DAWN Topology

A typical DAWN stack is shown in Figure 7.5. At the top of the stack sit various applications which have been developed for use in a DAWN network including a messaging application, a real-time phone and standard testing facilities such as pinging. Below this level sit the layers which make up the communications stack. A variety of layers exist including layers dealing with

networking functions such as routing, security, reliability and medium access control. By combining layers in different ways, many types of stacks can be created.

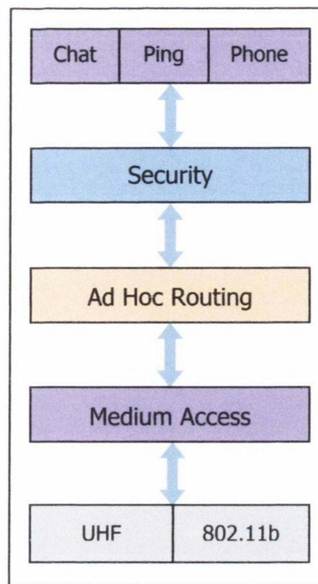


Figure 7.5 – A Typical DAWN Stack

In addition to protocol layers and research-based networking layers, various physical layers have been developed allowing communication over UHF transceivers and Wireless LAN (802.11b) [Gast2002]. The UHF module allows for low bit rate data transfers at around 30kbps and are used in experimental work for large populations of ad hoc networking nodes. Using Wireless LAN equipment speeds of up to 11Mbps can be achieved. The ultimate goal is that the physical layer moves from being a hardware dependent device (as it is with UHF radio and 802.11b) to an ideal software radio approach, in that any radio scheme can be implemented as part of the DAWN stack.

7.3.2 Applying IRIS to Wireless Networking

To investigate the role of software radio in the networking environment, the IRIS system has been used in conjunction with the DAWN stack. Using the IRIS API, a layer has been written that allows the IRIS system to act as a physical layer in the DAWN communication stack (see Figure 7.6). This layer acts as a bridge between the layer construct and the IRIS API. By interacting with the IRIS API and control logic, the generic physical layer can send data to an IRIS radio for transmission and likewise receive data. Using this approach, new physical layers can be created by using a different radio configuration with this generic layer.

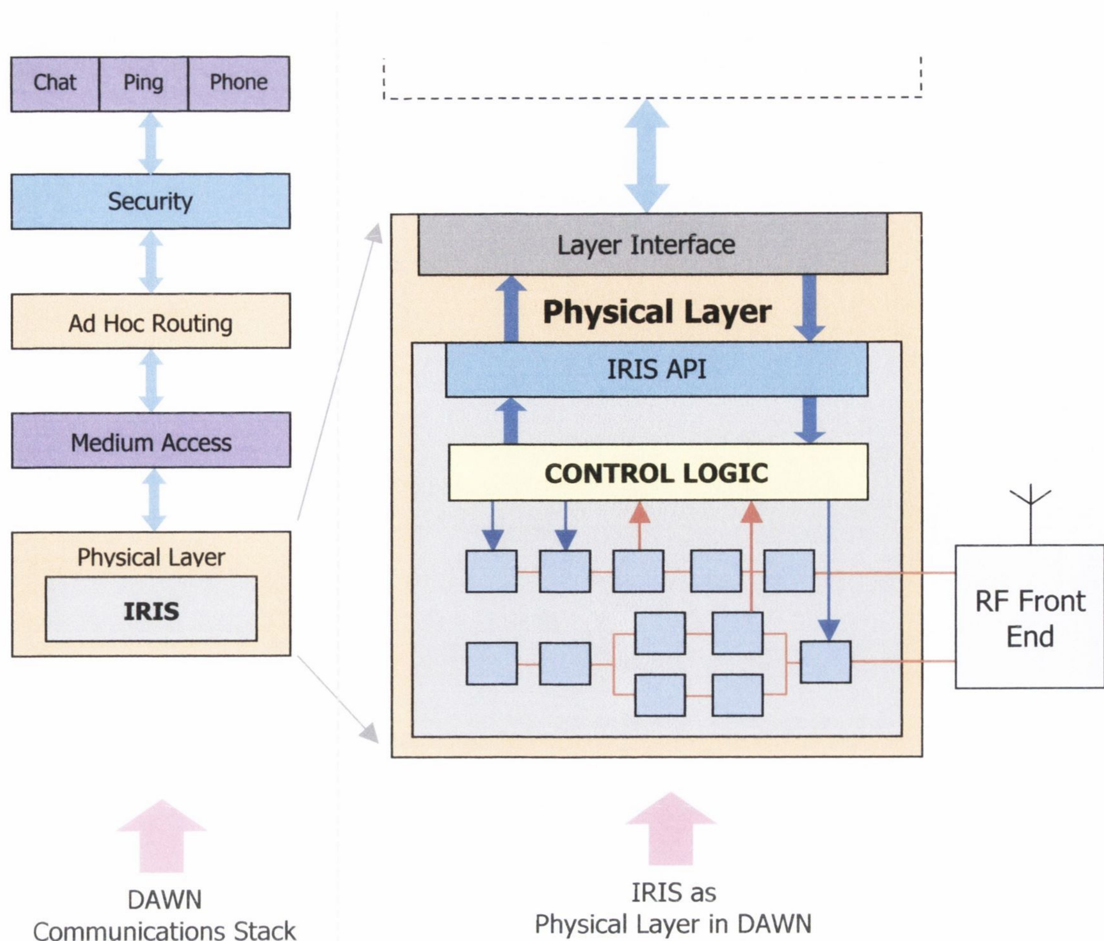


Figure 7.6 – IRIS Incorporated into DAWN

Even though current technology limits the ability for IRIS to replace CPU intensive schemes such as 802.11b, the integration of IRIS into DAWN has provided valuable insight into how software radio and in particular reconfigurability will play a role in the future of wireless networks. Incorporating IRIS into the stack has provided insight into how software radio can enhance the capabilities of a wireless network through reconfiguration. The main benefit of this has been to see how other layers in the communications stack such as medium access control and ad hoc routing can benefit from the reconfigurable nature of IRIS. This can manifest itself in various ways.

In most communication stacks the physical layer or communications medium in general is not subject to change. The physical layer is usually rigidly fixed in function. For example, the only parameters that are user changeable in the 802.11b standard are concerned with protocol procedures. It is not possible to alter the way in which the device communicates its radio signals. A better physical layer would allow any aspect of physical layer communications to be changed dynamically, facilitating new possibilities and types of communication. IRIS facilitates this in the DAWN stack. Through reconfiguration IRIS can allow upper layers in the stack to change aspects of the physical layer. For example, the MAC layer could alter the physical layer to better enable collision detection or the cryptography layer could interact with IRIS to instigate lower level

cryptography thus securing communication throughout the stack. This approach provides the flexibility needed to address the needs of future applications.

In addition to upper layers controlling the physical layer, information and control can occur in the opposite direction. Current physical layer standards provide only limited information about signals being received. In 802.11b for example, some vendors allow the incoming signal strength to be measured, but in practice this information is not used in the actual communications procedure. In contrast, the IRIS physical layer can provide additional information to upper layers about communications. Any information or occurrence in the physical layer can be relayed to upper layers, possibly allowing them to make better decisions therefore enabling more effective communications overall. For example, the IRIS system can provide a routing layer with details about the signal strength of received signals thus providing information that could be used to make an informed decision about which route to take. This is an emerging area of research in ad hoc networking and the need for such devices is discussed in the context of future work in Chapter 8. As another example, the IRIS physical layer could calculate the conditions of multi-path fading experienced at the receiver. Using this information an upper layer could make an informed decision about the environment it is operating in, possibly changing the protocols to suit that context. Again, this type of information and control facilitates new types of intelligent applications.

Intelligence however does not have to be integrated throughout the communication stack, in particular the physical layer should be flexible in that it can be integrated with a number of existing standards without requiring alteration. In this scenario it would be advantageous that the physical layer works intelligently to allow better communications. One possibility is that the IRIS physical layer could monitor the traffic sent and received by upper layers. Using this information it would make intelligent decisions about the type of communications to use. For instance, when experiencing a low volume of traffic the IRIS system could move to a more power efficient modulation scheme to save battery power. As another example (and this time considering the existence of an ideal front-end) the physical layer in detecting an increase in throughput could dynamically create new transceivers in software thus increasing the capacity of the radio link. While some of these examples are not possible at present, improvements in technology can only bring us nearer to this type of capability.

Overall, wireless networking has much to gain from reconfigurable radio. Instead of the physical layer being a statically configured device serving the lowest common denominator, it can become a dynamic intelligent device serving the diverse needs of a variety of applications.

7.3.3 Conclusions

This case study has shown how IRIS can be used as the physical layer in a communications stack without modification. It demonstrates how the reconfigurable framework of IRIS can enable new and improved capabilities in wireless networking. While some of these techniques may not be applicable today, new emerging technologies such as ad hoc networking are built with more flexibility in mind. These networks are highly reconfigurable as they have no fixed infrastructure and each node in the network is autonomous, acting both as host and router [Broch98]. In this type of environment the reconfigurability discussed can facilitate better communications enabling new types of applications.

7.4 Spectrum Management

7.4.1 Overview of Spectrum Management

The growth of wireless communications over the past two decades has generated an increasing demand for spectrum allocation. In response to these demands the communications regulators of many governments around the world have been taking a fresh look at how spectrum is allocated and managed with a view to improving spectrum capacity. (Existing methods for spectrum management are discussed in [Withers91]). One organisation making considerable progress in this field is the U.S. Federal Communications Commission (FCC). In their Spectrum Policy Task Force report [FCC2002] they outline bold new strategies for spectrum reform by introducing fundamental changes into spectrum management.

The main point emerging from the FCC report is that the current methods used to regulate spectrum are outdated and do not reflect current technological capabilities. In essence, the technological needs of today were unforeseen when these regulations were put in place. One example is the way mobile communications has changed the use of spectrum. Previously, use of the spectrum was largely via a broadcasting model whereby a small number of transmitters served a large number of receivers. This model was used throughout television and radio broadcasting, and for information devices such as pagers. Whereas the broadcast model required only a small number of frequencies to serve sometimes millions of users, the mobile phone requires both a downlink and an uplink channel for each individual user of the system. Consequently spectrum usage increases for each additional user of the system.

Current allocation policy results in even more demand due to the dimensions used for allocation. Allocation is performed mostly by frequency, yet other dimensions such as space, time and power also exist and offer great potential to increase capacity. Increased allocation by space would allow organisations in different regions to use the same frequencies. In time the granularity of space

could be reduced and in the extreme possibly allowing users to use the same frequencies in much smaller vicinities, for example in different floors of a building. Allocation by time would allow users access to underutilised spectrum, effectively filling the unused gaps of available spectrum time. Transmit power, although partly regulated today, could be made a more effective means of allocation and would go hand in hand with regulation by space. Overall, there are many new ways to allocate spectrum but until now the technology has not existed to allow its implementation.

The FCC and other bodies have recognised that software radio is an enabling technology in achieving more effective spectrum regulation. Software radio can offer the flexibility required to deliver devices that are dynamic in their use of frequency, power and time, thus reconfigurability has a significant role to play. The next section describes how the IRIS system can be used to build two types of systems that enable dynamic spectrum use.

7.4.2 Applying IRIS to Spectrum Management

This section shows how the IRIS system can be used to address two issues in dynamic spectrum use, namely interference temperature and spectrum monitoring.

One of the fundamental reasons spectrum is regulated is to ensure interference free communication. Therefore, in exploring new ways of allocating spectrum it has been important to address how interference will be managed in this new dynamic environment. The FCC have proposed the use of an 'Interference Temperature', a metric that measures the RF power available at the receiving antenna [FCC2003]. The idea is that spectrum aware devices can dynamically calculate the current temperature to determine whether it is permissible to communicate or whether the device should try a new frequency. The FCC has proposed that the metric be defined as 'the RF power generated by undesired emitters plus noise sources that are present in a receiver system per unit of bandwidth' . Regulators can assign different threshold levels for each band, effectively allowing them to control the noise floor. This requires a device that can measure the interference temperature and react accordingly.

An interference temperature device requires reconfigurability in that the device must be able to adjust its operation in relation to its environment. Using IRIS this type of system can be developed by employing control logic and parametric reconfiguration. Figure 7.7 shows a diagram of how the FSK transceiver example from Chapter 6 could be modified to react to Interference Temperature.

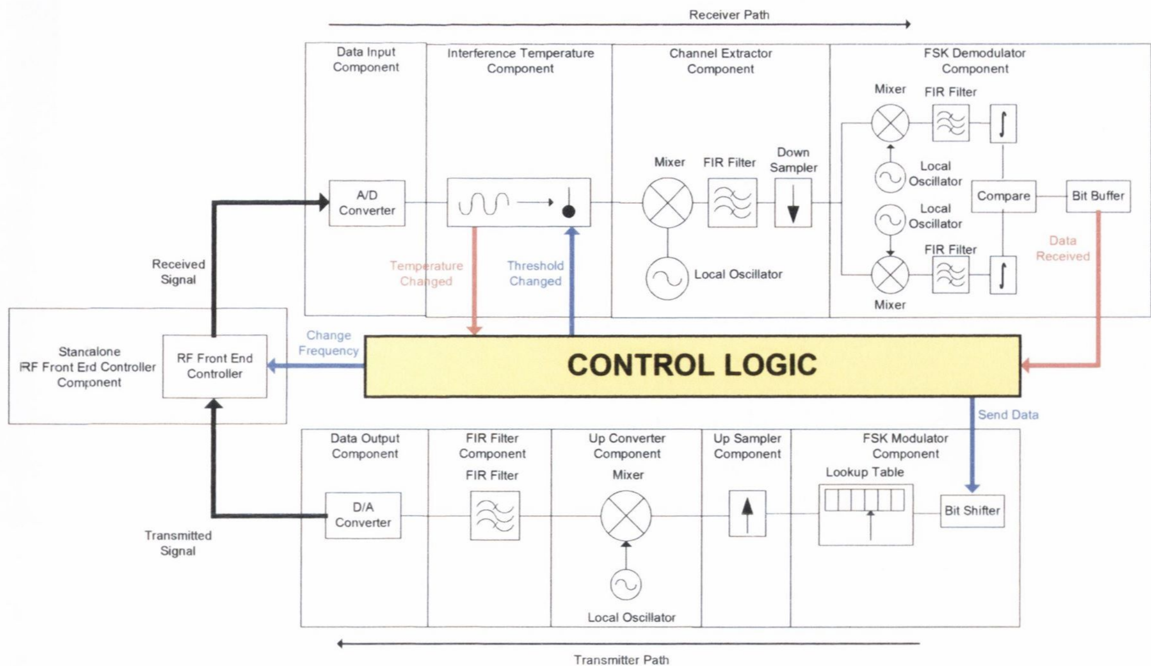


Figure 7.7 – FSK Transceiver with Interference Temperature Detector

In this design the radio monitors the temperature via an Interference Temperature component. This calculates the noise per unit bandwidth on the currently tuned frequency. This can be done in several ways. If a periodic transmission of a known sequence is taking place the component can compare the known transmission with the received signal. As the noise increases over time with the introduction of new transmitters into the band it can track the noise difference. Without a fixed periodic transmission to correlate against, the component could measure the overall power of the noise floor over time.

Using control logic the consequence of this temperature can be computed possibly changing the operating frequency of the radio system by controlling the RF front-end. The temperature could also change other aspects of the radio system perhaps changing the modulation scheme or data rate of the radio to reduce the interference it causes.

Because the IRIS system exists primarily in software, enabling interference temperature is straightforward, only requiring the inclusion of a new component and some additional control logic. The fact it is written in GPP-based software also means that the component can be reused in many other designs requiring this functionality. This reiterates a point from Chapter 3 about software components; they allow encapsulation and reuse of software and are a suitable method of deployment. For example, instead of regulating the requirements of individual radio devices regulatory bodies such as the FCC could regulate and approve particular software components for

use in Interference Temperature calculations. However, the actual method for doing this requires further research and is discussed in Section 8.3.

Besides the actual terminals that communicate, there are other types of devices that will be required in a dynamic spectrum environment, one such example being a device for monitoring spectrum usage. Many regulators already monitor spectrum on a regular basis but in a more demanding and dynamic environment communications will have to be more closely monitored to ensure that the policies in place are effective. This will require monitoring stations that can detect the Interference Temperature but possibly also analysing individual transmissions. This could be of use in enforcing regulations by tracking misuse of spectrum or by producing statistics and feedback information regarding the types of transmissions occurring in the medium.

The IRIS system can be used to develop such a monitoring device. Figure 7.8 shows the design of a spectrum monitoring system. This system sweeps any band of interest and continually analyses the signals received for any communications occurring in that band. This can be done by basic pattern matching in looking for strong signals and any signal of interest can be down converted to baseband where it enters a signal buffer. The buffer makes use of the large amounts of RAM available with a GPP design to store the most recently received radio signals. The control logic on receiving a particular signal of interest can retrieve a previous occurring signal and possibly record it to disk for later analysis. A signal database is also included and this could be used to store statistical information resulting from the analysis.

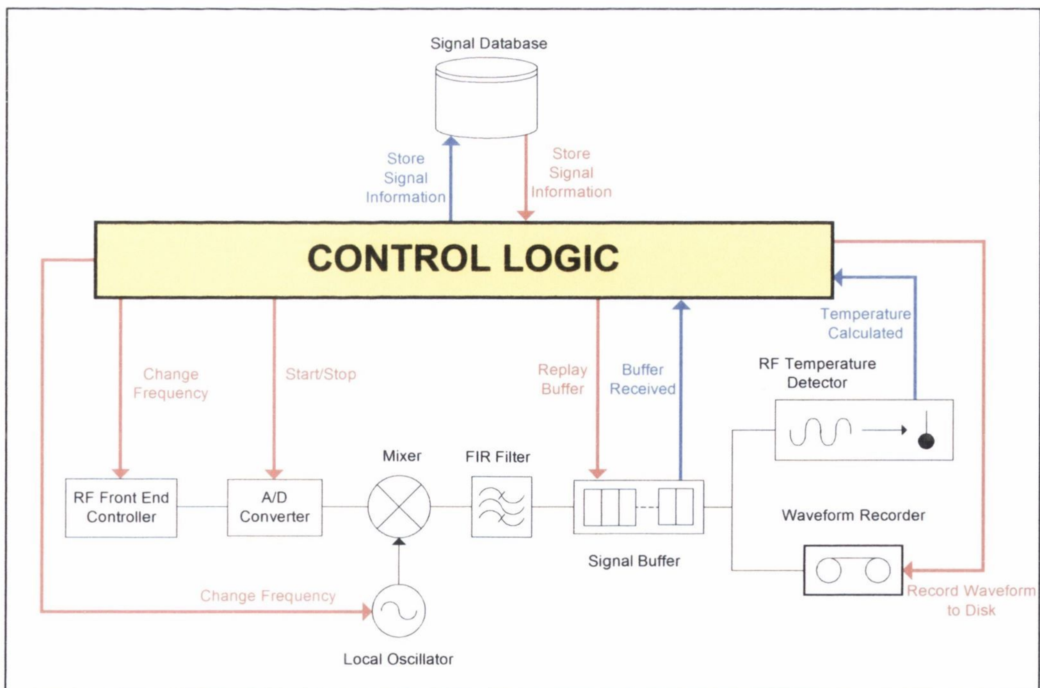


Figure 7.8 – Spectrum Monitoring System

7.4.3 Conclusions

This section has discussed the emerging area of spectrum management and how the IRIS system can address the technological needs of devices operating in this environment. The interference temperature device demonstrates how temperature calculations can be encapsulated in a Radio Component and subsequently used to reconfigure other parts of the radio system. This type of system can be built easily using the IRIS component framework. While this discussion has only provided a brief overview of the topic, spectrum management is an important and growing area of research. For this reason Chapter 8, Section 8.3.4 suggests future directions for research in this field.

7.5 Summary

These case studies have demonstrated that the approach to reconfigurability in this thesis and the IRIS system in general are applicable to emerging techniques and technologies in communications. OTAR showed how IRIS and its support for reconfigurability deliver the functionality required for software download, an important technique in the software radio space. Wireless networking demonstrated how IRIS can act as the physical layer in a communications stack and also offers new capabilities to wireless networking devices by introducing elements of physical layer reconfigurability to the communications stack. Spectrum Management showed how IRIS can assist the development of a new emerging technology, enabling a fundamental change in the way communications take place. Overall these case studies demonstrate that the IRIS system and the concepts of reconfigurability are important contributions as they show that a software-oriented component-based approach to software radio design yields highly reconfigurable radio devices.

8 Conclusions

8.1 Introduction

This chapter draws conclusions from this thesis. Section 8.2 revisits the specific contributions identified in the introductory chapter and shows how they were achieved. Section 8.3 highlights future work that can follow from this body of research and Section 8.4 concludes.

8.2 Summary of Contributions

The purpose of this work has been to show that a software-oriented component-based approach to software radio design yields highly reconfigurable radio devices. The following discussion reviews the contributions that have been presented to substantiate this claim.

A comprehensive overview of software radio technology

The thesis as a whole provided insight into software radio technology, the main discussion of which was presented in Chapter 2. That chapter presented a comprehensive review of the history and evolution of software radio, focusing both on the technology involved and the wider consequences of software radio itself. This chapter gave a unique perspective on the subject, showing how it is a wide and diverse field, bringing together various technologies to introducing a new paradigm in radio system design. The viewpoint taken in explaining software technology was unique because unlike most other research it did not concentrate on one particular application such as mobile telephony, wireless LAN or digital television. Instead this discussion was presented from a general perspective and thus provides a unique picture of the software radio space as a whole. This discussion also provided the background knowledge necessary to approach the problem of reconfigurability in radio systems. Thus, the software-oriented solution presented is generic and can be applied to many different applications including mobile telephony, etc.

Categories for assessing reconfigurability in radio systems

As the aim of the thesis was to develop highly reconfigurable devices, it was thus necessary to have a metric in assessing the level of reconfigurability. This need has resulted in the most important contribution of this work, which is the analysis, definition and categorisation of reconfigurability. This is a unique contribution as no prior work in this field has looked at the concept of

reconfigurability in this way. The three categories, application, structural and parametric reconfiguration, can now be used by others in assessing the reconfigurability of radio systems. These categories have also served as basic requirements for the IRIS system.

Analysis of software design for radio systems

Chapter 4 analysed eight topics in software development and how they apply in the field of software radio. These topics: - reuse, abstractions, adaptability and flexibility, complexity, security, portability, real-time behaviour and finally upgrading and versioning - have been the result of extensive research. That research demonstrates the need for component-based software in delivering reconfigurable radio systems and thus is an important contribution.

Design, implementation and analysis of a reconfigurable radio system

The IRIS system demonstrates some fundamental properties of the reconfigurable radio concept. Firstly, it shows that it is possible to develop a reconfigurable radio system that exhibits application, structural and parametric reconfigurability. Secondly, it demonstrates that component-based software can be used to develop highly reconfigurable radio systems and that it is possible to encapsulate signal processing algorithms into such a component without compromising its level of reconfigurability. Finally, it provides useful information to others who may consider following on from this research in developing either reconfigurable radio systems or other signal processing systems on GPPs. For these three reasons this is an important contribution to the field of software radio.

Case studies that apply the reconfigurable radio approach

The case studies presented in Chapter 7 are an important part of this work as they show that the IRIS system and hence the software-oriented component-based approach of this thesis is applicable to real problems in radio system design. It shows that the operations required in applications such as OTAR (Section 7.2) or network integration as in the case of DAWN (Section 7.3) can be mapped directly to the categories of reconfigurability. For example OTAR relies on structural reconfiguration and DAWN relies on parametric reconfiguration. This shows that the concepts presented in this thesis are valid and that the approach taken is justified.

8.3 Future Work

At various points throughout this research, areas of interest have been highlighted that require further research. The following sections discuss these in turn.

8.3.1 Hardware

Despite its software-oriented approach, reconfigurable radio requires hardware capable of delivering the type of platform it requires. The following areas in hardware development for reconfigurable radio require more attention:

- *More dynamic RF front-ends*

  Current technology cannot provide the requirements of the ideal software radio as discussed in Section 2.4.1. In the evolution towards the ideal software radio more dynamic RF front-ends are required that deliver better performance, larger bandwidths and can operate on multiple frequencies. New techniques and further research are required to deliver this goal.

- *Low power hardware*

  Further improvements in low power hardware need to be made to allow the reconfigurable radio concept to be feasible for mobile devices. A reduction in power consumption would allow reconfigurable radio devices to be embedded into mobile terminals while still allowing them to perform the majority of signal processing in software. This requires further research in semi-conductor technology.

- *Signal processing devices that inherently support reconfigurability*

  Chapter 4 discussed reconfigurability and defined application, structural and parametric reconfiguration as metrics for assessing reconfigurability. An interesting area of research would be to develop a domain-specific piece of signal processing hardware that inherently supports application, structural and parametric reconfigurability. Another interesting area would be the tools required to develop software for this platform. An environment that supported component-based software development would be of particular interest.

8.3.2 Software

The discussion of software design for radio systems and the IRIS system itself have demonstrated that component-based software can yield highly reconfigurable radio devices. There are however some areas which could benefit from further research:

- *Real-Time*

  The topic of real-time behaviour of signal processing algorithms needs to be investigated in the context of reconfigurable radio. There are potentially two major research areas within this topic; firstly, real-time constraints can hinder reconfigurability, as some aspects of performing a reconfiguration may not be deterministic. Further research needs to be carried out to investigate how application, structural and parametric reconfiguration can be achieved in a hard real-time environment. Secondly, with the reconfigurable approach it may be possible to relax the needs of real-time behaviour altogether. A radio system built from the ground up to be less stringent on real-time requirements may be able to operate without real-time constraints.

Further research could investigate the development of algorithms and software for supporting this type of communication.

- *IRIS Components*

  Other researchers are already using the IRIS system as a basis for experimentation and development of new radio systems [Flood2003, Nolan2003d, Nolan2002c]. These researchers have developed components for experimenting with modulation schemes and algorithmic techniques. Research on IRIS can continue either by extending the IRIS system itself and investigating its use on other platforms or by the development of more components. The latter could prove to be of great interest, as by developing a host of reusable components, many new applications are possible.

- *IRIS Framework*

  Chapter 6, Section 6.6 discussed some basic scalability and memory experiments carried out on the IRIS system. Further work on the IRIS system could concentrate on developing the implementation further, using it to develop complete communications systems to a commercial grade and reporting on the scalability and performance required. Further scalability work could concentrate on analysing more complex sets of components for linear scalability.

- *Better integration with networking protocols*

  Section 7.3 touched on the subject of ad hoc networking and mentioned the integration of a reconfigurable radio into an ad hoc node. This is an area that requires further research to investigate how the reconfigurable radio and ad hoc routing protocols can interact and share information that may improve overall communication in the mobile environment.

8.3.3 Security

Security has been touched on at various points throughout this thesis, in both the discussions of software radio security in Section 4.3.5 and in the discussion of OTAR in Section 7.2. This is an important area for future investigation as it is important that as radio systems move into the digital domain they do not succumb to the software problems experienced in mainstream software. These problems include viruses, worms and other malicious code written with the intent of exploiting weaknesses in radio systems. It still has to be proven that this is even possible. Perhaps the reason no such attacks have emerged as of yet is due to the inaccessibility of flexible RF equipment. When devices emerge that allow the development of flexible software radio systems, and particularly devices that allow arbitrary choice of transmit and receive frequency, this barrier will disappear. These types of devices are already beginning to appear, the GNURadio project discussed in Section 2.5 being an example. Increased accessibility to the RF spectrum will open the medium to attackers that will be able to write programs to target security weak implementations of public radio standards.

Several areas for further research are thus required:

- *Analysis of the threat*

  An analysis of the security threats in radio systems should be carried out to determine the feasibility of such attacks. Of particular focus should be to determine whether it is feasible that viruses and worms can affect the software radio as they would the average PC.

- *Cryptographic techniques*

  Air interfaces should be secured using cryptography and secure protocols. Further research should concentrate on deciding what the best security system is for the radio environment.

- *Defensive software development*

  An analysis should be carried out on how to develop secure software for reconfigurable radio systems. This work would ensure that the software development process itself does not introduce exploitable weaknesses into radio systems.

8.3.4 Spectrum Management

Section 7.4 gave a brief overview of spectrum management and how IRIS could potentially be used to develop systems that facilitate the types of systems required in a spectrum managed environment. This is still an emerging area of research and thus many unexplored areas exist.

The following topics require further investigation:

- *Interference temperature*

  The interference temperature metric discussed in Section 7.4 requires further attention. Although the FCC have suggested a metric based on noise per unit bandwidth, further research is required to determine if this is in fact the best way to measure interference. Additional schemes should be researched, proposed and investigated.

- *Spectrum-aware devices*

  Further research needs to be done on the whole topic of developing spectrum-aware devices. Research should concentrate on how they interact, manage their use of the spectrum and relating to security, how a dynamic device that can operate on any frequency can be trusted to not overuse or 'pollute' the RF spectrum.

- *Regulatory issues*

  The regulatory issues surrounding spectrum management need to be investigated as the new concept of dynamic spectrum allocation fundamentally changes the model by which spectrum has been allocated in the past. Research needs to be carried out to determine the best way to manage this resource and in particular how technologies such as reconfigurable radio can be used effectively in a managed spectrum environment.

8.4 Conclusion

This thesis represents a step forward in our understanding of what reconfigurability means in the context of radio systems. Instead of concentrating on delivering software radio using today's hardware technologies, this thesis has asked questions about how these radio systems will manifest themselves in the future. This research has shown that a software-oriented component-based approach to software radio design results in highly reconfigurable radio devices, something that will be important in delivering the next generation of wireless devices.

9 Bibliography

[Abeysekera2002] "FPGA Implementation of a Sigma-delta Architecture Based Digital I.F. Stage for Software Radio", In Proceedings of 15th Annual IEEE International ASIC/SOC Conference, September 2002

[Abu2003] Abu-Al-Saud, Wajih A., Stüber, Gordon L., "Modified CIC Filter for Sample Rate Conversion in Software Radio Systems", IEEE Signal Processing Letters, Vol. 10, No. 5, May 2003

[Ahlquist99] Ahlquist, Gregory., Rice, Michael., Nelson, Brent., "Error Control Coding in Software Radios: An FPGA Approach", IEEE Personal Communications, August 1999

[Akos97] Akos, Denis M., "A Software Radio Approach to Global Navigation Satellite System Receiver Design", Ph.D. Dissertation, Ohio University, August 1997

[Altera] http://www.altera.com

[Armstrong24] Armstrong, Edwin H., "The Super-Heterodyne, its Origin, Development and Some Recent Improvements", In Proceedings of IRE, Vol. 12, October 1924

[Baines95] Baines, Rupert., "The DSP Bottleneck", IEEE Communications Magazine, No. 5, May 1995

[Balen2000] Balen, Henry., "Distributed Object Architectures with CORBA", Cambridge University Press, 2000

[Beach2002] Beach, Mark., Warr, Paul., MadLeod, John., "Radio Frequency Translation for Software Defined Radio", Chapter 2, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Bertrand2002] Bertrand, John., Cruz, John W., Majkrzak, Bryan., Rossano, Thomas., "CORBA Delays in a Software-Defined Radio", IEEE Communications Magazine, February 2002

[Bonser98] Bonser, Wayne., "SPEAKeasy Military Software Defined Radio (presentation)", Symposium on Advanced Radio Technologies, 1998

[Booch87] Booch, Grady., "Software Components with Ada: Structures, Tools, and Subsystems", Benjamin-Cummings, 1987

[Bose99a] Bose, Vanu., Ismert, Michael., Welborn, Matt., Guttag, John., "Virtual Radios", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Bose99b] Bose, Vanu., "Design and Implementation of Software Radios Using a General Purpose Processor", Doctoral Thesis, Massachusetts Institute of Technology, June 1999

[Brannon2002] Brannon, Brad., et al, "Data Conversion in Software Defined Radios", Chapter 4, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Brinegar98] Brinegar, Cornelius., Naishadham, Krishna., "Design of an Integrated RF Filter for the Direct Digitization Front End of Dual GPS/GLONASS Software Radio Receiver", In Proceedings of IEEE Radio and Wireless Conference (RAWCON), August 1998

[Broch98] Broch, Josh., et al, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols", International Conference on Mobile Computing and Networking (MOBICOM), 1998

[Brock2001] Brock, Darren., Mukhanov, Oleg A., Rosa, Jack., "Superconductor Digital RF Development for Software Radio", IEEE Communications Magazine, February 2001

[Brock2002] Brock, Darren., "Superconductor Microelectronics: A Digital RF Technology for Software Radios", Chapter 5, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Buck94] Buck, J. T., Ha, S., Lee, H. A., Messerschmitt, D. G., "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", International Journal of Computer Simulation, Special Issue on Simulation of Software Development, Volume 4, pp. 155-182, April 1994

[Bucknell2000] Bucknell, Paul., "Software Radio and Reconfiguration", In Proceedings of the London Communications Symposium, 2000

[Bucknell2002] Bucknell, Paul., Pitchers, Steve., "Software Download for Mobile Terminals", Chapter 11, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Buracchini2000] Buracchini, Enrico. "The Software Radio Concept", IEEE Communications Magazine, September 2000

[Burns2003] Burns, Paul., "Software Defined Radio for 3G", Mobile Communications Series, Artech House, 2003

[Cadence2002] Cadence Ltd., "FPGA Design With Cadence SPW", Published at website http://www.cadence.com, 2002

[Chapin2001] Chapin, John., Lum, Victor., Muir, Steve., "Experiences Implementing GSM in RDL (The Vanu Radio Description Language)", In Proceedings of Military Communications Conference MILCOM, Communications for Network-Centric Operations, Volume 1, October 2001

[Chapin2002] Chapin, John., "Software Engineering for Software Radios: Experiences at MIT and Vanu, Inc.", Chapter 10, Software Defined Radio: Enabling Technologies, John Wiley and Sons Ltd, 2002

[Chapin2002] Chapin, John., "The Vanu Software Radio System", Software Defined Radio Forum Technical Conference, November 2002

[Chappell96] Chappell, David., "Understanding ActiveX and OLE", Microsoft Press, 1996

[Chappell2002] Chappel, Stephen., Sullivan, Chris., "Handle-C for Co-Processing and Co-Design of Field Programmable System on Chip", Published at Celoxica Website, http://www.celoxica.com

[Cordis] http://www.cordis.lu

[Cowan2000] Cowan, C., Wagle, F., Calton Pu, Beattie,. Walpole, J., "Buffer Overflows: Attacks and Defences for the Vunerability of the Decade", DARPA Information Survivability Conference and Exposition, DISCEX' 00, 2000

[Cummings2002] Cummings, Mark., "Practical Implementation of Commercial SDR RF Front Ends", In Proceedings of 2002 Software Defined Radio Forum Technical Conference (SDR '02), November 2002

[Cummings2002b] Cummings, Mark., "Radio Frequency Front End Implementations for Multimode SDRs", Chapter 3, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Cummings99a] Cummings, Jonathan., "Software Radio for Airborne Platforms", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Cummings99b] Cummings, Mark., Haruyama, Shinichiro., "FPGA in the Software Radio", IEEE Communications Magazine, February 1999

[Cummimgs99c] Cummings, Mark., Heath, Steve., “Mode Switching and Software Download for Software Defined Radio: The SDR Forum Approach”, IEEE Communications Magazine, August 1999

[Dahl70] Dahl, O. J., Nygaard, K. “Simula-67 Common Base Language”, Publications S-22, Norwegian Computing Centre, Oslo 1970

[Dijkstra72] Dijkstra, Edsger. W., “The Humble Programmer”, Communications of the ACM, Volume 15, Issue 10, October 1972

[Dillinger2003] Dillinger, Markus., Buljore, Soodesh., “Reconfigurable Systems in a Heterogeneous Environment”, Chapter 1 of “Software Defined Radio: Architectures, Systems and Functions”, John Wiley & Sons, 2003

[Dixon2001] Dixon, James. L., Wilkes, Joseph E., “A ‘Low-Cost’ Software Radio Test Bed”, Vehicular Technology Conference (VTC), 2001

[Doyle2001] Doyle, L., Kokaram, A., O’Mahony, D., “Error-resilience in Multimedia Applications over Ad Hoc Networks”, In Proceedings of International Conference on Acoustics, Speech and Signal Processing, May 2001

[Doyle2002a] Doyle L., Mackenzie P., O'Mahony D., Nolan K., Flood D., "A General Purpose Processor Component based Software Radio Engine", in Proceedings of the Second European Colloquium on Reconfigurable Radio, June 2002

[Doyle2002b] Doyle, L., Davenport, G., O’Mahony, D., “Mobile Context Aware Stories”, In Proceedings of the IEEE Conference on Multimedia and Expo, August 2002

[Doyle2002c] “Ad hoc Networks – A Welcome Disruption”, IST 2002, November 2002

[Doyle2003] Doyle, L., Mackenzie, P., “Exploring Reconfigurability”, To Appear in Proceedings of 2003 Software Defined Radio Forum Technical Conference (SDR’03), November 2003

[Drew2001] Drew, Nigel. J., Dillinger, Markus. M., “Evolution Toward Reconfigurable User Equipment”, IEEE Communications Magazine, Volume 39, Issue 2, February 2001

[Eichin89] Eichin, Mark. W., Rochlis, John. A., “With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988”, IEEE Symposium on Security and Privacy, May 2001

[Ellingson98] Ellingson, S. W., Fitz, M.P., “A Software Radio-Based System for Experimentation in Wireless Communications”, 48th IEEE Vehicular Technology Conference, Volume 3, May 1998

[FCC2001] Federal Communications Commission, “Software Defined Radios”, Federal Register, Vol 66, No 2, January 2001

[FCC2002] Federal Communications Commission, “Spectrum Policy Task Force Report”, ET Docket No. 02-135, published at website http://www.fcc.gov, November 2002

[FCC2003] Federal Communications Commission, “Notice of Inquiry and Notice of Proposed Rulemaking: Establishment of an Interference Temperature Metric to Quantify and Manage Interference”, ET Docket No. 03-237, published at website http://www.fcc.gov, November 2003

[Fettweis2002] Fettweis, Gerhard., Hentschel, Tim., “The Digital Front End: Bridge Between RF and Baseband Processing”, Chapter 6, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Fines95] Fines, P., “Radio Architectures Employing DSP Techniques”, Microwave and Millimetre-Wave Communications - the Wireless Revolution, IEE Workshop on, 29 November 1995

[Flood2003] Flood, D., Doyle, L., Mackenzie, P., Nolan, K., O'Mahony, D., “Exploiting The Dynamic Flexibility Of Software Radio In FM Broadcast Receivers”, ARRL and TAPR 22nd Digital Communications Conference, September 2003

[Forde2000] Forde, T., Doyle, L. E., O'Mahony, D., "An Evaluation System for Wireless Ad-Hoc Network Protocols", Irish Signals and Systems Conference (ISSC), June 2000

[Fuencisla2002] Fuencisla, María., "Market Impact of Software Radio: Benefits and Barriers", Masters Thesis, Master of Science in Technology and Policy, Massachusetts Institute of Technology, June 2002

[Fujimaki2001] Fijimaki, Akira., et al, "Broad Band Software-Defined Radio Receivers Based on Superconductive Devices", IEEE Transactions on Applied Superconductivity, Vol. 11, No. 1, March 2001

[Gamma95] Gamma, Erich., et al, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[Gast2002] Gast, Matthew S., "802.11 Wireless Networks: The Definitive Guide", O'Reilly, 2002

[Gazis2002] Gazis, Vangelis.,et al, "Evolving Perspectives of 4th Generation Mobile Communication Systems", The 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, Volume 1, 2002

[Glossner2003] Glossner, John., et al, "Multiple Communication Protocols for Software Defined Radio", In Proceedings of the IEE Colloquium on DSP-enabled Radio, 2003

[GNU] http://www.gnu.org/software/gnuradio

[Goldberg83] Goldberg, A., Robson, D., "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading, MA, 1983

[Gosling96] Gosling, A., Joy, B., Steele, G., "The Java Language Specification", Addison-Wesley, 1996

[Green2002] Green, Peter J., Taylor, Desmond P., "Smart Antenna Software Radio Test System", Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA'02), 2002

[Gschwind2002] Gschwind, Thomas., "Adaption and Composition Techniques for Component-Based Software Engineering", PhD Dissertation, Institut für Informationssysteme, 2002

[Gu2002] Gu, Jian., "Zero-IF and Near-Zero-IF Quadrature Receivers for SDR", In Proceedings of 2002 Software Defined Radio Forum Technical Conference (SDR '02), November 2002

[Gunn99] Gunn, James. E., "A Low-Power DSP Core-Based Software Radio Architecture", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Harris2001] Harris, Fredric J., Rice, Michael., "Multirate Digital Filters for Symbol Timing Synchronization in Software Defined Radios", IEEE Journal on Selected Areas in Communications, Vol. 19, No. 12, December 2001

[Haruyama2001] Haruyama, Shinichiro., Morelos-Zaragoza, Robert., "A Software Defined Radio Platform with Direct Conversion: SOPRANO", 54th IEEE Vehicular Technology Conference, Volume 3, October 2001

[Hentschell2000] Hentschell, Tim., Fettweis, Gerhard., "Sample Rate Conversion for Software Radio", IEEE Communications Magazine, August 2000

[Hentschell99] Tim Hentschell, et al., "The Digital Front-End of Software Radio Terminals", IEEE Personal Communications, August 1999

[Honda2001] Honda, Makota., Harada, Hiroshi., Fujise, Masayuki., "Efficient Configuration Data Transmission Scheme for FPGA-based Downloadable Software Radio Communication Systems", 54th IEEE Vehicular Technology Conference, Volume 3, October 2001

[IEEE2000] IEEE, "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-2000, January 2000

[IEEE2001] IEEE, "IEEE Standard Verilog Hardware Description Language", IEEE Std 1364-2001, 2001

[Ikemoto2002] Ikemoto, Kentaro., Kohno, Ryuji., "Adaptive Channel Coding Schemes Using Finite State Machine for Software Defined Radio", In Proceedings of International Symposium on Information Theory, 2002

[Ikonomou99] Ikonomou, Demosthenes., Pereira, Jorge. M., "EU funded R&D on Re-configurable Radio Systems and Networks: The story so far", Infowin Issue on Mobile Communications, published at website http://www.cordus.lu, 1999

[Johnson96] Johnson. David. B., Maltz. David. A., "Dynamic Source Routing in Ad Hoc Wireless Networks", Mobile Computing, volume 353. Kluwer Academic Publishers, 1996

[JTRS2001] Joint Tactical Radio System Joint Program Office, "Software Communications Architecture Specification v2.2", MSRC-5000SCA, November 2001

[JTRS2002] Joint Tactical Radio System Joint Program Office, "Joint Tactical Radio System: SCA Developer's Guide", Document Number: Rev 1.1, June 2002

[Jung2000] Jung, Matthias., "Software Engineering Techniques for Support of Communciation Protocol Implementation", PhD Dissertation, L'Université de Nice – Sophia Antipolis, 2000

[Jub87] Jubin, John., Tornow, Janet D., "The DARPA Packet Radio Network Protocols", In the Proceedings of the IEEE, Volume 75, Issue 1, January 1987

[Kenington2000] Kenington, Peter B., "Power Consumption of A/D Converters for Software Radio Applications", IEEE Transactions on Vehicular Technology, Vol. 49, No. 2, March 2000

[Kenington2002] Kenington, Peter., "Linearized Transmitters: An Enabling Technology for Software Defined Radio", IEEE Communications Magazine, February 2002

[Kennedy95] Kennedy, Joseph., Sullivan Marc C., "Direction Finding and 'Smart Antennas' Using Software Radio Architectures", IEEE Communications Magazine, May 1995

[Kiczales97] Kiczales, Gregor., et al, "Aspect-Oriented Programming", European Conference on Object-Oriented Programming (ECOOP), June 1997

[Kim2001a] Kim, Hahnsang., Turletti, Thierry., "An Esterel-based Development Environment for Designing Software Radio Applications", Rapport de recherche, No. 4256, Unité de recherche INRIA Sophia Antipolis, September 2001

[Kim2001b] Kim, Hahnsang., Turletti, Thierry., Bouali, Amar., "Epspectra: A Formal Approach to Developing DSP Software Applications", Rapport de recherche, No. 4293, Unité de recherche INRIA Sophia Antipolis, October 2001

[Kokozinski2002] Kokozinski, Rainer., Greifendorf, Dieter., Stammen, Joerg., Jung, Peter., "The Evolution of Hardware Platforms for Mobile 'Software Defined Radio' Terminals", In Proceedings of 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), September 2002

[Lackey95] Lackey, Raymond. J., Upmal, Donald. W., "Speakeasy: The Military Software Radio", IEEE Communications Magazine, page 56-61, May 1995

[Lapsley97] Lapsley. Phil, Bier., Jeff, Shoham. Amit, Lee. Edward A., "DSP Processor Fundamentals", IEEE Press, 1997

[Lehr2002] Lehr, William., Merino, Fuencisla., Gillett, Sharon Eisner., "Software Radio: Implications for Wireless Services, Industry Structure, and Public Policy", The 30th Research Conference on Information, Communication, and Internet Policy, Telecommunications Policy Research Conference (TPRC) 2002, September 2002

[Liberty2001] Liberty, Jesse., "Programming C#", O'Reilly and Associates, July 2001

[Löwy2003] Löwy, Juval., "Programming .NET Components", O'Reilly & Associates, Inc, 2003

[Lumpe99] Lumpe, Marcus., "A π-Calculus Based Approach for Software Composition", PhD Dissertation, Institut für Informatik und angewandte Mathematik, Universität Bern, 1999

[Mackenzie2001] Mackenzie P., Doyle L., O'Mahony D., Nolan K., "Software Radio on General-Purpose Processors", In Proceedings of the First Joint IEI/IEE Symposium on Telecommunications Systems Research, Dublin, November 2001

[Mackenzie2002a] Mackenzie, Philip., Doyle, Linda., Nolan, Keith., O'Mahony, D., "Selecting Appropriate Hardware for Software Radio Systems", In Proceedings of 2002 Software Defined Radio Forum Technical Conference (SDR '02), November 2002

[Mackenzie2002b] Mackenzie P., Doyle L., Nolan K.E., O'Mahony D., "An Architecture for the Development of Software Radios on General Purpose Processors", in Proceedings of the Irish Signals and Systems Conference, pp275-280, 2002

[Mackenzie2003] Mackenzie, Philip., Doyle, L. E., Nolan, K. E., Flood, D., "IRIS – A System for Developing Reconfigurable Radios", In Proceedings of the IEE Colloquium on DSP-enabled Radio, September 2003

[MacLeod2001] MacLeod, J.R., Beach, M.A., Warr, P.A., Nesimoglu, T., "A Software Defined Radio Receiver Test-bed", In Proceedings of IEEE Vehicular Technology Conference, Volume 3, October 2001

[Master2002] Master, Paul., Plunkett, Bob., "Adaptive Computing IC Technology for 3G Software-Defined Mobile Devices", Chapter 9, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Mathworks] http://www.mathworks.com

[Mehta2001] Mehta, Mehul., Drew, Nigel., Niedermeier, Christoph., "Reconfigurable Terminals: An Overview of Architectural Solutions", IEEE Communications Magazine, August 2001

[Melby2002] Melby, Jason., "JTRS and the Evolution Toward Software-Defined Radio", In Proceedings of IEEE Military Communications Conference (MILCOM), Volume 2, October 2002

[Michael2002] Michael, Lachlan B., et al, "A Framework for Secure Download for Software-Defined Radio", IEEE Communications Magazine, July 2002

[Mitola] http://ourworld.compuserve.com/homepages/jmitola

[Mitola2000] Mitola, Joseph., "Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio", Doctoral Dissertation, Royal Institute of Technology, Sweden, May 2000

[Mitola2000] Mitola, Joseph., "Software radio architecture: Object-oriented Approaches to Wireless Systems Engineering", John Wiley and Sons, 2000

[Mitola92] Mitola, Joseph., "Software Radios: Survey, Critical Evaluation and Future Directions", National Telesystems Conference (NTC-92), May 1992

[Mitola95] Mitola, Joe., "The Software Radio Architecture", IEEE Communications Magazine, May 1995

[Mitola99a] Mitola, Joe., "Software Radio Architecture: A Mathematical Perspective", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Mitola99b] Mitola, Joseph., Maguire, Gerald Q., "Cognitive Radio: Making Software Radios More Personal", IEEE Personal Communications, August 1999

[Mitola99c] Mitola, Joseph., Chester, David., Haruyama, Shinichiro., Turletti, Thierry., Tuttlebee, Walter., "Globalization of Software Radio", Guest Editorial, IEEE Communications Magazine, February 1999

[Mitola99d] Mitola, Joseph., "Technical Challenges in the Globalization of Software Radio", IEEE Communications Magazine, February 1999

[Mitola99e] Mitola, Joseph., Bose, Vanu., Leiner, Barry M., Turletti, Thierry., "Guest Editorial Software Radios", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Monson2001] Monson-Haefel, Richard., "Enterprise JavaBeans, Third Edition", O'Reilly & Associates, Inc, 2001

[Moore65] Moore, Gordon. E., "Cramming More Components onto Integrated Circuits", Electronics, Volume 38, Number 8, April 19, 1965

[Morris98] Morris, Kevin., Kenington, Peter., "A Broadband Linear Power Amplifier for Software Radio Applications", 48th IEEE Vehicular Technology Conference, Volume 3, May 1998

[Nierstrasz95] Nierstrasz, O., Tsichritzis, D., "Object Oriented Software Composition", Prentice Hall, 1995

[Noblet98] Noblet, C., "Assessing the Over-the-Air Software Download for Reconfigurable Terminal", IEE Colloquium on Personal Communications in the 21st Century (II), February 1998

[Nolan2001] Nolan, K., Doyle, L., O'Mahony, D., & Mackenzie, P., "Modulation Scheme Recognition Techniques for Software Radio on a General Purpose Processor Platform", In Proceedings of the First Joint IEI/IEE Symposium on Telecommunications Systems Research, November 2001

[Nolan2002a] Nolan, K. E., Doyle, L., Mackenzie, P., and O'Mahony, D., "Modulation Scheme Classification for 4G Software Radio Wireless Networks", In Proceedings of the IASTED International Conference on Signal Processing, Pattern Recognition, and Applications (SPPRA 2002), June 2002

[Nolan2002b] Nolan, K. E., Doyle, L., O'Mahony, D., and Mackenzie, P., "Signal Space based Adaptive modulation for Software Radio" In Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC), pp510-515, March 2002

[Nolan2002c] Nolan K. E, Doyle L., Mackenzie P., Ammann M. J., "Software Radio Modulation Scheme Recognition Techniques for ISI Channels", In Proceedings of the Irish Signals and Systems Conference (ISSC), June 2002

[Nolan2003a] Nolan, K. E., Mackenzie, P., Doyle, L., Flood, D., "Implementation of a General-Purpose Processor Platofrm Based OFDM Software Radio System", In Proceedings on the Irish Signals and Systems Conference (ISSC), 2003

[Nolan2003b] Nolan, K. E., Mackenzie, P., Doyle, L., O'Mahony, D., "OFDM/Flash-OFDM Reconfigurable Transceivers Using General Purpose Processors", In Proceedings of IEE Colloquium on DSP-enabled Radio, September 2003

[Nolan2003c] Nolan, K. E., Mackenzie, P., Doyle, L, Flood, D., "A Pattern Recognition Approach for OFDM Frame Synchronisation Using General Purpose Processors", In Proceedings of the IASTED International Conference on Signal Processing, Pattern Recognition and Applications (SPPRA 2003), 2003

[Nolan2003d] Nolan, K. E., Mackenzie, P., Doyle, L., O'Mahony, D., "Flexible Architecture Software Radio OFDM Transceiver System and Frame Synchronisation Analysis", To Appear in Proceedings of IEEE Global Communications Conference, December 2003

[Nyquist24] Nyquist, Harry., "Certain Factors Affecting Telegraph Speed", Bell System Technical Journal, April 1924

[OMG2002] Object Management Group, "Unified Modelling Language 1.4.1", Published at OMG Website, http://www.omg.org, July 2002

[O'Mahony2001] O'Mahony, D., Doyle, L., "Architectural Imperatives for 4th Generation IP-based Mobile Networks", In Proceedings of the Fourth International Symposium on Wireless Personal Multimedia Communications, September 2001

[O'Mahony2002] O'Mahony, Donal., Doyle, Linda., "Beyond 3G: 4th Generation IP-Based Mobile Networks", Wireless IP and Building the Mobile Internet, Chapter 6, p71-86, Artech House, November 2002

[O'Mahony2002b] O'Mahony, D., Doyle, L., "An Adaptable Node Architecture for Future Wireless Networks", Mobile Computing: Implementing Pervasive Information and Communication Technologies, Kluwer series in Interfaces in OR/CS, Kluwer Academic Publishers, 2002

[Patel2000] Patel, Milan., Lane, Phil., "Comparison of Downconversion Techniques for Software Radio", Proceedings of the London Communications Symposium, 2000

[Patel2002] Patel, Milan., Darwazeh, Izzat., O'Reilly, John. J., "Bandpass Sampling for Software Radio Receivers, and the Effect of Oversampling on Aperture Jitter", IEEE Vehicular Technology Conference, 2002

[Patti99] Patti, John J., Husnay, M., Pintar, Joseph., "A Smart Software Radio: Concept Development and Demonstration", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Pereira2001] Pereira, Jorge M., "Reconfigurable Radio: the evolving perspectives of different players", Proceedings of 12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, Volume 1, 2001

[Pereira99] Pereira, Jorge M., "Beyond Software Radio, towards Re-configurability across the whole System and across Networks", Proceedings of 50th IEEE Vehicular Technology Conference, Volume 5, pp.19-22, September 1999

[Pereira2000] Pereira, Jorge M., "Re-Defining Software (Defined) Radio: Re-Configurable Radio Systems and Networks", IEICE Transactions on Communications, Volume E83-B, No. 6, June 2000

[Pérez2001] Pérez-Neira, Ana., Mestre, Xavier., Fonollosa, Javier Rodríguez., "Smart Antennas in Software Radio Base Stations", IEEE Communications Magazine, February 2001

[Perkins99] Perkins, Charles E., Royer, Elizabeth M., "Ad hoc On-Demand Distance Vector Routing.", Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp. 90-100, February 1999

[Razavilar99] Razavilar, Javad., Rashid-Farrokhi, Farrokh., Ray Liu, K.J., "Software Radio Architecture with Smart Antennas: A Tutorial on Algorithms and Complexity", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, April 1999

[Reichhart99] Reichhart, Stephen P., Youmans, Bruce., Dygert, Roger., "The Software Radio Development System", IEEE Personal Communicatons, August 1999

[Ribeiro2001] Ribeiro-Justo, George R., Imre, Sándor., "Intelligent Decision-making within 4th Generation Wireless Networks", International Conference on Internet Computing, Volume 1, 2001

[Rice2001] Rice, Michael., Dick, Chris., "Maximum Likelihood Carrier Phase Synchronization in FPGA-Based Software Defined Radios", In Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, Volume 2, May 2001

[RSA78] Rivest, R. L., Shamir, A., Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM Archive, Volume 21, Issue 2, February 1978

[Salkintzis99] Salkintzis, Apostolis K., Nie, Hong., Mathiopoulos, P. Takis., "ADC and DSP Challenges in the Development of Software Radio Base Stations", IEEE Personal Communications, August 1999

[Sametinger97] Sametinger, Johannes., "Software Engineering with Reusable Components", Springer-Verlag, Town, 1997

[Schacherbauer2001] Schacherbauer, W., et al, "A Flexible Multiband Frontend for Software Radios Using High IF and Active Interference Cancellation", IEEE MTT-S International Microwave Symposium Digest, Volume 2, May 2001

[Schneider99] Schneider, Jean-Guy., "Components, Scripts and Glue: A conceptual framework for software composition", PhD Dissertation, Institut für Informatik und angewandte Mathematik, Universität Bern, 1999

[Scourias96] Scourias, John., "Overview of GSM: The Global System for Mobile Communications", Master of Mathematics Term Paper, University of Waterloo, Ontario, Canada, Published at website http://ccnga.uwaterloo.ca/~jscouria, 1996

[SDRForum] http://www.sdrforum.org

[SDRForum2] Software Defined Radio Forum, "Software Defined Radio Semantics", Published at website http://www.sdrforum.org

[Semenov99] Semenov, Vasili K., Likharev, Konstantin K., "RSFQ Front-end for a Software Radio Receiver", IEEE Transactions on Applied Superconductivity, Volume 2, Issue 2, June 1999

[Seskar99b] Seskar, Ivan., Mandayam, Narayan B., "Software-Defined Radio Architectures for Interference Cancellation in DS-CSMA Systems", IEEE Personal Communications, August 1999

[Smith97] Smith, Michael J. S., "Application-Specific Integrated Circuits", Addison-Wesley, 1997

[Srikanteswara2000a] Srikanteswara, Srikathayayani., Reed, Jeffrey H., Athanas, Peter., Boyle, Robert., "A Soft Radio Architecture for Reconfigurable Platforms", IEEE Communications Magazine, Volume 38, Issue 2, February 2000

[Srikanteswara2000b] Srikanteswara, Srikathayayani., Hosemann, Michael., Reed, Jeffrey H., Athanas, Peter M., "Design and Implementation of a Completely Reconfigurable Soft Radio", IEEE Radio and Wireless Conference (RAWCON), September 2000

[Steinberg99] Steinberg Soft- und Hardware GmbH, "ASIO Interface Specification v 2.0: Steinberg Audio Stream I/O API", 1999

[Steinheider2003] Steinheider, J., Lum, V., Santos, J., "Field Trials of an All-Software GSM Base Station", 2003 Software Defined Radio Technical Conference, Orlando, November 2003

[Streifinger2003] Streifinger, M., Müller, T., Luy, J.F., Biebl, E.M., "A Software-Radio Front-End for Microwave Applications", In Proceedings on Topical Meeting on Silicon Monolithic Integrated Circuits in RF Systems, April 2003

[Sun2001] Sun Microsystems, "Enterprise JavaBeans Specification Version 2.0", Published at website http://java.sun.com/beans

[Sun97] Sun Microsystems, "JavaBeans API Specification", Published at website http://java.sun.com/beans

[System-C] http://www.systemc.org

[Szyperski2002] Szyperski, Clemens., "Component Software: Beyond Object-Oriented Programming, Second Edition", Addison-Wesley, 2002

[Tennenhouse95] Tennenhouse, David L., Bose, Vanu G., "SpectrumWare – A Software-Oriented Approach to Wireless Signal Processing", ACM Mobile Computing and Networking 95, Berkeley, CA, November 1995

[Tennenhouse96] Tennenhouse, David L., Bose, Vanu G., "The SpectrumWare Approach to Wireless Signal Processing", Wireless Network Journal, Volume 2, No. 1, 1996

[Thara2002] Thara, V.B., Siddiqi, M. U., "Power Efficiency of Software Radio Based Turbo Codec", In Proceedings of IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering, Volume 2, October 2002

[Tsurumi99] Tsurumi, Hiroshi., Suzuki, Yasuo., "Broadband RF Stage Architecture for Software-Defined Radio in Handheld Terminal Applications, IEEE Communications Magazine, February 1999

[Tuttlebee98] Tuttlebee, Walter., "Software Radio – Impacts and Implications", IEEE 5th International Symposium on Spread Spectrum Techniques and Applications, 1998

[Tuttlebee99a] Tuttlebee, Walter., "Software Radio Technology: A European Perspective", IEEE Communications Magazine, February 1999

[Tuttlebee99b] Tuttlebee, Walter., "Software-Defined Radio: Facets of a Developing Technology", IEEE Personal Communications, April 1999

[VanVliet2000] Van Vliet, Hans., "Software Engineering: Principles and Practice, Second Edition", John Wiley & Sons, Ltd, 2000

[Vasconcellos2000] Vasconcellos, Brett W., "Parallel Signal-Processing for Everyone", Masters Thesis, Massachusetts Institute of Technology, February 2000

[VenturCom] http://www.vci.com

[Watson2002] Watson, J., "Adaptive Computing IC Technology Enabled SDR and Multifunctionality in Next-Generation Wireless Devices", In Proceedings of 2002 Software Defined Radio Forum Technical Conference (SDR '02), November 2002

[Welborn99a] Welborn, M.L., "Direct Waveform Synthesis for Software Radios", Wireless Communications and Networking Conference (WCNC), September 1999

[Welborn99b] Welborn, M.L., "Narrowband Channel Extraction for Wideband Receivers", In Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Volume 3, March 1999

[Wepman95] Wepman, Jeffery., "Analog-To-Digital Converters and their Applications in Radio Receivers", IEEE Communications Magazine, No. 5, May 1995

[Wiesler2002] Wiesler, Anne., Jondral, Friedrick K., "A Software Radio for Second-and Third-Generation Mobile Systems", IEEE Transactions on Vehicular Technology, Vol. 51, No. 4, July 2002

[Willink2002] Willink, Edward., "The Waveform Description Language", Chapter 13, Software Defined Radio: Enabling Technologies, John Wiley and Sons, 2002

[Withers91] Withers, D. J., "Radio Spectrum Regulation and Management", Institution of Electrical Engineers, Peter Peregrinus Ltd, 1991

[Xilinx] http://www.xilinx.com

[Yang2002] Yang, Lie-Liang., Hanzo, Lajos., "Software-Defined-Radio-Assisted Adaptive Broadband Frequency Hopping Multicarrier DS-CSMA", IEEE Communications Magazine, March 2002

[Yourdon79] Yourdon, Edward., Constantine, Larry L., "Structured Design", Prentice Hall, 1979

[Zhao2002] Zhao, Minjian., "A Non-coherent GMSK Receiver for Software Radio", In Proceedings of 53rd IEEE Vehicular Technology Conference, Volume 3, May 2001

10 Appendix

10.1 Methods Exposed by a Radio Component

A Radio Component exposes the following methods.

Methods providing information about a Radio Component:

```
char* GetDefaultXML();
char* GetInfo();
char* GetName();
char* GetVersion();
char* GetAuthor();
char* GetValue(char *name);
```

Methods for dealing with parameters:

```
int GetNumParameters();
char* GetParameterName(int identifier);
char* GetParameterDefaultValue(int identifier);
char* GetParameterInfo(int identifier);
char* GetParameterDataType(int identifier);
bool IsParameterDynamic(int identifier);
bool IsParameterDynamic(char *name);
bool SetValue(char *name, char *value);
bool SetValue(int parameterId, bool value);
bool SetValue(int parameterId, char value);
bool SetValue(int parameterId, unsigned char value);
bool SetValue(int parameterId, short value);
bool SetValue(int parameterId, unsigned short value);
bool SetValue(int parameterId, int value);
bool SetValue(int parameterId, unsigned int value);
bool SetValue(int parameterId, __int64 value);
bool SetValue(int parameterId, unsigned __int64 value);
bool SetValue(int parameterId, float value);
bool SetValue(int parameterId, double value);
bool SetValue(int parameterId, char* value);
bool SetValue(int parameterId, unsigned char* value, unsigned int size);
bool GetValue(int parameterId, bool* value);
bool GetValue(int parameterId, char* value);
bool GetValue(int parameterId, unsigned char* value);
bool GetValue(int parameterId, short* value);
bool GetValue(int parameterId, unsigned short* value);
bool GetValue(int parameterId, int* value);
bool GetValue(int parameterId, unsigned int* value);
bool GetValue(int parameterId, __int64* value);
bool GetValue(int parameterId, unsigned __int64* value);
bool GetValue(int parameterId, float* value);
bool GetValue(int parameterId, double* value);
bool GetValue(int parameterId, char* value, unsigned int size);
bool GetValue(int parameterId, unsigned char* value, unsigned int* size);
```

Methods for dealing with commands:

```
int GetNumCommands();
char* GetCommandName(int identifier);
char* GetCommandInfo(int identifier);
char* GetCommandDeclaration(int identifier);
int GetCommandDeclarationValue(int identifier);
```

Methods for dealing with ports:

```
int GetNumPorts();
char* GetPortName(int identifier);
char* GetPortInfo(int identifier);
char* GetPortDeclaration(int identifier);
int GetPortDeclarationValue(int identifier);
```

Methods for dealing with events:

```
int GetNumEvents();
char* GetEventName(int identifier);
char* GetEventInfo(int identifier);
char* GetEventDataType(int identifier);
char* GetEventDeclaration(int identifier);
int GetEventDeclarationValue(int identifier);
int GetEventCallbackType(int identifier);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackData callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackBool callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackByte callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackByteUnsigned callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackInt16 callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackInt16Unsigned callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackInt32 callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackInt32Unsigned callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackInt64 callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackInt64Unsigned callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackFloat callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackDouble callback);
bool SubscribeToEvent(char *event, int identifier, pEventCallbackString callback);
```

10.2 Code Generator Commands

The following commands can be used with the code generator described in Section 6.2.3:

Declaration	Description
`//@component <description>`	Indicates the class declaration immediately following the declaration is a Radio Component. Example: `//@component GMSK Modulator`
`//@version <version number>`	Identifies the version of the component. Example: `//@version 1.3b`
`//@author <author's name>`	Indicates the author of the component. Example: `//@author Philip Mackenzie`
`//@event <name> <datatype> <info>`	Indicates that an event will be fired from the component. Example: `//@event SignalReceived int signal received`
`//@port <name> <info>`	Indicates that this component supports an input data port for receiving data from external sources. Example: `//@port ModulateData modulates received data`
`//@command <name> <info>`	Indicates that this component exposes a command which can be fired from external control logic. Example: `//@command ResetFilter resets the filter`
`//@param <name>`	Indicates that the next member variable declaration will be exposed as a parameter of the component. Example: `//@param frequency cutoff`
`//@default <default value>`	Specifies the default value for the parameter. Example: `//@default 3.14`
`//@dynamic`	Indicates that the parameter is dynamic Example: `//@dynamic`