

Software Architecture Design of a Software Defined Radio System

JOHANNES J. CRONJÉ

*Thesis presented in partial fulfilment of the requirements for the degree
Master of Science in Electronic Engineering
at the University of Stellenbosch*

SUPERVISOR: Mr G-J van Rooyen
CO-SUPERVISOR: Prof J.G. Lourens

December 2004

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Abstract

The high pace of technological advancement enables the realisation of ever more advanced mobile communications standards with more functionality than simple voice communications. The hardware that is used to implement the radio sections of these systems generally require long design cycles, much longer than the design cycles of the other components of a communications system. Another problem is that, once new communications standards are introduced, the current hardware platforms used in the terminal equipment becomes obsolete because they can generally not be used with the new standards. This has serious cost implications for both the service provider and the consumer, because both parties have to acquire new equipment to be able to use the new standards.

An elegant solution to the above issues is to use *software-defined* radio sections to replace the hardware radio components. New communications standards can then be supported by simply loading new software onto the equipment, provided the maximum processing capacity of the processor(s) that the software runs on can accommodate the bandwidth requirements of that specific standard.

This thesis investigates the ideas behind software defined radio and also describes the design and implementation of a software architecture that can be used to implement software defined radios on general-purpose platforms such as personal computers.

Opsomming

Die hoë tempo van tegnologiese vordering maak dit moontlik om baie gevorderde mobiele kommunikasie standaarde te implementeer wat meer funksionaliteit bied as blote spraakkommunikasie. Die hardeware wat gebruik word om die radios van sulke stelsels te implementeer neem gewoonlik langer om te ontwikkel as die ander komponente van die stelsels. Die ander probleem is dat hierdie hardeware gewoonlik nie hergebruik kan word wanneer nuwe kommunikasie standaarde in gebruik geneem word nie omdat die standaarde nie versoenbaar is nie. Dit het tot gevolg dat beide die verbruiker en die diensverskaffer groot bedrae geld moet spandeer om die nuwe tegnologie te kan gebruik.

'n Elegante oplossing vir hierdie probleme is om gebruik te maak van radios waarvan die funksionaliteit in *sagteware* gedefiniëer word. Nuwe kommunikasie standaarde kan dan gebruik word deur slegs die nodige sagteware op die toerusting te laai, solank die verwerkingskapasiteit van die mikroverwerkers in die stelsel die benodigde bandwydte kan akkommodeer.

Hierdie tesis ondersoek die konsepte van sagteware-gedefiniëerde radio en beskryf die ontwerp en implementering van 'n sagteware argitektuur vir die implementering van sagteware-gedefiniëerde radios op veeldoelige platforms soos persoonlike rekenaars.

Acknowledgements

- Thank you to my parents for always believing in me and supporting me all the way, also for their encouragement to finish my work and most of all for their financial support during the course of my studies.
- I would like to thank my supervisor, Mr. Gert-Jan van Rooyen, for his support and guidance throughout the course of my studies. Thank you for always being willing to assist me whenever I ran into problems and wanted to explore new ideas.
- Thank you to my co-supervisor, Professor Johan Lourens, for his advice and encouragement to finish my thesis.
- I would also like to thank all the people on the omniORB support mailing list for all their assistance with my CORBA related issues. Thank you in particular to Duncan Grisby, Thomas Lockhart, Davy Croonen, Kendall Bailey, Adriaan Joubert, Gary Duzan and Clarke Brunt.
- I would also like say thank you to the people on the dsp_help mailing list for always answering my (somewhat trivial) programming-related questions, in particular Gerhard Esterhuizen, Jan Pool and Louis Cordier.
- Thank you to everybody of the Digital Signal Processing and Telecommunications Lab of the University of Stellenbosch for their advice, help and moral support all the days and late nights in the lab. Thanks to Ludwig Schwardt, Hermann Engelbrecht, André du Toit and Albert Visagie for their programming advice, to Francois Cilliers for keeping the computer systems up-and-running at all times, to Schalk Visser for keeping the legendary coffee in good supply and to everyone else for all the wonderful company.
- I would also like to thank all my friends for all their wonderful support and love. Without you I would probably not be where I am today, and know that you have all made everlasting impressions on my heart.
- Lastly I would like to thank my Heavenly Father for all the grace, talents and opportunities He has bestowed on me. Without Him I am nothing.

Contents

1	Introduction	1
1.1	The software defined radio philosophy	1
1.2	Defining the term “Software-defined radio”	2
1.3	The purpose of this research	2
1.4	Thesis overview	3
2	Theoretical Background	4
2.1	Software-defined radio basics	4
2.1.1	Receiver architectures	4
2.1.2	Sample rate conversion	6
2.1.3	Digital signal generation	7
2.2	Hardware platforms	9
2.2.1	Signal converters	9
2.2.2	Field programmable gate arrays	11
2.2.3	Digital signal processors	11
2.2.4	General-purpose processors	12
2.2.5	Application-specific integrated circuits	13
2.2.6	Comparing these platforms	13
2.2.7	Conclusion	16
2.3	Software design concepts	16
2.3.1	Object-oriented analysis and design	16
2.3.2	Choosing a programming language	17
2.4	Other software-defined radio projects	18
2.5	Conclusion	19
3	System Architecture	20
3.1	The ISO OSI Layer Approach	20
3.2	Overview of the architecture	21
3.2.1	The converter layer	23
3.2.2	The Subcontroller	24
3.2.3	The CORBA interface	25

3.2.4	The Main Application	26
4	The Converter Layer	27
4.1	Basic converter design	27
4.2	Design considerations	29
4.3	Class <code>sdr_converter</code>	30
4.3.1	The attribute system	30
4.3.2	Data input and output in <code>sdr_converter</code>	33
4.3.3	Miscellaneous methods available in class <code>sdr_converter</code>	35
4.3.4	Exceptions in <code>sdr_converter</code>	36
4.4	Input buffers: class <code>sdr_buffer_base</code>	38
4.4.1	Data abstraction	38
4.4.2	Choosing a container for the buffers	39
4.4.3	Class <code>sdr_buffer_base</code>	39
4.4.4	A practical buffer class	42
4.4.5	Proprietary data types	43
4.4.6	A practical buffer class using the <code>sdr_data</code> type	43
4.5	An ethernet interface converter implementation	44
4.6	Conclusion	47
5	The Subcontroller	48
5.1	Subcontroller overview	48
5.1.1	Component management	49
5.1.2	Scheduling execution	50
5.2	Adding components: the SDR factory class	51
5.2.1	The benefits of using the factory model	52
5.2.2	Using the factory	52
5.3	The subcontroller interface	53
5.3.1	Subcontroller attributes	53
5.3.2	The RPC interface	53
5.4	Conclusion	56
6	The CORBA Layer	57
6.1	The RPC mechanism: text versus binary	57
6.2	Some CORBA background information	58
6.3	Technical aspects of CORBA	59
6.3.1	A note on the terms “server” and “servant”	59
6.3.2	The interface definition language	60
6.3.3	ORB interface	62

<i>CONTENTS</i>	vi
6.3.4 Object adapters	62
6.3.5 The CORBA naming service	63
6.4 Integrating CORBA into the SDR architecture	66
6.4.1 Interface definition	66
6.4.2 User exceptions in CORBA	67
6.4.3 The Subcontroller ORB	69
6.5 Conclusion	69
7 The Main Application: SDR Builder	70
7.1 Interface requirements	70
7.2 The SDR Builder user interface	71
7.2.1 Selecting a user interface library	71
7.2.2 User interface layout	72
7.2.3 Subcontroller and component management	76
7.3 Threads in SDR Builder	77
7.4 Some last remarks about GUI libraries	78
7.5 Conclusion	79
8 Evaluation of the Architecture	80
8.1 Evaluation process overview	80
8.1.1 System configurations	81
8.1.2 Terminology	81
8.2 Data throughput performance	82
8.2.1 Purpose of the experiment	82
8.2.2 Experimental setup	83
8.2.3 Results of the experiment	83
8.2.4 The effect of attribute modification overhead on data throughput performance	87
8.3 Memory usage	89
8.3.1 Purpose of the experiment	89
8.3.2 Experimental setup	90
8.3.3 Results of the experiment	90
8.4 Latency measurements	90
8.4.1 Purpose of the experiment	91
8.4.2 Experimental setup	91
8.4.3 Results of the experiment	92
8.5 RF performance evaluation	93
8.5.1 Purpose of the experiment	93
8.5.2 Experimental setup	93

<i>CONTENTS</i>	vii
8.5.3 Results of the experiment	93
8.6 Conclusion	98
9 Conclusion	102
9.1 Overview of the work	102
9.1.1 Other software defined radio projects	102
9.1.2 Hardware platforms	103
9.1.3 The current SDR platform	103
9.2 Future Research	104
A Class sdr_converter: API Overview	109
A.1 API Reference Documentation	109
B The Unified Modelling Language	111
B.1 UML Class Diagrams	111

List of Figures

2.1	A software radio system	4
2.2	A software-defined radio system	5
2.3	The sample rate conversion process	6
2.4	Signal sample values in a LUT	8
2.5	Constructing an analogue sine wave from a LUT	8
2.6	A pulse output DDS system	9
3.1	The OSI Seven-Layer Model [5]	20
3.2	The architecture of the software defined radio system	22
3.3	The association between the various components of the system	23
3.4	UML association diagram of the subcontroller	25
4.1	A basic converter representation	28
4.2	The strategy design pattern [7]	28
4.3	<code>sdr_converter</code> attributes	30
4.4	<code>sdr_converter</code> attribute table	31
4.5	The structure of the output ports	34
4.6	Exception mechanism	37
4.7	A UML Class Diagram Of The <code>sdr_buffer_base</code> Class	40
4.8	Class <code>sdr_buffer</code> inheritance diagram	42
4.9	Class <code>sdr_buffer</code>	43
4.10	An example of a distributed radio system	45
5.1	A conceptual subcontroller representation	48
5.2	Structure of the component map container	49
5.3	Round robin scheduling processing sequence example	50
5.4	Creating a component with the factory	52
6.1	The ORB and its interfaces to servants and clients	62
6.2	The SDR INS Context Structure	64
6.3	Function call flow diagram showing CORBA encapsulation	68

LIST OF FIGURES

ix

7.1	SDR Builder main window	73
7.2	The subcontroller menu	73
7.3	Adding a subcontroller to the system	74
7.4	The converter menu and attribute submenu	75
7.5	Linking two converters	75
7.6	Querying a component for its available attributes	76
7.7	Subcontroller and component management in SDR Builder	77
7.8	Flowchart of the slotSystemRun method	78
8.1	The converters used to measure the throughput of the software radio system	83
8.2	Time required to process 1 million samples as a function of the number of components in the system	84
8.3	Data throughput rate of the test system	86
8.4	Execution times of processing 1 million samples on various systems. Refer to Table 8.1 for details of these systems.	87
8.5	Increase in processing times of 1 million samples with attribute modification overhead	88
8.6	Allocated memory as a function of the number of components in the system	89
8.7	Latency in a system due to batch processing	91
8.8	The converters used to measure the latency of the software radio system	91
8.9	The latency of the system as a function of the number of components	92
8.10	Spectrum of the frequency modulated RF signal that was generated with the C-Media audio controller.	94
8.11	Plot of in-phase versus the quadrature component of the baseband FM signal	95
8.12	Spectrum of the frequency modulated RF signal that was generated with the Creative Labs audio controller	96
8.13	Three spectrum plots showing the spectrum of a baseband frequency modulated sine wave signal generated by the Creative Labs audio controller. (a) The frequency modulated baseband signal spectrum. (b) The spectrum of the SMIQ output with no signal output from the sound card. (c) The resulting frequency modulated signal.	97
8.14	Feedback lag in a system that employs batch processing	99
8.15	Latency of the system versus throughput	100
B.1	Two UML Class Diagrams Showing Attribute And Function Types	111
B.2	Two UML class diagrams showing association and inheritance	112

LIST OF FIGURES

B.3	A Detailed UML Class Diagram Of The <code>output_port</code> Structure	113
B.4	A Detailed UML Class Diagram Of The <code>attribute_struct</code> Structure	113
B.5	A Detailed UML Class Diagram Of The <code>sdr_data</code> Class	113
B.6	A Detailed UML Class Diagram Of The <code>sdr_buffer_base</code> Class	113
B.7	A Detailed UML Class Diagram Of The <code>sdr_converter</code> Class	114

List of Tables

4.1	Methods available for manipulating attributes	32
4.2	Methods available for manipulating the ports	35
4.3	Miscellaneous methods available in <code>sdr_converter</code>	36
4.4	Specialised exception classes	38
4.5	Methods available for inserting data into a buffer	40
4.6	Methods available for retrieving data from a buffer	41
4.7	Miscellaneous methods available in the buffer class	42
4.8	Methods available in the <code>sdr_buffer_data</code> class	44
5.1	The subcontroller RPC interface	54
6.1	CORBA helper functions	66
6.2	CORBA IDL function declarations	67
7.1	The SDR Builder interface layout	72
8.1	The various system configurations used to evaluate the architecture	81
A.1	Methods Available in Class <code>sdr_converter</code>	109
B.1	UML visibility modifiers	112

Nomenclature

Acronyms

ADC	Analogue-to-digital converter
ASIC	Application-specific integrated circuit
BB	Baseband
BOA	Basic object adapter
BPF	Band-pass filter
CAD	Computer-aided design
CDMA	Code-division multiple access
CORBA	Common object request broker architecture
COTS	Commercial off-the-shelf
CPU	Central processing unit
C++	Pronounced "C plus plus", programming language
DAC	Digital-to-analogue converter
DDS	Direct digital synthesis
DECT	Digital enhanced cordless telecommunications
DII	Dynamic invocation interface
DNS	Domain name service
FIR	Finite impulse response
FFT	Fast Fourier transform
FM	Frequency modulation
FPGA	Field-programmable gate array
GIOP	General inter-ORB protocol
GPP	General-purpose processor, general-purpose platform
GSM	Global system for mobile communications, originally Groupe Spéciale Mobile
GUI	Graphical user interface
IANA	Internet assigned numbers authority
IC	Integrated circuit
IDL	Interface definition language
IF	Intermediate frequency

LIST OF TABLES

xiii

IIOIP	Internet inter-ORB protocol
IIR	Infinite impulse response
INS	Interoperable naming service
IOR	Interoperable object reference
IP	Internet protocol
I/Q	In-phase/quadrature
LNA	Low-noise amplifier
LO	Local oscillator
LPF	Low-pass filter
LUT	Lookup table
MAC	Multiply-and-accumulate
ORB	Object request broker
POA	Portable object adapter
ROM	Read-only memory
RPC	Remote procedure call
SDR	Software-defined radio
SFDR	Spurious-free dynamic range
SQNR	Signal-to-quantisation noise ratio
SRC	Sample rate conversion
STL	Standard template library
SWR	Software radio
TCP	Transmission control protocol
TUI	Text-based user interface
UID	Unique identifier
VCO	Voltage-controlled oscillator
XML	Extensible markup language

Units and unit multipliers

dB	Decibel
Hz	Hertz
Sa	Sample
s	Second
μ	micro, 10^{-6}
k	kilo, 10^3
M	mega, 10^6

Chapter 1

Introduction

1.1 The software defined radio philosophy

The rapid advancements in communications technology has made the world we live in smaller than ever before. Two or more people can communicate with each other from almost anywhere in the world as if they were in the same room. People have also become more and more accustomed to, even dependent upon, mobility as mobile communications technology evolved. A few years ago a mobile phone was seen as a status item; nowadays it is seen as a very important commodity.

The problem is that newer, better communications standards cannot replace old standards without disrupting the service to customers. These problems are mostly caused by the fact that mobile terminals are not reconfigurable after they are manufactured. The new standards are therefore implemented alongside old standards to enable existing customers to have uninterrupted service. Implementing a new standard also requires the customer to purchase new terminal equipment that are compatible with the new standard, and often incompatible with the old standard. This hampers technological advances because when a standard is entrenched people are very reluctant to invest in new technology.

The philosophy behind *Software-defined radio* (hereafter SDR) is to try to eliminate these hardware-related issues. For instance, if a new modulation scheme is implemented for some mobile phone standard, all that is needed at the client side is a software upgrade. Another possibility is to have one terminal that is, for example, both a GSM transceiver and a DECT transceiver. Switching between the various supported standards can either be done according to service availability or user selection.

SDR also has its drawbacks, though. The biggest problem is that the fact that processing *Radio frequency* (RF) signals at frequencies in the high hundreds of megahertz range requires very fast microprocessors running at very high clock frequencies. The high processing speeds in turn cause the power consumption of these processors to

be beyond the capability of the batteries of mobile equipment. A workaround for this problem is to use a heterodyne receiver front-end to mix the signal down to an intermediate frequency before the signal is digitised. Processing RF signals at intermediate frequencies reduces the power consumption and processing ability of processors considerably.

1.2 Defining the term “Software-defined radio”

According to the SDR Forum [22]:

Software Defined Radio (SDR) is a collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multi-mode, multi-band, multi-functional wireless devices that can be enhanced using software upgrades. As such, SDR can really be considered an enabling technology that is applicable across a wide range of areas within the wireless industry.

Whenever the term “Software-defined radio” is used in this document, the above definition shall apply.

1.3 The purpose of this research

The purpose of this research can be summarised as follows: create a software architecture or framework to enable rapid development of new software radio components and applications. The architecture should be as flexible as possible without sacrificing signal processing efficiency, and should ideally also be portable across various hardware platforms with as few implementation issues as possible. This was done according to the following identified outcomes:

- Study other software defined radio projects to gain insight into the software radio paradigm.
- Study various hardware platforms to identify integration and portability issues.
- Study object-oriented design methods.
- Specify suitable development platforms.
- Specify the interfaces between the various modules of the system.
- Implement a working demonstration system.

1.4 Thesis overview

The first three outcomes listed above form the basis of chapter 2. Documentation on various existing software-defined radio projects was consulted to get a better idea of the design constraints that apply to SDR systems.

The interface specifications are described in chapters 3 through 7. An overview of the architecture is given in chapter 3 to give the reader an overall feel for the system architecture. The architecture features a three-tier layered structure that can be divided into a component layer (lowest level), a subcontroller layer (intermediate level) and a user interface layer (highest level). A strong modular design approach was followed throughout the design of the architecture to ensure orthogonality between the various layers as well as between the components in the component layer.

The subsequent chapters follow the same bottom-up approach that was used in the development of the architecture. In chapter 4 the lowest level of the architecture is described in detail from the various considerations that had to be taken into account to a thorough description of the component layer interface. The main aim was to create a component layer that is as efficient, portable and fast as possible. It must also be very modular in nature to enable any type of component to interface with any other type of component.

Chapter 6 provides a detailed explication of the basics of implementing a CORBA-based RPC system in the architecture. It also gives an overview of the CORBA interface between the subcontroller and the user interface before the rest of the architecture is discussed.

The subject of chapter 5 is the design of the subcontroller layer. The subcontroller is responsible for managing all the components in its memory space and also provides a platform-independent communication mechanism with the user interface layer. The various elements of the subcontroller are described in detail before a description of the interface is given.

The design and implementation of the user interface is discussed in chapter 7. It is a very simple user interface application used to demonstrate how a user can interact with the underlying SDR system.

A working SDR FM transmitter system was implemented to demonstrate a practical application of the system. It was also used to evaluate the performance of the system along with a few specially designed components. The evaluation of the architecture is discussed in chapter 8.

Chapter 9 concludes the document and provides an overview of the the statements and conclusions made in this document.

Chapter 2

Theoretical Background

In theory, there is no difference between theory and practise. In practise, there is.

— Chuck Reid

In this chapter the theoretical aspects of software-defined radio will be investigated. Firstly some background information about the technology is presented. The hardware platforms that are likely to be used in SDR design and implementation will also be discussed. This will be followed by a discussion of some software analysis and design aids.

2.1 Software-defined radio basics

This section provides an overview of some of the principles of software-defined radio.

2.1.1 Receiver architectures

The ideal software radio

A *software radio* (SWR) is sometimes also referred to as pure or ideal software radio. The SWR is a system that uses minimal analogue components and processes the *radio frequency* (RF) signal directly. A block diagram of a software radio is shown in Figure 2.1.

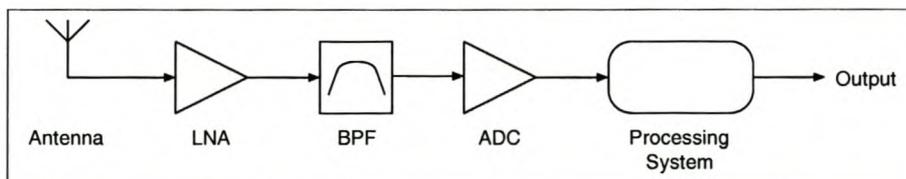


Figure 2.1: *A software radio system*

With current hardware technology it is not yet possible to implement true SWR.¹ The operating frequencies of current telecommunication standards are simply too high, and become even higher as more channels and higher bandwidth are needed.

Another attribute of such systems is that these systems should be able to deal with signals with extremely high dynamic ranges. This is especially true for cases where the system employs a wideband signal converter to sample communication signals with very narrow channel bandwidths [11]. Refer to section 2.2.1 for a more detailed explanation.

Software-defined radio

A software *defined* radio is a system that does not process the RF signal directly—the signal is first mixed down to an *intermediate frequency* (IF) or directly to *baseband* (BB) before it is processed in the digital domain. A block diagram of a typical software defined radio is shown in Figure 2.2.

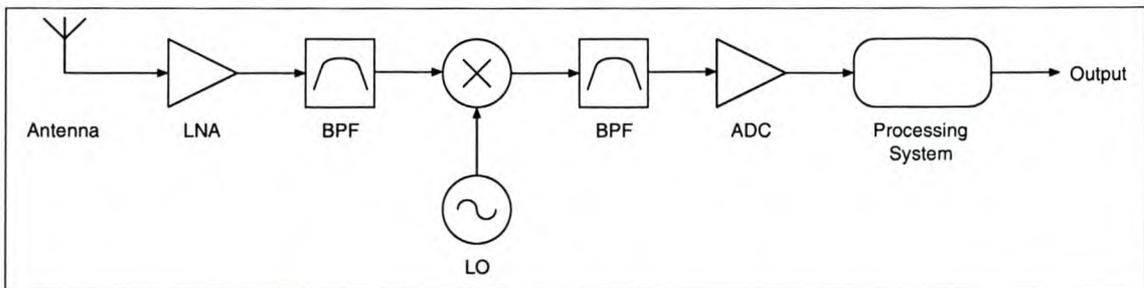


Figure 2.2: A software-defined radio system

The system is so called because all the modulation and signal processing is done by software so that the function of the radio is defined by the software running on it. Mixing the signal down to IF or BB is still achieved with analogue mixers and not in software, thus it is not a “pure” software radio.

SDR offers an attractive interim solution on the way to SWR. The drawback is that extra hardware is needed, and with every piece of specialised hardware added to the system the flexibility decreases.

In-phase/quadrature (I/Q) down mixing of the signal is also possible and it can be achieved in the digital domain using a software I/Q downconverter. It is also possible to implement a modified version of the above architecture with the single downconversion stage replaced with an I/Q downconverter that mixes the input down to baseband before it is converted to a digital signal.

¹Keep in mind that “true” software radio can process *any* RF signal.

2.1.2 Sample rate conversion

The various different communications standards in use today generally make use of unique bit rates in their signals. Information transmitted with CDMA, for example, have fixed bit rates, and these information signals are multiplied with a high-frequency pseudo-noise signal that has a fixed chip rate that is several times higher than the information bit rate. GSM signals, on the other hand, have a single fixed bit rate that is several times lower than the CDMA chip rate.

The generic nature of SDR calls for a system that can process all these different signals, and do so efficiently. One approach to cope with these diverse bit rates is to employ a clock generator for each supported standard. Unfortunately this is severely limiting to the flexibility of the system and it is also a very inefficient solution because it requires that several additional components be added to the system. A more attractive solution is to use a single clock signal and convert the sample rate of the signal to the desired value [10].

The process of *sample rate conversion* (SRC) can perhaps be understood most easily if it is thought of as resampling the signal with a different sampling rate after it has been reconstructed to analogue form [10]. A block diagram of this process is shown in Figure 2.3.

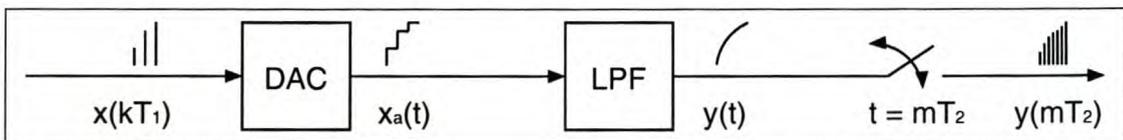


Figure 2.3: The sample rate conversion process

This whole process can be realised in the digital domain (that is, without actually converting the signal to analogue format) by finding an appropriate transfer function for the system shown in Figure 2.3.

The problem of aliasing is still present if the desired sampling rate is lower than the current sampling rate due to the fact that the SRC process represents resampling after reconstruction. In cases where the desired sample rate is lower than the original sample rate the signal has to be decimated before it is resampled. This implies that the LPF must have a sufficiently low cutoff frequency to avoid aliasing when the signal is resampled.

If the desired sampling rate is higher than the original sampling rate the aliasing problem disappears. This up-sampling process (also called interpolation) can be achieved by various means, like zero-insertion, zero-order hold or zero insertion and raised-cosine filtering [20].

According to Reed [20], it is more computationally efficient to perform the conver-

sion in several stages if the conversion factor is relatively large ($> \sim 50$). He shows as an example that decimating a signal with a 90kHz sample rate with a factor of 90 using a two-stage decimator reduces the number of mathematical operations by a factor of 8 over a single-stage decimator.

2.1.3 Digital signal generation

To implement a radio transmitter with a SDR system, as with any other type of transmitter, requires the use of a carrier signal. This carrier is modulated with the message signal before the resultant signal is passed to the antenna.

It is possible but highly inefficient and generally not feasible to generate this carrier signal in the analogue domain and then convert it to a digital signal with an ADC. A better approach will be to generate the carrier signal directly in the digital domain. Systems that employ digital signal synthesis are immune to component drift that plagues analogue systems, offer better frequency control and -resolution and offers a simplified hardware architecture [20].

There are two basic ways to synthesise a digital signal. One makes use of a set of values that represent the amplitude of the signal at various points in time over a period. The other does not use any memory and generates a periodic pulse signal that can be used to create other waveforms [20].

Using lookup tables to perform DDS

As mentioned above, a *lookup table* (LUT) is a collection of the samples of a signal that is stored in memory (usually ROM). These sample values are read from the memory and used as if it were a signal that has been digitised by an ADC. Note that the sampled waveform can have an arbitrary form, but usually sine waves are generated with this method because of their widespread use in telecommunications systems.

Figure 2.4 shows how a few samples of a sine wave is stored in a LUT. The most efficient algorithm used to generate sample values are by Taylor series approximation. In practical systems the number of sample values in the LUT is much higher to obtain a more accurate representation of the signal. The memory required to store the LUT increases linearly as the number of samples increase. By exploiting half- or even quarter wave symmetry the number of samples stored in the LUT is reduced by 50% and 75% respectively.

Figure 2.5 shows how the values in the LUT shown in Figure 2.4 can be used to generate an analogue signal. The frequency of the generated sine wave can be varied by reading every n 'th sample in the table instead of every sample, in other words by varying the phase increment. For a LUT with M entries and a clock period of T_{clk} the

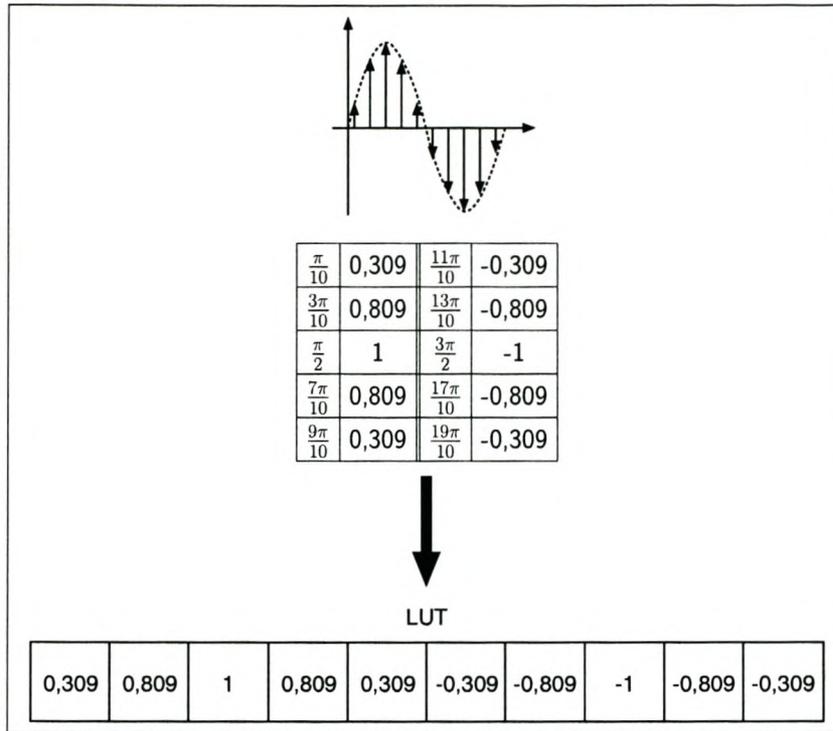


Figure 2.4: Signal sample values in a LUT

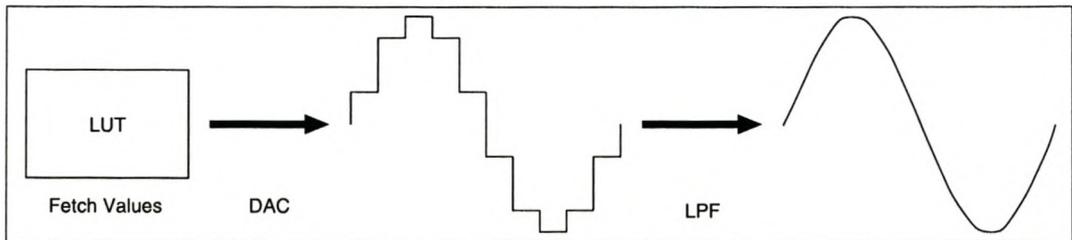


Figure 2.5: Constructing an analogue sine wave from a LUT

output frequency will be

$$f_{out} = \frac{n}{M \cdot T_{clk}} \text{ Hz} \quad \text{or} \quad f_{out} = \frac{2\pi \cdot n}{M \cdot T_{clk}} \text{ rad.s}^{-1}$$

The phase increment is defined as

$$\Delta\phi = 360 \cdot \frac{n}{M} \text{ degrees} \quad \text{or} \quad \Delta\phi = 2\pi \cdot \frac{n}{M} \text{ radians.}$$

Pulse output

A pulse output DDS system generates periodic pulses, square- and sawtooth waveforms. These waveforms can be used to create other waveforms (like sine waves)

without the need for any memory devices.

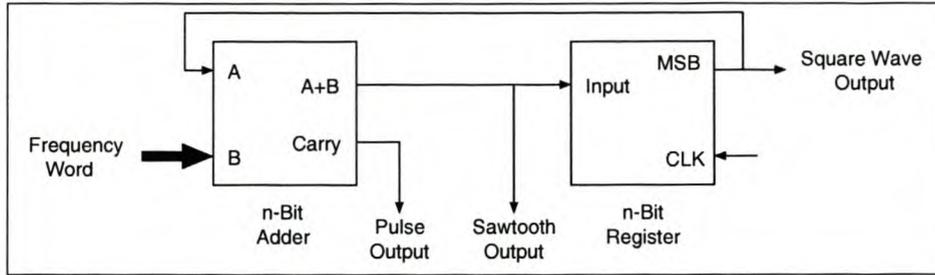


Figure 2.6: A pulse output DDS system

Figure 2.6 shows a pulse output DDS system and the outputs that generate the various waveforms. The frequency of the various waveforms are determined by the frequency word (F_r), the number of adder and register bits and the register clock frequency as follows [20]:

$$F_{out} = \frac{F_r}{2^n} \cdot F_{clk}$$

The output frequency can be varied by changing the frequency word. A larger frequency word value will cause the adder's accumulator to overflow sooner than a smaller value, leading to an increase in output frequency.

2.2 Hardware platforms

A practical SDR system consists of both hardware and software subsystems. The various hardware components that form part of a SDR system will be discussed in this section.

2.2.1 Signal converters

In a SDR receiver, a converter is needed to transform the input signal from a continuous analogue signal to a discrete digital signal. This converter is known as an *analogue-to-digital converter* (ADC), and it works by taking samples of the input signal at fixed intervals and converting these values to a numeric format that can be interpreted by equipment in the digital domain.

Likewise, a SDR transmitter needs a converter to transform the discrete digital output signal to a continuous analogue signal. This converter is known as a *digital-to-analogue converter* (DAC), and it is used to convert data from the numerical format used in the digital domain back to analogue signal values.

Signal converters² present one of the greatest challenges to SDR at the moment. As with most engineering problems, signal converter design presents a tradeoff. Converters with a high enough sample rate for use in SDR systems have too low bit resolution while converters with adequate resolution can not sample at the rates required by today's mobile phone systems.

At the time of writing, the fastest 14-bit³ ADC available has a sample rate of 105MSa/s, and the fastest 14-bit DAC has a sample rate of 800 MSa/s. This implies maximum frequency values of around 50MHz for receivers and just under 400MHz for transmitters, well below the 900MHz used by GSM, 1.9-2.2GHz used by UMTS and the 2.4GHz ISM band used by DECT. This necessitates the use of IF downmixing with current hardware technology.

In order for a SWR/SDR system to be able to handle mobile communications signals it will almost certainly feature wideband converters. This is necessary to cover all the available channels of the standards that the system will support. Using a wideband converter to digitise or synthesise a signal for a communication system that has relatively narrow channels presents an interesting problem: the converter must have a very high dynamic range.

For the sake of argument, assume that the digitisation bandwidth of a particular SDR system is fixed, that is, the sample rate of the analogue-to-digital converter and the anti-aliasing filter characteristics are fixed. The number of channels that will be digitised by this system depends on the bandwidth of the channels. The narrower the channels are, the more will be digitised and vice versa.

When more than one channel of a particular standard is simultaneously digitised by the system, the adjacent channels will interfere with each other. This interference increases the dynamic range of the signal, and as more channels are digitised the dynamic range increases further. When only one channel is digitised the dynamic range of the signal is minimised because there is no interference from adjacent channels. It follows that by adjusting the digitisation bandwidth (that is, the sampling rate and filter cut-off) to equal the channel bandwidth of the standard being used this behaviour can be exploited to reduce the dynamic range the system has to cope with.

A more detailed discussion can be found in [11] and [20].

²In the current context "signal converters" include both ADCs and DACs.

³14-Bit converters were chosen to illustrate the point because they offer a reasonably good ideal dynamic range and SQNR of about 86dB.

2.2.2 Field programmable gate arrays

A *field programmable gate array* (FPGA) is an *integrated circuit* (IC) that contains an array of logic gates that have programmable interconnects and logic functions. These interconnections and functions can be redefined at any time after the manufacturing of the FPGA to change its functionality.

FPGAs are highly suited to signal processing algorithms that are more parallel than sequential in nature, like FIR or IIR filters, correlation-, convolution- and FFT algorithms [4]. This is due to the fact that FPGAs are reconfigurable *logic circuits* and not processors. A processor carries out a series of instructions in sequence that operate on a single data stream, whereas a FPGA implements a logic circuit, which can handle more than one data stream simultaneously.

FPGAs can be configured in various ways. The first is to use a graphical *computer-aided design* (CAD) tool. Such CAD tools are provided by the FPGA vendor because each vendor's FPGA implementation is unique. These tools also provide predefined components, in a supplied library, for the convenience of the developer. Examples of such components include shift registers, adders, and so on.

The other alternative is to use a *hardware description language* (HDL) like Verilog or VHDL.⁴ These languages are used to describe the structure of the components on the FPGA, unlike conventional programming languages (like C) that contain a series of instructions that the processor must execute. A VHDL compiler translates the code to a form suitable to be used to configure the FPGA. The aforementioned library components included in CAD tools are simply macros that contain VHDL code.

Both these approaches have merit—CAD tools are generally much faster when used to build a system with relative “standard” components, and a HDL can be used to construct a highly specialised system. CAD tools usually allow the developer to add specialised components to the library. This approach allows the developer to use a HDL in a CAD environment, blurring the distinction between the two⁵.

2.2.3 Digital signal processors

A *digital signal processor* (DSP) is a single stream processor that is optimised for high-speed arithmetic calculations.

Each DSP model has its own instruction set, and although one vendor may have several processors that share some instructions, no two are likely to have the same instruction set. This makes code portability somewhat difficult, especially if the code

⁴The “V” in VHDL is derived from VLSI, short for *very large scale integration*.

⁵This is much like using assembly language in a high-level language like C.

has been written in low-level languages like assembly language. On the other hand, high-level languages used for programming DSPs (like C) enable greater portability, but the lack of support for C++ on many DSP platforms makes code re-use by means of inheritance impossible.

The sequential nature of DSP processors make it suited to signal processing algorithms that are sequential in nature [4]. Unfortunately the processes cannot be parallelised, thus to implement a signal processing algorithm that is parallel in nature is very inefficient compared with an implementation on a FPGA.

One drawback of DSPs is that the power consumed by the DSP is proportionate to the square of its clock frequency. Furthermore, as feature sizes⁶ decrease, heat dissipation increases along with power efficiency and speed. Due to the high frequencies employed in mobile phone standards, very high DSP clock rates are needed and this may keep pure software radio from being implemented on mobile terminals for some time because of the high power consumption and heat dissipation [4].

Most DSP processors implement “hard” real-time scheduling. In a “hard” real-time scheme all time-critical functions must finish execution before their deadlines are reached [2]. This feature allows easier modelling of the system and ensures that functions execute when expected.

2.2.4 General-purpose processors

Bose [2] did a study into the feasibility of using *general-purpose processors* (GPP) to implement software radios. GPPs were long believed to be unsuitable for use in a RF signal processing environment due to the lack of real-time processing support by the operating systems running on these processors.

In his dissertation Bose investigated all the aspects of the GPP environment and how they differ from proper real-time environments. The first aspect that must be taken into account is the fact that, unlike real-time environments, the execution times on a GPP platform are variable, making prediction of real-time performance near impossible. Worst-case performance must be used as a guideline for real-time performance prediction, and can be as much as one order of magnitude worse than real figures.

Variable execution is caused by both the hardware and software used in a GPP environment. Cache memory and dynamic execution are examples of hardware elements that influence the execution times. These factors improve the average performance of a system, but make the worst-case performance even worse. System interrupts and

⁶Feature Size is the size of the smallest structure that can be created on a semiconductor wafer. As an example, the Intel Pentium 4 has a feature size of 130nm, and a 90nm manufacturing process is currently in development.

the operating system's scheduler are examples of software elements that cause variable execution times.

The migration of real-time processing of audio and video signals from specialised hardware to desktop processors has driven multi-media development of both hardware and software elements of the GPP environment. GPPs also offer parallelism,⁷ a feature unavailable on DSPs, that enables acceptable performance with inherently parallel computations such as those listed above.

GPPs also suffer from the problem that plagues DSPs, namely that the power dissipated is proportionate to the square of the clock speed of the processor. Specialised instructions that allow more optimal use of clock cycles can be added to a processor's instruction set. Together with these instructions, dynamic clock management allows the clock frequency to be dynamically adapted to the load of the processor, and can also help reduce power dissipation.

Bose concluded that:

...general purpose processors and operating systems are capable of implementing a broad class of real-time signal processing applications. Furthermore, the scope of the applications that can be implemented will increase as processors and memories get faster.

2.2.5 Application-specific integrated circuits

Of all the processor platforms discussed here, *application-specific integrated circuits* (ASICs) are probably the least suited as an SDR platform. An ASIC is precisely what its name suggests: an IC that was designed for a specific application. ASICs have the benefit that they are probably the best at processing signals in the system they were designed for, but they offer very little programmability.

2.2.6 Comparing these platforms

Pucker [19] wrote an article about the differences between *commercial off-the-shelf* (COTS) ASICs, DSPs and FPGAs, and where each fits into the SDR picture. Some additional observations are made in the case of GPPs as Pucker did not consider them in his article.

The criteria he selected for evaluating these platforms are discussed next.

⁷For example, Intel's *multi-media extensions* (MMX) technology offers parallelism at instruction level, super scalar architectures offer parallelism at the instruction unit level.

Programmability

Changing the layout of FPGAs and programming of DSPs so these devices have added or even totally different functionality is a relatively straightforward process. The new configuration of the device, written in a human-readable language, just needs to be translated to binary form and then uploaded to the device to take effect.

GPPs also offer the level of reconfigurability of FPGAs and DSPs. Due to the fact that systems built with GPPs have highly flexible operating systems that support multi-tasking and object-oriented programming languages these processors tend to be the most reconfigurable of all. To implement new functionality only requires running another or revised executable on top of the operating system. Uploading new code to the processor is therefore not necessary.

COTS ASIC devices are usually designed for a single purpose. Some COTS ASIC devices offer very limited programmability in the sense that they support more than one function, and the function can be selected dynamically. No new functionality can be added to an ASIC, though, which makes the use of ASICs in a design less future-proof than other approaches.

Level of integration

Advancements in manufacturing technology enables more and more functionality to be integrated into single devices. This enables FPGAs, GPPs and DSPs to become more and more versatile and powerful because more functionality relates to broader applicability.

ASICs, on the other hand, become *less* flexible as the level of integration increases due to the fact that, as more specialised components are added to an ASIC, its universal applicability decreases. It follows that, as integration levels increase, the number of discrete ASICs in a system must also increase to ensure multiple standards support, and this is not practical.

Development cycle

FPGA development cycles are considerably longer than the other platforms discussed here—large designs⁸ can literally take hours to synthesise and route. Typical routing times for modern day FPGAs are somewhere on the order of 400,000 to 3,000,000 gates per hour [29], [30] (depending on the FPGA and synthesis tool), which means that debugging can be an arduous process, especially with slower design tools. Designs

⁸“Large” is defined here as any design that utilises in excess of one million gates on an FPGA.

done on a FPGA platform are therefore first modelled and tested extensively in the CAD environment before the design is committed to the device for testing.

Initial DSP and GPP platform development cycles can also take some time, but debugging the software on these platforms is generally not as time-consuming as with FPGAs. Compiling and loading software that runs on these processors typically takes only seconds to complete, even for large designs. Another benefit offered by GPPs are code re-use by means of inheritance, a concept that forms part of the *object-oriented design* (OOD) process (refer to section 2.3.1 for more information).

ASICs, on the other hand, enjoy shorter development cycles because the bulk of the development work is hardware related, with only little software needed to access the programmable features of the ASIC.

Performance

Comparing the performance of these devices illustrates how the reconfigurability, clock speed and architecture of a device influences its performance. FPGAs generally run at low clock speeds as the clock is mostly just necessary for data flow and not signal processing. Due to the ability of FPGAs to process parallel data streams, though, it offers performance that compares to and even better DSP performance with some applications.

DSPs run at a higher clock speeds than FPGAs, but suffer from the fact that only one stream of data can be processed. DSPs are very good at performing tasks that involve decision making (like recognition systems) and algorithms that are sequential in nature.

GPPs suffer a bit in the performance comparison because they (generally) are not explicitly optimised for signal processing applications. Some do feature instructions previously only found on DSPs and this helps to boost their signal processing performance. In a head-to-head comparison, though, DSPs outperform GPPs due to their specialised instruction sets.

ASICs offer the best performance of the lot, and generally outperforms DSPs and FPGAs with any given function. This is because an ASIC is highly optimised for the particular function that it must perform.

Power

As already mentioned, the power consumption of DSPs and GPPs is proportional to the square of the clock speed. Another factor that increases power consumption is feature size. As feature sizes decrease, the leakage current between the components on the IC increases [8]. The maximum possible clock frequency also increases due to the shorter

paths between components, causing a further increase in power consumption.

FPGAs generally have better power consumption figures than DSPs and GPPs, perhaps due to the lower clock rates employed in FPGAs. ASICs also feature very low power consumption figures compared to DSPs and GPPs and again maybe this is due to the fact that ASICs run at lower clock rates than DSPs and GPPs. Another factor that contributes to the lower power consumption figure is the highly optimised designs employed in ASICs.

2.2.7 Conclusion

From the above discussion it is clear that no single platform is ideally suited to SDR systems. The above discussion yields the following results regarding the applicability of the various platforms:

- ASICs have low power consumption and high performance, but offer no reconfigurability.
- DSPs feature excellent reconfigurability and high speed at the expense of high power consumption. They are also best for decision making algorithms or any non-parallel signal processing algorithms.
- GPPs share most features with DSPs, and offer better code reuse possibilities. Some GPPs also offer some degree of parallelism at the instruction level.
- FPGAs are best suited to parallel signal processing algorithms, and have better power consumption than DSPs and GPPs. They are also highly reconfigurable, albeit that the reconfiguration is much slower than is the case with DSPs.

This thesis proposes that, until a platform is developed that caters specifically for SDR systems, the optimal solution is a hybrid solution that contains FPGAs and DSPs or GPPs.

2.3 Software design concepts

2.3.1 Object-oriented analysis and design

Object-oriented analysis (OOA) is used to model and analyse the design requirements and overall architecture of a system [21]. A tool like UML is typically used for this purpose.

To ensure that a class conforms to a specified interface, OOA provides the abstract base class concept. When a class inherits from an abstract base class, it must implement

all the pure virtual methods of the base class. If it does not, the derived class is also considered an abstract base class, and an instance of the class cannot be created.

When the OOA process is finished, *object-oriented design* (OOD) is used to translate the components of the architecture to programming language constructs like classes or method descriptions.

The final step is to implement these constructs in a object-oriented programming language.

2.3.2 Choosing a programming language

The development of SDR systems requires a programming language that is more oriented to hardware programming than application development. This is because programming languages that are tailored for application development are not necessarily optimised for speed.

SDR systems need to have real-time or near-real-time performance in order to process signals effectively and compete with their analogue counterparts. The programming language used must therefore be able to deliver highly optimised native binary code. Programming in a processor's assembly language yields the best performance at the expense of long development cycles. C is a good option—it can include assembler code, it delivers high-performance binary code and most DSP manufacturers today offer C compilers for their processors.

C++ is also a good option, as it can also include assembler code and most C++ compilers can also compile C code. A subset of C++, known as Embedded C++, is available for embedded environments [18]. C++ is an object-oriented programming language, and this enables the use of base classes to enable code re-use through inheritance. Most good C++ compilers offer code optimisation to further enhance the performance of the compiled binary code as well as processor-specific optimisations.

Java provides most of the features of C++ and also includes an automatic memory management system called “garbage collection”. Due to the fact that Java compiles to byte code and not to native machine code it runs on a virtual machine, and this makes the performance of the language about one order of magnitude slower than C++ binary code [6]. Microprocessors that can execute compiled Java byte code appeared in late 2002. These processors offer the portability of Java with the speed of native compiled code. This is unfortunately only true for embedded environments, as the code must still run through virtual machines on most other platforms.

Another popular object-oriented language is Python. It is an interpreted language and therefore also suffer a performance impedance like Java. It is very well suited to GUI or application development but not hardware programming.

C++ was eventually chosen as the main development language because of its good

balance between high performance binary code, portability and object-oriented language support. The other factor that contributed to this decision is the fact that most C++ compilers can also compile C and assembler code that is included in the C++ source code, if the need arises to write a specific routine in one of these languages. C must be used if C++ is not supported on a specific platform (for example embedded platforms or DSPs).

2.4 Other software-defined radio projects

Vanu Inc., a company started by Vanu Bose, implemented a hand-held software radio on a Hewlett-Packard iPAQ handheld computer [26]. The iPAQ uses an Intel XScale processor and the operating system the software runs on is Linux. The analog transceiver is housed in an “expansion pack” and operates from 100 to 475MHz. The configuration currently supports commercial analog FM services. Prototypes operating at frequencies up to 900MHz, with support for mobile phone standards, are in development. Little else is known about this system, therefore it will not be evaluated here.

Mackenzie and others implemented a software radio receiver system that supports analog AM and FM radio signals on a GPP platform [14]. They developed a “dynamic stack” architecture that features dynamic reconfiguration by removing and inserting layers from the stack to alter the functionality of the system. The data is passed between the layers of the system in special message blocks, a similar concept to the data packets used in the TCP/IP protocol. This ensures that the data is processed by the correct demodulation block if more than one block is present on the stack. A possible drawback of this system is the overhead introduced by parsing the message header. No performance evaluation information is available for this architecture, therefore no comments can be made about the efficiency of this approach.

They also developed a modulation scheme recognition component as part of the stack [15]. An important goal for the recognition system was that it should not adversely affect the real-time performance of the system, and it did not. The system developed was theoretically able to recognise both digital and analogue modulation schemes, but only analogue cases were considered in tests and simulations. It was found that the threshold SNR for analogue modulation scheme recognition in their system was approximately 6,5dB.

2.5 Conclusion

The main focus of this chapter was to provide insight to the enabling technologies involved in developing software-defined radio systems. Aspects that were covered include an overview of a few software-defined radio front-ends, the digital generation of signals and sample rate conversion. A comparison of the various hardware platforms available for use in software-defined radio systems was also done, to determine which platform would be best suited for use in a SDR system. Lastly, other SDR projects were also briefly discussed.

In the next chapter the architecture is introduced from a global perspective, to provide a clear overview of the system as a whole, before it is further dissected and described in subsequent chapters.

Chapter 3

System Architecture

This chapter provides an overview of the software architecture that was developed for use in a software-defined radio system. It is included here so that the design choices made in the various parts of the architecture can be judged in terms of the system level design of the architecture. Detailed discussions of the various parts of the architecture is done in subsequent chapters.

3.1 The ISO OSI Layer Approach

In February 1947, the *International Standards Organisation* (ISO) was formed “to facilitate the international coordination and unification of industrial standards” [13]. One of the most well-know ISO standards is the *Open Systems Interconnect* (OSI) Seven-Layer model [5]. The OSI was created in the early 1980’s in an effort to standardise networking interfaces, because before the OSI different vendors implemented their own proprietary network interfaces and protocols.

Each layer in the model is defined such that it is not dependent on how the layer beneath it is implemented. All that the current layer requires of the layer beneath it is that data be passed to it *in the correct format*. A good demonstration of this concept is the *carrier pigeon internet protocol* (CPIP) concept [12], and an implementation [1]. In this protocol, carrier pigeons are used to implement the physical layer of a TCP/IP system,

Application
Presentation
Session
Transport
Network
Data Link
Physical

Figure 3.1: *The OSI Seven-Layer Model* [5]

to show that the network layer is indeed not dependent on the implementation of the layers beneath it.

The modular architecture of the OSI model is a very attractive attribute. It allows every layer to be independently designed, developed and tested. After the various layers have been implemented, a whole system can be created by simply linking the layers together.

The OSI seven-layer model was not directly used in the SDR architecture. The SDR architecture was, however, designed with the concepts presented in the OSI model in mind.

3.2 Overview of the architecture

To illustrate the basic concepts of a SDR system, the requirements of the various subsystems are briefly discussed before an overview of the architecture is given.

As the development of the SDR project progresses, more and more components will be developed for use in complex radio systems. To ensure that these components can also be reused in other SDR systems they need to have standardised interfaces. A standard interface also ensures that any two components can be connected together.

These components also need to each perform a well-defined function. As an example, consider COTS phase-locked loop ICs. These circuits perform a well-defined function and also have a well-defined hardware interface (albeit generally not standardised between manufacturers). Due to the atomic nature of these circuits they can be used wherever phase-locked loop functionality is needed.

The atomic nature of these components means that some form of management system is needed to construct a working SDR system. This management system will be responsible for controlling the flow of information in the system, and it will also be responsible for managing the components.

The last part that is required is a user interface system. The user interface will be responsible for translating input from a user to a format that is suitable for the management system or the components. This is an optional component, as not all radio systems require user interaction to function.

As mentioned in the previous section, a modular approach to system design produces a system with well-defined sections that can be implemented independently. This orthogonality between the subsystems of the system ensures that a subsystem implementation can be replaced with a new or updated system without affecting the subsystems around it. Modular system design also allows the components of the system to be re-used in other systems with totally different functionality. A graphical representation of the architecture is shown in figure 3.2.

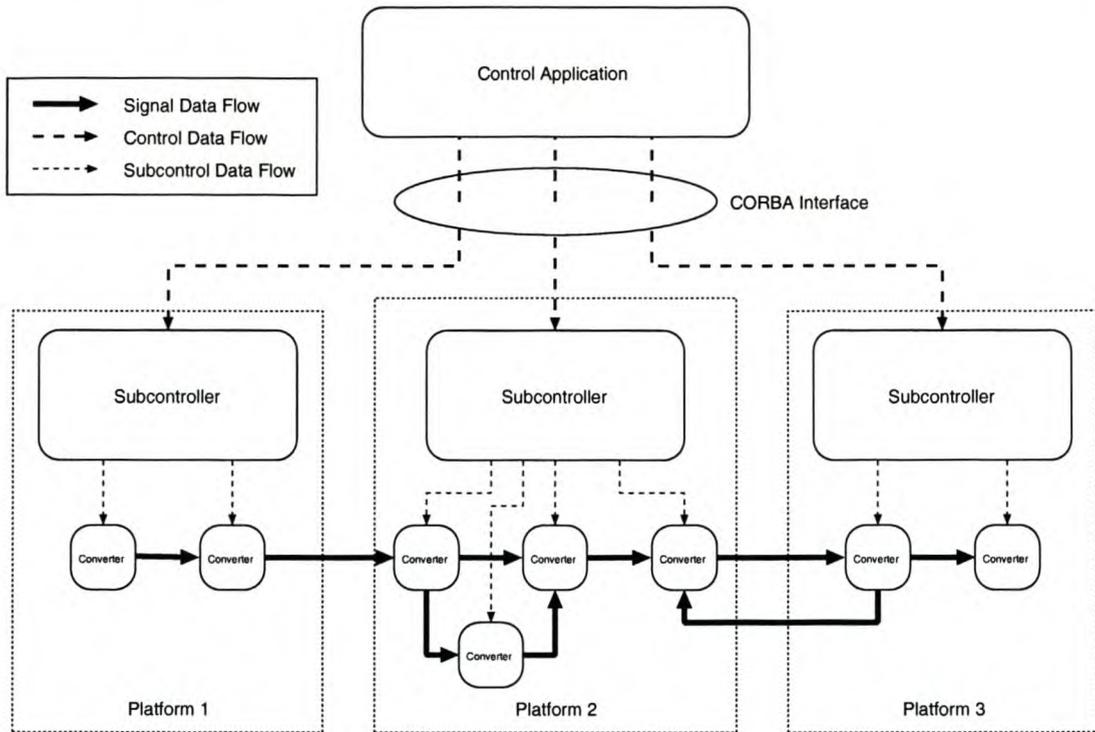


Figure 3.2: *The architecture of the software defined radio system*

The lowest level of the system is called the converter layer. The converter layer represents the equivalent of the analogue subsystems (such as the VCO, filters, mixers and so on) of a hardware-defined radio system. All information signal processing is done in the converter layer. Every converter is an atomic unit and is totally independent from other converters.

The middle layer is called the subcontroller layer. The subcontroller does not have a direct equivalent in hardware-defined radio systems. It is responsible for handling all communications with the top-level application. It must also manage all the converter components under its control and must also relay control data from the top-level application to the converter components.

The control application provides a user interface for the system and it is responsible for managing the subcontrollers in the system. It allows the user to change the configuration of the system as well as subcontroller- and converter parameters. Communication between the control application and the subcontrollers is achieved via a CORBA interface.

The management responsibilities of the system are handled by both the subcontroller layer and the control application. This was done to enable SDR systems to be implemented on distributed systems or on systems where more than one subcontroller has to be used. An example of such a SDR system is a system in which the signal is at

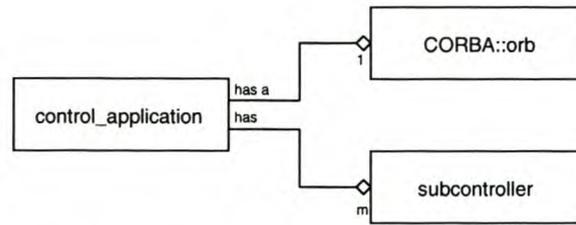


Figure 3.3: *The association between the various components of the system*

some point processed by specialised hardware.

Figure 3.3 shows a UML association diagram of the system from the perspective of the control application. It shows that the control application contains m subcontroller containers that each contain a CORBA reference to a subcontroller as well as all the ID's of the converters present in that subcontroller. The control application also has a single ORB object reference for all CORBA communications.

A detailed overview of the above concepts is presented in the following sections.

3.2.1 The converter layer

The converter layer is responsible for all the information signal processing functionality of the system. The design of the converter layer must therefore be highly optimised for high-speed signal processing to maximise the throughput rate and minimise the amount of latency in the system.

The converter layer is made up of a collection of converters that derive from an abstract base class called `sdr_converter` that provides a fixed interface for all converters of the system. Due to the standard interface these converters can be used interchangeably in a software radio system to redefine the functionality of the radio.

Information signal flow between converters (the horizontal flow illustrated in figure 3.2) is based on a “push” principle. This simply means that once a converter has finished processing a sample (or a set of samples) the result is placed into the input buffer of the next converter as opposed to an output buffer in the current component. The push system was chosen because it represents a more intuitive representation of analogue signal flow. A “pull” system could have been implemented in which the current converter fetches samples from the preceding converter, but this approach offers no advantages over a push system from a signal flow perspective.

To successfully implement a push signal flow architecture the input buffers must have the ability to be dynamically resizable. This is needed because the current converter has no way of knowing how many samples the preceding block will generate during its signal processing cycle. The size of the input buffers have to be monitored in

the subcontroller because it can determine the optimal buffer sizes according to the system setup and processing conditions. The design and implementation of the converter layer will be discussed in detail in chapter 4.

3.2.2 The Subcontroller

The converter layer mentioned above provides the lowest level building blocks of the software radio system. A system is needed that can control the converters as well as the flow of data through the software radio.

The subcontroller layer is responsible for the management of the various converter components of the SDR system. New converters are added to the system with a converter factory that can create converter instances as they are needed. This factory is based on the abstract factory design pattern [7] and its implementation is described in section 5.2. The main reason for using a factory is that it enables new converter classes to be added to the converter library without modifications to the subcontroller source code, and this enables the addition of new functionality to the system without interrupting the signal flow.

Controlling the flow of information between the various converters in the system is also handled by the subcontroller. This flow control mechanism can be implemented in various ways, from naive sequential execution to threaded simultaneous execution. Naive scheduling is achieved by simply letting the converters process samples in an arbitrary order without any constraints with respect to execution time. This approach is commonly known as round-robin scheduling. A more complex form of sequential scheduling that mimics the scheduling used in real-time systems can also be implemented [2]. This approach will reduce latency at the expense of added subcontroller complexity and possibly some extra computational overhead for the scheduling mechanism. Yet another way to implement scheduling is with the help of extra process threads to achieve some degree of parallelism in the system. The scheduling approach used in the current subcontroller implementation will be discussed in detail in section 5.1.2.

The subcontroller must also monitor the size of the input buffers to prevent unbounded buffer growth, as mentioned in the previous section. This is needed because the converters do not offer any buffer size control as the size of the input buffer is dependent on the configuration of the system and can therefore not be determined by the converters themselves. Specifying a fixed buffer size is possible, but it is generally not feasible because the optimal buffer size cannot be determined at design or compile time. The optimal buffer size for a converter is also dependent on the signal processing algorithm implemented in that converter, and this value may be different for different algorithms.

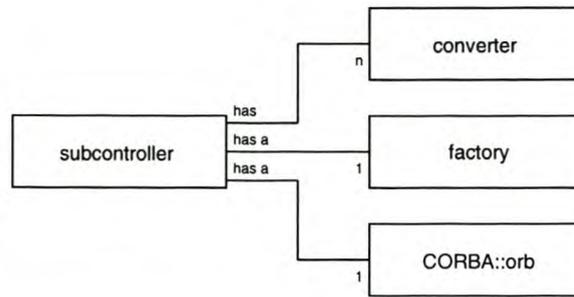


Figure 3.4: UML association diagram of the subcontroller

Figure 3.4 shows an UML association diagram from the perspective of the subcontroller object. It shows that a subcontroller can have n converter objects under its control. A subcontroller also contains one factory object to construct new converter objects and one ORB object for CORBA communications.

The subcontroller layer is also responsible for handling communications between the main application and the converter attribute system via the CORBA interface. When the subcontroller is executed its CORBA reference is registered with the CORBA naming service to allow the main application to obtain a reference to it. Communication between the main application and the subcontroller then takes place via their ORB objects.

All the converters of a system (in a particular memory space) are conceptually part of the subcontroller running in that memory space, and are only illustrated otherwise for the sake of clarity. Both the converters of the system and the subcontroller are part of the physical layer of the system. The subcontroller layer will be discussed in detail in chapter 5.

3.2.3 The CORBA interface

From figure 3.2 it is clear that the converters and subcontrollers of a particular SDR system can be distributed across various memory spaces. The main application that is in control of all these various subcontrollers therefore needs a uniform way to communicate with all the subcontrollers.

The CORBA layer provides the communication layer between the main application and all the subcontrollers of the system. CORBA is a platform- and programming-language independent remote procedure call architecture that allows a client application to invoke methods in a remote servant object as if it were local to the client.

It is important to note that the CORBA interface is only used to channel control information between the main application and the subcontroller layer. Communication between the control application and the subcontroller layer only takes place when the

parameters of the system are modified, new components are added to the system and for initiating and terminating signal processing. It is therefore an infrequent, low bandwidth operation. CORBA is not used for signal processing or for communication between the subcontroller layer and the converter layer.

CORBA was selected to handle RPC because of its high level of portability across platforms¹ and programming languages, and because it provides all the RPC functionality needed by the system. Microsoft provides a similar RPC architecture known as the *distributed component object model* (DCOM) but it is not as portable across platforms as CORBA.

Another option was to design a proprietary RPC mechanism, but this was not a feasible solution because designing a portable RPC architecture from scratch did not fit in with the timing constraints of the project.

The integration of CORBA into the SDR architecture will be discussed in chapter 6.

3.2.4 The Main Application

The main application (also called the *control application*) provides the user interface to the system. This can be a simple *text-based user interface* (TUI) or a *graphical user interface* (GUI).

TUIs are generally used in environments that do not support any graphical visual feedback, like some embedded systems or computer operating system installations that do not have graphical interfaces installed.

GUIs, on the other hand, are the most popular user interfaces due to their visual appeal and because they hide the implementation from the user. The use of pointing devices like mice and graphical tablets also allow user interaction with the application to be more intuitive than by using only the keyboard.

The main application controls all the subcontrollers in the system. It can modify subcontroller attributes and also issue commands that control the processing of samples. It can also modify converter parameters via the string-based attribute system (section 4.3.1).

A GUI was implemented using the QT library by Trolltech, and the implementation will be discussed in chapter 7.

¹ORBs are available for the Intel x86 platform running Windows, Unix and its variants, the Apple Macintosh platform running Mac OS and even for embedded platforms and operating systems.

Chapter 4

The Converter Layer

The proper design of the converter layer is probably the most important aspect of the whole system. The converter layer is responsible for all the signal processing, and therefore needs to have a highly optimised architecture. To accomplish high-speed horizontal signal flow it must feature a highly efficient horizontal transport mechanism as well as highly optimised signal processing routines. To enable easy converter attribute modification a highly flexible vertical transport mechanism is needed. In this chapter all the design requirements, constraints and issues involved in designing the converter layer architecture will be discussed.

4.1 Basic converter design

In the context of this project, a converter is an atomic unit that performs a well-defined signal processing function. In other words, a converter will receive data from a source, process this data according to some predefined algorithm and produce a result. This whole process must be highly optimised for speed in order to achieve high data throughput rates.

The algorithm will most likely have adaptable parameters to ensure efficient operation in any configuration on any platform, or to alter the operating characteristics. To achieve this, some mechanism is needed whereby the parameters of the algorithm can be dynamically adjusted to suit the current configuration. This modification process does not need to be optimised for speed because modifying the parameters of an algorithm is generally an infrequent operation, but it needs to be highly flexible to accommodate the various different types of converter parameters.

Figure 4.1 shows a basic representation of a converter. The converter has some input ports and output ports. The number of ports are determined by the developer of the converter and is dependent on the function the converter has to perform. To adjust the parameters of the signal processing algorithm a text-based set of attributes

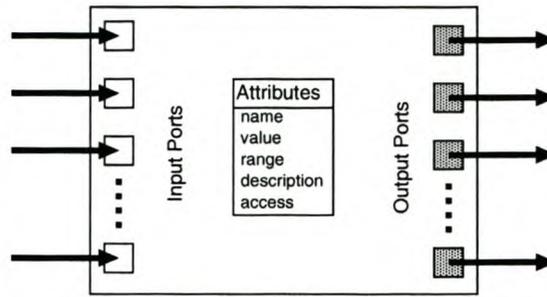


Figure 4.1: A basic converter representation

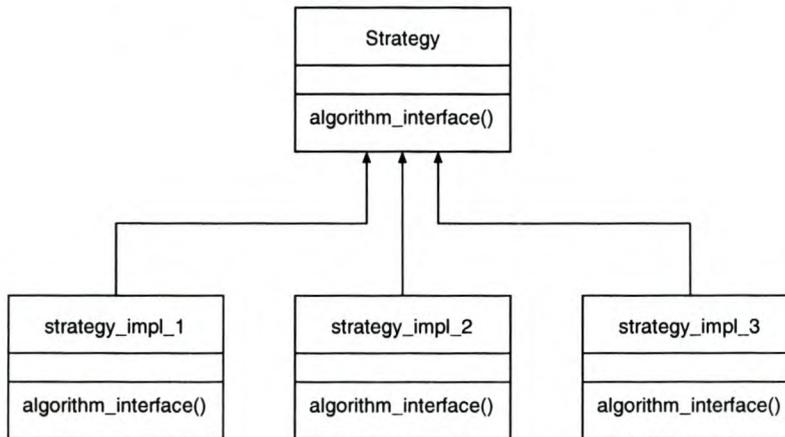


Figure 4.2: The strategy design pattern [7]

is provided. Each parameter has an attribute associated with it with fields describing the name, value, and range of values the parameter may take on.

A desirable attribute of the converters of the system is a single, uniform interface. A single interface for all converters simplifies new converter creation as well as the subcontroller design. For this reason the design of the converters is modelled after the Strategy design pattern [7]. This design pattern allows a single abstract algorithm interface to be specified from which specialised implementations are derived. A UML diagram of the Strategy design pattern is shown in figure 4.2. The abstract Strategy class provides a common interface for the various implementations that is derived from it.

The advantage of this design pattern is that various algorithm implementations can be created that can be used interchangeably in a larger system. The functionality of the system or one of its subsystems can therefore be easily modified by just using a new algorithm implementation. This property of the Strategy design pattern makes it especially useful in the context of software defined radios. A full discussion of a practical converter class is given in section 4.3.

4.2 Design considerations

This section provides a summary of the most important aspects that had to be considered during the development of the converter layer.

The first and perhaps most obvious consideration in the design of the converter layer is that it should be optimised for high-speed processing. Any data structures that form a part of a converter should therefore be created only once, and preferably when the converter itself is constructed because constructing objects incurs some processing overhead.

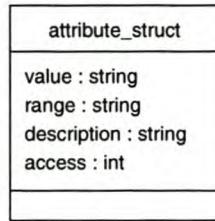
The container classes used for the buffers must be highly optimised for efficient data access. This will ensure that as little time as possible is spent on data transfer between converters. The operations these containers provide for dynamic size monitoring and control must also be as efficient as possible to ensure as little overhead is generated as possible during these operations. Refer to section 4.4 for a detailed description of input buffers.

Pointers to classes offer an attractive way to optimise the architecture. Pointers allow the use of a single instance of a class, and this eliminates expensive object creation overhead incurred when passing objects by value. The overhead associated with dereferencing the pointer is much smaller than the overhead due to construction of an object. Standard C pointers are very dangerous, though, and should be used with great care. If a pointer to an object is dereferenced after the object has been destroyed (or before it is created), a segmentation fault will occur and the operating system will immediately terminate the application.

Another problem caused by standard C pointers is memory leaks. If an object is created on the heap with the C++ keyword “new” it must also be deleted explicitly once it is no longer needed. If the pointer to the current object that was created with “new” is assigned to a new object instance before the current object is deleted, the current object will remain in memory for as long as the program is running, and the memory it occupies will not be available until the program terminates.

To minimise the dangers associated with “raw” pointers, a special kind of pointer called a *reference-counted pointer*¹ (RCPtr) was used in the system. This pointer class keeps track of the number of RCPtrs to a specific object. Only the last RCPtr instance will destroy the object to which it points, and only when the object is no longer needed. RCPtrs therefore offer a much safer way to develop applications than standard C pointers.

¹The copyright of the RCPtr Class is owned by Prof. Johan du Preez, University of Stellenbosch, and the source code for this class was released under the GNU Lesser General Public License in 2003.

Figure 4.3: *sdr_converter* attributes

4.3 Class `sdr_converter`

The strategy design pattern introduced in section 4.1 provides a means to specify a common interface in a base class for all the classes that derive from that base class. This common interface allows the various derived classes to be used in the system interchangeably.

The `sdr_converter` class is such an abstract base class. It provides the interface for all the converter implementation classes present in the system. It also provides some common protected and public methods for the derived classes that mainly deal with data flow control and that handle the manipulation of attributes.

The functionality of each converter is defined in a function called `process`. This function is defined in the classes that are derived from the `sdr_converter` base class because it is unique for every derived converter class.

The fact that the data transfer operates on a “push” principle implies that the output ports of the system do not need any buffering. When the current converter is finished with the processing of a sample (or a set of samples) the result can be directly written to the input port of the next converter. The output ports therefore only indicate where the processed samples should go. The input ports, on the other hand, need to have a variable-sized buffer because it is not possible to know how many samples the preceding converter will generate during its process cycle.

4.3.1 The attribute system

The parameters of the algorithm that is implemented in a converter will seldom have fixed, predetermined values. Examples of such parameters are filter coefficients or amplifier gain. To be able to modify these parameters, each one has an associated entry in the attribute table of the converter. The string-based attribute system allows the user of the system to directly modify the value(s) of a certain attribute of a converter. A graphical representation of the attribute structure is shown in Figure 4.3.

The attribute table is made up of a collection of these attribute structures. The STL

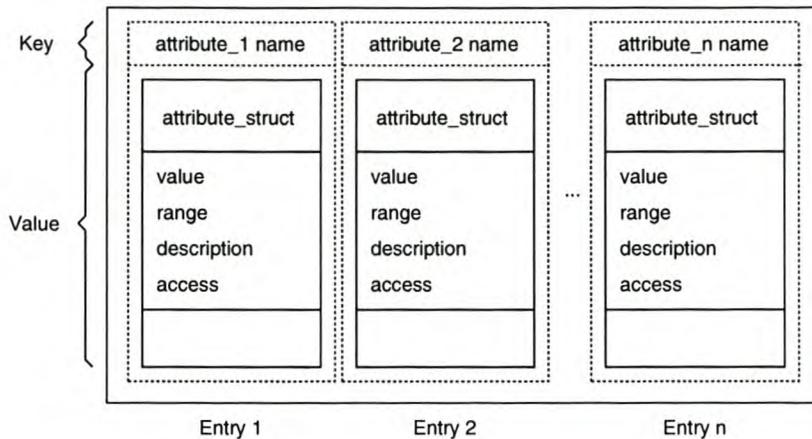


Figure 4.4: *sdr_converter* attribute table

map container is used to store the various attributes the converter may have. The map container is a sorted associative container and the association between the fields is unique [23]. This means that no two elements in the map have the same key or identifier and therefore each element can be uniquely identified. This ensures that the attributes of a converter are unique. The structure of the attribute table is shown in figure 4.4.

The idea behind the attribute system is to provide a string-based high-level application interface to the member variables of the converters of the system. The string type was chosen because it allows the attribute interface to be more intuitive for the person using the system. Attributes therefore represent the member variables in the `sdr_converter` derived classes, for example if an oscillator has a member variable “freq” that determines its oscillating frequency, a corresponding attribute named “freq” must be added. The user is then able to change the frequency of the oscillator module directly via the attribute system.

The methods in class `sdr_converter` that are used to manipulate the attributes of the converter are summarised in table 4.1.

Attribute descriptions are added to the class during construction of the converter object. The name of the attribute, a default value, the range of possible values and short description must be specified when the attribute is added to the attribute table with the `add_attribute` method. By default, attribute access is read-write but it can be specified explicitly with this method as being read-only, write-only or read-write. The `add_attribute` method is protected to prevent external classes from adding attributes to the current converter.

To obtain the current value(s) of a specific attribute the `get_attribute` method is used. All the fields of an attribute can be obtained with this method because it returns

Name	Parameters	Return Type	Description
add_attribute [protected]	string attribute_name string attribute_value string attribute_range string attribute_description int access (SDR_READ_WRITE)	void	This function is used to add attributes to the attribute table of the converter
get_attribute	string attribute_name	const attribute_struct&	This function is used to obtain the value of an attribute
set_attribute	string attribute_name string attribute_value	void	This function is used to modify the value of an attribute
apply [protected]	string attribute_name string attribute_value	int	This function is used by set_attribute to change the values of the member variables. It must be defined in the class implementation
properties	-	std::string	This function is used to obtain a list of the attributes of a converter class.

Table 4.1: *Methods available for manipulating attributes*

a reference to the attribute struct of the particular attribute.

The `set_attribute` method is used to modify attributes at runtime. Both the attribute name and its desired value must be specified as strings. After it has verified that the attribute does indeed exist it calls the `apply` method of the converter.

The `apply` method is declared as a pure virtual function in the base class and therefore it has to be implemented in all the derived classes. The `apply` function provides a way to translate the value of the attribute in string format to the format of the member variable that is to be modified. For example, the “freq” variable above may be a 32-bit floating point numerical type. The `apply` method is responsible for converting the string in the `attribute_value` parameter to a 32-bit floating point number and then assigning it to the “freq” variable. The `apply` method can also be used to perform various tasks within a converter. As an example, if the converter performs soundcard input and output, `apply` may be used to modify the state of the converter from, for example, “playback” to “record” or vice versa.

The `properties` method is provided to retrieve a list of the attributes of a certain converter, to enable the main application to discover the characteristics of the various components in the system. The method returns a single newline-delimited string that contains the names of all the attributes of the converter.

4.3.2 Data input and output in `sdr_converter`

Implementing a push-based signal flow model requires the use of buffered input ports. This enables the data to be stored in the buffer until the converter can process the data. Without buffers the samples would have to be consumed as they arrive due to the lack of storage space. The input “ports” of the converters are therefore just data containers that store the data until it can be processed. The buffer classes are discussed in section 4.4.

The fact that the system operates on a push principle means that the output ports do not need to be buffered. In fact the output ports only need to indicate where the data must be delivered to when the current converter has finished processing it. A UML representation of the output ports of the converters is shown in figure 4.5. It shows that an output port is merely a combination of a pointer and an integer. The pointer points to the converter that the processed samples must be delivered to and the integer specifies the input port this output is connected to.

Input- and output ports have to be added to a converter when it is constructed. To add input and output ports a method called `add_ports` is used that specifies the data type of each port. The created buffers are unbounded, and monitoring the growth of the input buffers should be done in the subcontroller. The input- and output port structures are declared private to prevent direct manipulation by derived and external

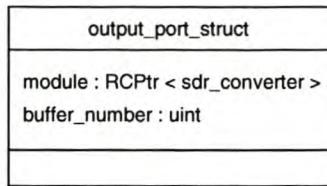


Figure 4.5: *The structure of the output ports*

classes and therefore the derived classes are not allowed to access the contents of the ports directly. The method is protected to ensure that external objects cannot add ports to a converter.

Sometimes it is useful to know the number of input- and output ports of a specific converter. The subcontroller may, for example, want to monitor all the input ports of a converter to control the size of the input buffers. The `num_ports` function can be used to obtain the number of input or output ports.

When the current converter is finished with processing its samples it has to transfer the samples to the input port of the next converter. It cannot directly access the input port of the next converter because it is a private member. A publicly accessible method is therefore needed to write the samples to the input buffer of the next converter. The `write_to_buffer` method is provided for this purpose. It accepts an integer and a sample as parameters. The integer is used to identify the input buffer number that the sample `T` must be inserted into. The `write_output_port` method is provided to write processed samples to the input port of the next converter. It does this by calling the `write_to_buffer` method of the converter that `module` is connected to using the value of `buffer_number` as the `port_number` parameter. The `write_output_port` method is protected to ensure that the current converter has exclusive control over its output ports.

Reading samples from the input buffer is done in two steps. Firstly a reference to the desired buffer must be obtained. This reference must then be used to retrieve the sample from the buffer. The `read_input_port` method is provided to obtain a reference to a specific input buffer. Refer to section 4.4 about class `sdr_buffer` for more information on retrieving data from a buffer.

To ensure that processing will only take place if the buffer contains data the size of the buffer can be monitored. The `ready` method is used to check whether the module is ready to begin processing samples and it returns “true” if the input buffer has a non-zero size. This method is provided because it is more efficient than checking if the size of the buffer is greater than 0 (zero). The `ready` method uses the `empty` method of the input buffer class which is discussed in section 4.4.3.

To enable the subcontroller to manage the size of the input buffers a way is needed

Table 4.2: *Methods available for manipulating the ports*

Name	Parameters	Return Type	Description
<code>add_port</code>	<code>int port_type</code>	<code>int</code>	This function is used to add ports to the converter.
<code>num_ports</code>	<code>int port_type</code>	<code>int</code>	This function returns the number of input or output ports.
<code>write_to_buffer</code> [Templated]	<code>uint port_number</code> <code>T sample</code>	<code>void</code>	This function is used to write the sample <code>T</code> to the buffer of input port <code>port_number</code> .
<code>read_input_port</code>	<code>uint port_number</code>	<code>RCPtr<sdr_buffer_base></code>	Returns a pointer to the input buffer <code>port_number</code> .
<code>ready</code>	<code>uint port_number</code>	<code>bool</code>	Checks if there are any samples in input buffer <code>port_number</code> .
<code>buffer_size</code>	<code>uint port_number</code>	<code>int</code>	Returns the size of the input buffer <code>port_number</code> .
<code>next_buffer_size</code>	<code>uint port_number</code>	<code>int</code>	Returns the size of the input buffer that output port <code>port_number</code> is connected to.
<code>write_output_port</code> [Templated]	<code>uint port_number</code> <code>T sample</code>	<code>void</code>	write the sample <code>T</code> to the output port <code>port_number</code> .
<code>link</code>	<code>uint output_port_number</code> <code>RCPtr<sdr_converter> destination_module</code> <code>uint destination_portnumber</code>	<code>void</code>	Links the output port <code>output_port_number</code> to the input port <code>destination_portnumber</code> of the converter module <code>destination_module</code> .

to determine the number of elements in a buffer. The `buffer_size` method returns the number of elements in the input buffer specified with the `port_number` parameter.

Connecting an output port to an input port is done with the `link` method. “Linking” output- and input ports is achieved by updating the `RCPtr` of an output port to point to the next converter and modifying the value of the associated `buffer_number` to hold the input port number that the output is associated with. If an output port is not connected to an input port the `write_output_port` method silently discards the sample. This enables the system to still operate properly in case it has converters that have output ports that are not connected to other converters or external devices.

A summary of the methods that are available in `sdr_converter` to manipulate the input and output ports of a converter is given in table 4.2.

4.3.3 Miscellaneous methods available in class `sdr_converter`

In this section all the remaining methods available in `sdr_converter` that do not fit into the above categories are discussed.

Table 4.3: *Miscellaneous methods available in sdr_converter*

Name	Parameters	Return Type	Description
process	-	void	This method defines the functionality of the converter and must be implemented in all derived classes.
is_a	-	const char*	This method is used to identify the converter at run time. It must be implemented in all derived classes.

Every converter in the system performs a predefined signal processing task on the samples it receives. Some converters may also perform data transfer operations between the SDR system and peripheral devices as part of its signal processing cycle. The `sdr_converter` base class contains a pure virtual method called `process` that must contain the signal processing algorithm that defines the functionality of the converter. To ensure efficient operation the developer must ensure that the algorithm is implemented as efficiently as possible with as little unnecessary overhead as possible. Any objects that are needed have to be created outside the `process` method because the construction of objects generally incur considerable overhead. The `process` method is defined as an inline function in the `sdr_converter` base class. This avoids call overhead attributed to jumping to the correct location to execute the method, and increases the efficiency of the system at the expense of an increase in object code size.

The single abstract interface provided by the `sdr_converter` class ensures that all converters share the same interface. It also allows the system to use the base class representation of the interface. This makes it necessary to be able to identify a method at run time to ensure that the correct class is used. The `is_a` method was defined to identify the converter type. The method takes no arguments and returns the class name as a C string.

A summary of these methods is given in table 4.3. All the methods are public because they will be invoked by external classes.

4.3.4 Exceptions in `sdr_converter`

When an error occurs in a program or a function, whether because of an unanticipated hardware error or a software error, that error condition must be reported to the calling function. One way to indicate erroneous conditions in C and C++ is to return special error codes. The problem with this approach is that a return value can be ignored, and therefore errors can occur undetected.

C++ therefore also offers the concept of *exceptions*. An *exceptional condition* is any

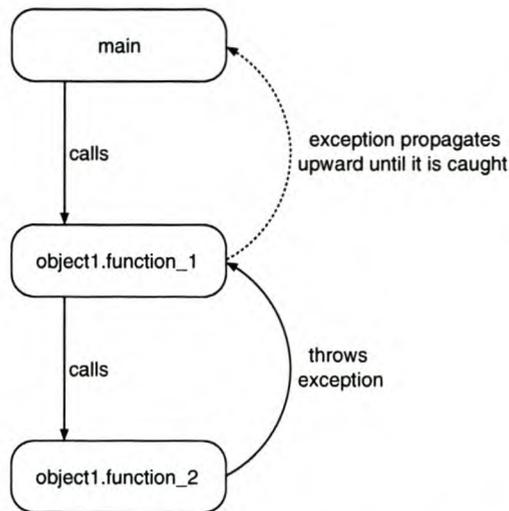


Figure 4.6: *Exception mechanism*

condition that occurs that cannot be handled in the current context of the program, whether it is an error or not. The function that encounters the exceptional condition *throws* an exception and then exits. This exception must be *caught* in a higher context and appropriate action has to be taken to rectify the situation.

Figure 4.6 shows how an exception is thrown from a function and how it propagates through the system. The main method calls object1’s function1 method, which in turn calls object1’s function2 method (this can be a method from another object as well). If function2 encounters an exceptional condition, it throws an exception to indicate it. Function1 now has the option to either catch the exception or let it propagate up to the main function. If function1 does not catch and handle the exception it exits immediately, returning control to the main function. If the exception is not caught in main, the `std::terminate` function is called and the application exits. This behaviour of the exception mechanism therefore ensures that any erroneous or exceptional conditions are dealt with.

In the `sdr_converter` class all errors have to be dealt with via the exception mechanism. This will ensure that all errors that are encountered are rectified. A special exception class called `sdr_base_exception` was defined for this purpose. Several specialised exception classes that are derived from this base class are also provided for specific exceptional cases. These classes are listed in table 4.4.

The exception base class defines two member variables that are used to identify the exception: an integer that is used to indicate the type of exception and a C string that is used to provide a short description of the exception. The derived classes then have to assign values to these variables according to the nature of the exceptional condition. For example, when the sound card encounters an error the `sdr_snd_exception` may use

Table 4.4: *Specialised exception classes*

Name	Purpose
<code>sdr_port.exception</code>	Used to indicate errors on converter ports
<code>sdr_snd.exception</code>	Used to indicate sound card errors
<code>sdr_network.exception</code>	Used to indicate network-related errors
<code>sdr_attribute.exception</code>	Used to indicate errors while using attributes
<code>sdr_buffer.exception</code>	Used to indicate buffer errors
<code>sdr_corba.exception</code>	Used to indicate errors with the RPC mechanism
<code>sdr_subcontroller.exception</code>	Used to indicate errors in the subcontroller
<code>sdr_daq_2010.exception</code>	Used to indicate data acquisition card errors

the value in the global error variable `errno` to indicate the type of exception. It must also provide a short textual description of the error. This is generally used to indicate from where in a function the exception was thrown.

4.4 Input buffers: class `sdr_buffer_base`

The input buffers used in `sdr_converter` is totally independent from the `sdr_converter` class. This was done to ensure that changes to one of these classes do not influence the other class. It also simplifies data abstraction because the `sdr_converter` class does not need to know what data types are supported by the buffer class. This section covers the design and implementation of a buffer class for the `sdr_converter` class.

4.4.1 Data abstraction

A modular interface specification requires that the data transferred across that interface be of a known type. This ensures that any implementation using the interface will be able to interact with any other implementation using the interface. Limiting the interface to a single data type is not desirable though, because the optimal data type for a specific converter cannot be determined in the base class. Some way to abstract the data types used in the converters is thus needed. This abstraction is done by the input buffer class.

The input buffers used in class `sdr_converter` achieve abstraction of the input data by providing methods that allow samples to be inserted into the buffer regardless of their type. The methods provided by the buffer for data insertion and removal allows the converter producing data to be unaware of the data type that the converter consuming the data uses. A filter converter may, for example, prefer to use the 32-bit floating point data type. Another converter may for some reason prefer to use integers

to represent its data. The abstraction of data in the buffers allows the filter to write floating-point values into the input buffer of the other converter, which will extract the data from the buffer in integer format before it is processed.

4.4.2 Choosing a container for the buffers

It was mentioned in section 3.2.1 that the data flow of the system operates on a “push” principle. This requires *first-in, first-out* (FIFO) buffers for the input ports in the converter class. These buffers have to be dynamically resizable because the current converter can not know beforehand how many samples the converter preceding it will generate during execution of its process method. The variable size of the buffers also allows stable operation under variable load conditions (in a multi-threaded subcontroller implementation) and enables samples to be processed in batches.

To achieve this goal the STL *double-ended queue* (deque) container type was used for the input buffers. The deque class is a templated class that mimics the behaviour of the STL vector class. The main difference between vector and deque is that the deque class supports insertion and removal of data from both the front and the back of the container whereas a vector only supports insertion and removal of data from the back of the container. The deque container class can therefore be used to implement a FIFO queue.

The deque container was chosen because the STL containers are highly optimised containers and therefore offer very efficient manipulation of data.

4.4.3 Class `sdr_buffer_base`

A UML diagram that shows the methods that are available in the `sdr_buffer_base` class is shown in Figure 4.7. The diagram shows that the `sdr_buffer_base` class has no data members—the data member that holds the data in the buffer has to be declared in the class derived from `sdr_buffer_base`.

The diagram also shows that all the member functions of the class are public. This is because the buffer does not process the signal—it only acts as a temporary storage place for data, and the classes that use the buffer must be able to access all its functions.

The `sdr_buffer_base` class is an abstract base class, and only provides the interface for buffer implementations. The interface defines a few overloaded `write` methods for inserting data into a buffer. A list of these methods is given in table 4.5.

As already mentioned, the class that produces data need not be aware of the data type that the class that consumes the data is built on. Therefore the producer can call any of the `write` methods on any buffer; the data is converted to the correct type inside the write methods.

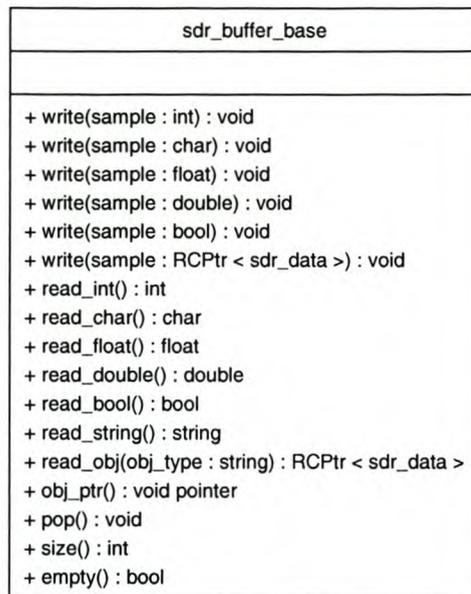


Figure 4.7: A UML Class Diagram Of The *sdr_buffer_base* Class

Overloading the write method for various data types provides a single data insertion method to the class to keep the interface straightforward. A single write method also provides a means to achieve data abstraction, because the data inserted into the buffer is not bound to a single type.

Table 4.5: Methods available for inserting data into a buffer

Name	Parameters	Return Type	Description
write	int sample	void	Insert an int into the buffer.
write	char sample	void	Insert a char into the buffer.
write	float	void	Insert a float into the buffer.
write	double	void	Insert a double into the buffer.
write	bool	void	Insert a bool into the buffer.
write	RCPtr < sdr_data >	void	Insert a sdr_data object into the buffer.

To retrieve data from the buffer several read methods are provided. Due to the fact that C++ methods cannot be overloaded only on return types the methods used to read from the buffers must each have a unique name. The naming scheme used was chosen to be as intuitive as possible, and the various methods are listed in table 4.6. The methods return the data from the front slot of the buffer, converted to the type associated with that method. Note that reading from a buffer also removes the front element from the buffer.

The `read_string` method returns the element in the candidate consumption slot in a STL string. This is provided to convert numerical types to string format and is defined

Table 4.6: *Methods available for retrieving data from a buffer*

Name	Parameters	Return Type	Description
read_int	-	int	Return an int from the buffer.
read_char	-	char	Return a char from the buffer.
read_float	-	float	Return a float from the buffer.
read_double	-	double	Return a double from the buffer.
read_bool	-	bool	Return a bool from the buffer.
read_string	-	std::string	Return a STL string from the buffer.
read_obj	-	RCPtr < sdr_data >	Return a sdr_data object from the buffer.

for convenience.

Sometimes it may be useful to be able to retrieve a value from the buffer without removing it as well, because it may not be feasible to create temporary storage for the sample if it has to be re-used. The `obj_ptr` method is used to obtain a pointer of type `void*` to the candidate consumption slot of the buffer. This pointer can be used to cast the data in the candidate consumption slot to any other type that can be cast to or from a `void*`. When the value in the candidate consumption slot is referenced in this way, it is *not* removed from the buffer.

After the `obj_ptr` method was used to reference the sample in the candidate consumption slot the sample will still remain in the buffer. Removing it with one of the read methods may not be feasible in all cases. A method called `pop` is provided to remove samples from the buffer that are no longer needed. When this method is called, the data is removed from the buffer and discarded. This method may also be called when samples need to be discarded, for example when sub-sampling or decimation is performed.

In order to monitor the size of the input buffers an aptly named `size` method is defined. This method returns the number of samples currently in the queue and will typically be used for flow control purposes by the subcontroller.

If an attempt is made to retrieve a value from an empty input buffer the buffer returns 0 (zero). This is a property of the deque and may or may not be a legal value for the current converter. The `empty` method was defined in order to check if a buffer contains any elements. One way to check if a buffer contains data is to count the elements and testing if the result is greater than 0 (zero). The `empty` method defined in the STL deque tests the iterator pointing to the front of the deque and the iterator pointing to the back of the deque for equality and returns the result. This approach is more efficient than comparing the size of the buffer with zero because it does not count the number of elements in the deque.

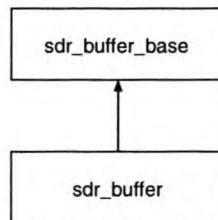
Table 4.7: *Miscellaneous methods available in the buffer class*

Name	Parameters	Return Type	Description
obj_ptr	-	void*	Returns an untyped pointer to the value in the candidate consumption slot.
pop	-	void	Removes and discards the data in the candidate consumption slot.
size	-	int	Returns the number of elements in the buffer.
empty	-	bool	Returns “true” if there are no elements in the buffer.

4.4.4 A practical buffer class

Unlike the `sdr_converter` base class the `sdr_buffer_base` base class only provides declarations of the methods it provides and no definitions. This is because none of the methods can be implemented in a universal way—input and output of various data types may differ significantly. The `sdr_buffer_base` class therefore only provides the interface to the buffer class and this ensures that more than one buffer class may be implemented in the system.

An implementation of the `sdr_buffer_base` class called `sdr_buffer` was created that can be used in the converter classes. An inheritance diagram of the `sdr_buffer` class is shown in figure 4.8. Figure 4.9 shows a detailed UML class diagram of the `sdr_buffer` class. The `sdr_converter` class method `add_port` is used in the constructor of each converter to create new instances of the `sdr_buffer` class. Each instance of the `sdr_buffer` class forms one input port of a converter.

**Figure 4.8:** *Class `sdr_buffer` inheritance diagram*

The `sdr_buffer` class is a templated class and the template is used to determine the data type that the buffer will use. Conversion between the type of the buffer and the types that can be retrieved from the buffer is done with a simple casting operation known as `static_cast`. The `static_cast` keyword allows a variable of a certain type to be treated as another type and it is used for conversion between related types [24]. This means that the `sdr_buffer` class is only compatible with intrinsic numerical types like `int`, `float` or `double`. A special mechanism is provided for more complex proprietary data types, and is discussed in sections 4.4.5 and 4.4.6.

sdr_buffer
- buffer : std::deque < T > - result : T
+ write(sample : int) : void + write(sample : char) : void + write(sample : float) : void + write(sample : double) : void + write(sample : bool) : void + write(sample : RCPtr < sdr_data >) : void + read_int() : int + read_char() : char + read_float() : float + read_double() : double + read_bool() : bool + read_string() : string + read_obj(obj_type : string) : RCPtr < sdr_data > + obj_ptr() : void pointer + pop() : void + size() : int + empty() : bool

Figure 4.9: *Class sdr_buffer*

4.4.5 Proprietary data types

The intrinsic data types provided in the C++ language may not be sufficient to represent all types of data that may be processed by a software defined radio system. It may, for example, be more efficient and logical to represent an image as a matrix of values that represent the pixels, as opposed to a single stream of values. To accomplish this a proprietary class is needed that can be used to store more complex data representations.

A generic base class was defined for proprietary data types. The `sdr_data` class is defined in the `sdr.base_classes` namespace and has only one pure virtual method, `is_a`, that is used to identify its type. All other methods that the class supports have to be declared in the class itself.

The `sdr_data` class is provided so that more complex data structures can be supported in the input buffers. This is done via the `sdr_buffer_data` buffer implementation, discussed in section 4.4.6.

4.4.6 A practical buffer class using the `sdr_data` type

It was mentioned in section 4.4.4 that the `sdr_buffer` class only supports intrinsic data types due to the way type conversion is performed in the class. To use buffers with the generic `sdr_data` class, a special buffer implementation called `sdr_buffer_data` was created. It allows the SDR system to use more complex data types than the intrinsic scalar data types provided by the C++ language.

Table 4.8: *Methods available in the `sdr_buffer_data` class*

Name	Parameters	Return Type	Description
<code>write</code>	<code>RCPtr < sdr_data ></code>	<code>void</code>	Inserts a <code>sdr_data</code> object into the buffer.
<code>read_obj</code>	<code>const char * obj_type</code>	<code>RCPtr < sdr_data ></code>	Returns an <code>sdr_data</code> object from the candidate consumption slot.
<code>obj_ptr</code>	-	<code>void *</code>	Returns an untyped pointer to the data in the candidate consumption slot.
<code>pop()</code>	-	<code>void</code>	Removes and discards the data in the candidate consumption slot.
<code>size()</code>	-	<code>uint</code>	Returns the number of elements in the buffer.
<code>empty()</code>	-	<code>bool</code>	Returns “true” if there are no elements in the buffer.

To insert a `sdr_data` object into the buffer a single `write` method is provided. It inserts a `RCPtr` to `sdr_data` into the input buffer. To retrieve the `RCPtr` to the object from the buffer the `read_obj` method has to be used. The type of object must be specified with the `read_obj` method to ensure that the correct object type is retrieved from the buffer. The other read member functions (as defined in `sdr_buffer_base`) have no meaning in the `sdr_buffer_data` class because it does not enforce the extraction or insertion of single values from a more complex type derived from `sdr_data`. In future versions of `sdr_data` a generic base interface may be defined to ensure that singular values can be retrieved from a `sdr_data` descendant if it is needed.

An `obj_ptr` method is provided that provides an untyped pointer to the contents of the buffer. In the current context this method may not provide any advantage over the `read_obj` method, because it will only make sense if it is cast back to its original type. It does however return an untyped pointer to the data in the candidate consumption slot because returning an untyped null pointer can have devastating effects on the program if that pointer is dereferenced. This function must be used with extreme caution.

The `pop`, `size` and `empty` member functions have the same functionality as in the `sdr_buffer` class.

4.5 An ethernet interface converter implementation

Signal processing tasks that require high levels of computing power are often distributed over various processors on different platforms. These systems are mostly interconnected with an ethernet network because of its availability and speed. Another advantage of such networked systems is the ability to carry out specialised signal processing tasks on dedicated processors.

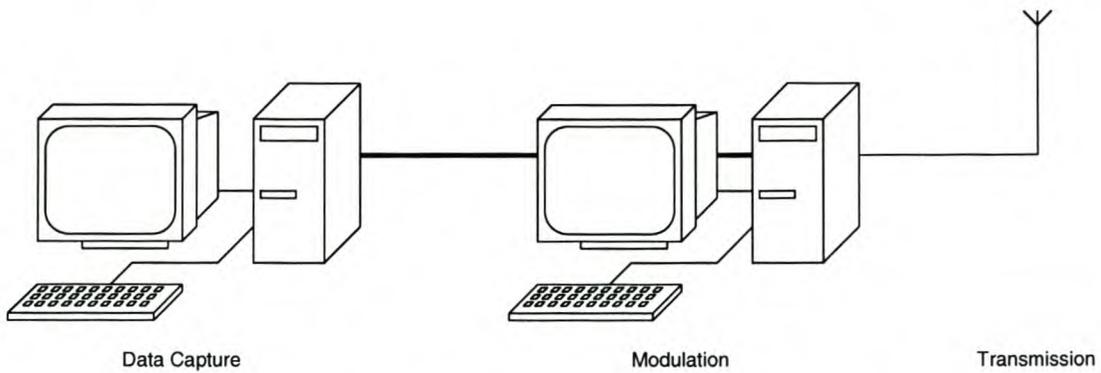


Figure 4.10: An example of a distributed radio system

Consider video broadcasting as an example. The user may use a personal computer to capture speech and music with the sound card and video with a special video capture card. He can then mix the signals to form a single data stream. This data stream can now be transferred across an ethernet interface to a computer that performs a dedicated UHF modulation function. From this computer the signal can then be transmitted. Refer to figure 4.10 for a graphical representation of such a system.

To enable the use of the software radio system in such distributed systems two special converter implementations were created. The first is used to send data packets over an ethernet interface and the second is used to receive the packets sent over the network. The TCP protocol is used in the system because it provides a reliable, connection-oriented transport mechanism. Both classes are templated and the data type used in instances of these classes is determined by the type used to instantiate objects of the classes.

Sending data

A special converter class called `sdr_network_out_tcp` was created for sending data across an ethernet interface. The converter needs to know the machine to which it must send the data as well as the port number that must be used for the connection. These parameters are not initialised in the constructor, but have to be specified before a connection can be established. This has to be done via the attribute mechanism; the class has a `hostname` and a `port` attribute to accomplish this. The machine can be identified by its canonical host name or by its IP address. The port value can be any integer in the range 1024 to 65535, but the range 49152 to 65535 is recommended because no port assignments are made by the *internet assigned numbers authority* (IANA) in this range. The constructor *does not* initiate a connection because at the time the object is instantiated there is no way to know if the receiver object is listening for a connection.

* An attribute named `connect` is defined in the constructor and is used to initiate a

connection. It firstly determines the IP address of the target host if the host name was supplied. It then tries to connect to the remote host. If no receiver objects are listening on the remote host, the `connect` system call will fail and an exception will be generated.

If the `connect` system call was successful, data transfer may take place. The process function of the converter checks if the input buffer is ready and if it is, the first value is retrieved from the buffer. If the connection is still active this value is sent across the network to the remote host. If the connection is broken the sample is discarded to avoid unbounded buffer growth. If the converter receives data on its input port before a connection is established, it also discards the data in the input port during its process cycle.

Receiving data

A separate class called `sdr_network_in_tcp` was created for receiving data from an ethernet interface. The port that the converter must listen on have to be specified with the `port` attribute. The constructor of this class does not initialise any connection parameters or initiate a connection. This is because the `accept` call blocks until an attempt to connect to the socket is made by a client.

An attribute called `init` provides all the socket setup routines that are necessary to initialise a network socket. First of all it opens a new TCP socket. This socket must then be bound to the port specified with the `port` attribute and to the ethernet interface. This is achieved with the `bind` system call. If the `bind` operation succeeded the converter has to open the port and it must start listening for connections to the port. For this the `listen` system call must be used.

When a “client” tries to connect to the port the connection has to be accepted by the converter with the `accept` system call. A state attribute called `connect` is defined in the constructor. This attribute is used to accept incoming TCP connections from the `sdr_network_out_tcp` class. If a connection is accepted the transmission of data between the two converters may commence. The converter simply writes the data it receives to the input port of the next converter during its process cycle.

Creating a network link between two converters

The link between two ethernet converters has to be managed by the subcontroller. If a distributed radio is needed the user must add a network output converter to the radio on the source computer and a network input converter to the radio on the destination computer. The host name of the destination computer as well as the port to be used for the connection has to be specified on the source computer with the `hostname` and `port` attributes respectively. Due to the fact that the connections are not established

when the converters are constructed the subcontroller has to set the `connect` attributes of both these converters to “true” to establish a connection. Once a connection is established the data will be transferred across the network interface when the radio starts to operate.

4.6 Conclusion

In this chapter the development of the converter layer was discussed. The requirements of the system were identified and a suitable converter class was created from these functional specifications. A need for input buffers was also identified and this led to the development of a base buffer class as well as two buffer class implementations.

The converter layer is responsible for all signal processing but it does not possess any signal management capabilities. It cannot control the flow of information between the various converters and it has no buffer size management capabilities. The converter layer is also not capable of management of remote procedure calls required in a distributed system. These functions of the system are the responsibility of the subcontroller layer which will be discussed next.

Chapter 5

The Subcontroller

The subcontroller has an important role to play in the software defined radio system: it handles all physical layer component processing, and it is responsible for handling the CORBA communication between the top-level application and the physical layer components. In this chapter the design of the subcontroller layer will be discussed in detail.

5.1 Subcontroller overview

The subcontroller has some very important roles to play in the system. Its first and perhaps most important function is managing the components under its control. This includes execution of each component's process method, monitoring input buffer growth of each component and managing execution scheduling. It is important to note that the subcontroller is the process that controls the signal processing and that every subcontroller is an independent executable.

The subcontroller layer also allows the main application to manage the components in the subcontroller via the CORBA interface. It does this by providing wrapper

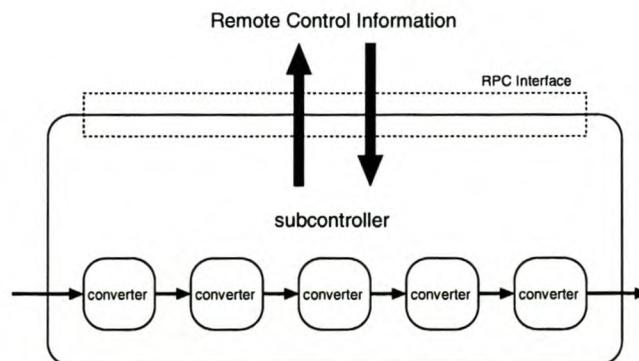


Figure 5.1: *A conceptual subcontroller representation*

functions for the attribute manipulation methods available in the `sdr_converter` class.

A conceptual representation of the subcontroller is given in figure 5.1. It shows that the subcontroller contains a few converter objects that are responsible for the signal processing algorithms. The communication with the main application is achieved via the RPC interface. The interface implementation is discussed in chapter 6.

5.1.1 Component management

To effectively manage the components that form part of the subcontroller, a container is needed to store the components or references to the components. A container class is used because it enables easy addressing of its contents through its iterator interface. It is feasible to place all the converters in one container due to the fact that all converters are derived from a common base class. The subcontroller therefore does not need to know the types of the converters present in the system because it only interacts with the `sdr_converter` base class. The C++ run-time type identification mechanism ensures that the member functions of the derived classes will be used instead of the base class methods.

The STL vector is one container option. The advantage of using the STL vector container is that it is very easy to address the elements in the vector through the “[]” operator. Using the STL vector container is not a viable option though because when an element is removed from the vector all iterators pointing to elements following the element removed that was removed are invalidated. This is not desirable as it complicates component management.

A better option is the STL map container introduced in chapter 4. The STL map container is used to provide a mapping between an integer Key (identifier) value and a RCPtr to a converter implementation. The integer Key value is obtained when a component is added to the system and the RCPtr is obtained from the `sdr_factory` class discussed in section 5.2.

This scheme was implemented because it allows the system to identify a component using an integer *unique identifier* (UID) value. The STL map container also has the important property that deleting an element from the map does not invalidate any iterators, except of course the iterator pointing to the deleted element. Therefore, when

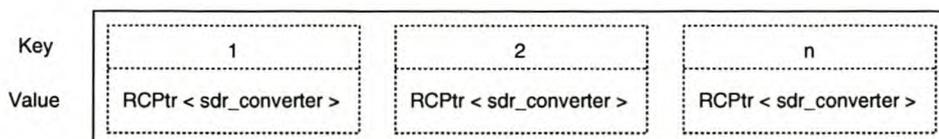


Figure 5.2: Structure of the component map container

a component is removed from the system all the other component UIDs stay valid.

5.1.2 Scheduling execution

Execution scheduling can be implemented in various ways. The first and simplest way is to just let the process methods of the components execute sequentially. This approach is very straightforward, but it does not provide any sequence-based scheduling. It produces correct results because of the fact that a converter will only process samples if there is data in its input buffer.

To better illustrate this point, consider figure 5.3. Assume that the numbers represent the sequence the converters were added to the system. With a round robin scheduling system this will also represent the sequence in which the process methods of the converters will be called. Assume that data is available on the input of converter 3 when processing commences. The process function of converter 1 will be called first, but because there is no data in its input buffer its process method will exit without writing data to the input of converter 4. The process method of converter 2 is called next, but it also exits immediately. When the process method of converter 3 is called, it processes the data in its input buffer and outputs the results to the inputs of converters 2 and 5. The process method of converter 4 is called next but it also exits immediately because converter 1 did not produce any data. Converter 5 has data in its input buffer because the process method of converter 3 produced output in its first cycle. All 5 converters have now had a chance to process any data on its input ports and the cycle starts from converter 1 again.

Converter 1 still has no data available on its input port and therefore its process method exits immediately. Converter 2 has data in its input buffer and that data is processed and output to converter 1. Converter 3 will again process the data in its input buffer and output the data to converters 2 and 5. Converter 4 now has data on one of its input ports (the one connected to converter 5). Whether or not its process method will execute is entirely dependent upon its implementation. Lastly, the process method of converter 5 is called and it processes the samples in its input buffer and

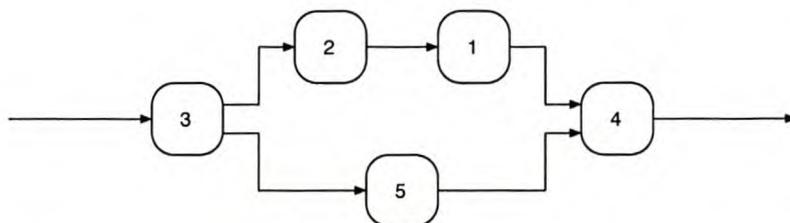


Figure 5.3: Round robin scheduling processing sequence example

outputs the result to converter 4. After the second round of processing all converters in this system will have data to process and the system will continue to process samples until it is stopped or the input to converter 3 is disconnected.

A shortcoming of the above scheme is that other CPU-intensive processes on a general-purpose platform can have an adverse effect on the execution time of a converter's process method and therefore increase the latency of the system. Another approach is to build a form of real-time scheduling into the subcontroller. This enables every component to be assigned a fixed amount of time in which to execute its process method. The subcontroller can then decide what action to take if an execution deadline is missed. This approach will enable the subcontroller to compensate for variable execution times on a general-purpose processor due to interrupts or high system loads. A more detailed discussion on the subject can be found in [2].

In its current form the subcontroller implements the first method as its scheduling mechanism. This approach was chosen because it is simple to implement and it produces acceptable results. It is also possible to implement a simple buffer size limiting mechanism to prevent buffer sizes from growing without bounds. Due to the above-mentioned effect of high processor usage on the system latency, this approach is best suited to machines with relatively light loading.¹

5.2 Adding components: the SDR factory class

One important aspect of a software radio system is that it should be as easy as possible to add new components to the system to enhance the functionality of the system. Without this feature a software radio presents very few benefits over a "hardware" radio. The ideal is to be able to add new components while the system is running, or at least without requiring a full software update.

On most systems it is also not desirable to recompile the whole radio to add new functionality to the system, because this can take a considerable amount of time. Compiling source code is also highly impractical on most platforms, especially on handheld devices. On these platforms adding new functionality to the system should be as easy as downloading an extra library.

To pave the way to achieve these goals, the `sdr_factory` class was created. This class is based on the Abstract Factory and Singleton design patterns [7]. The idea of the `sdr_factory` class is to provide an interface for creating instances of classes that are derived from an abstract base class.

¹Light loading means that no other processes are running that require fairly high amounts of CPU time.

5.2.1 The benefits of using the factory model

In its present form the `sdr_subcontroller` class is statically linked to the SDR component library. The `sdr_factory` class must be made aware of any new components that are available to the system, which means that the `sdr_factory` class must be recompiled before a new component can be used in the system. The subcontroller also has to be relinked to the new `sdr_factory` object code before the new components can be used.

A better way to implement this scheme would be to let the subcontroller *load* the `sdr_factory` library at run time instead of linking to it statically or dynamically. This approach requires that only the `sdr_factory` class be recompiled when a new object is added. To further enhance the system, a scheme can be implemented that dynamically loads the *converter* classes into the `sdr_factory` class at run time, without the need to recompile or relink anything. To add a component would then only require adding the compiled object code to the system.

5.2.2 Using the factory

A graphical representation of the process of creating a component with the `sdr_factory` is shown in figure 5.4.

The subcontroller makes a request to the factory to create a new `sdr_basic_amp`, which is a descendant of `sdr_converter`. The factory creates an object on the heap with the C++ keyword “new” and a reference to the object is obtained. The factory then returns this reference to the subcontroller wrapped in a `RCPtr` object. This `RCPtr` is then put in a STL map container that contains references to all the components in that subcontroller.

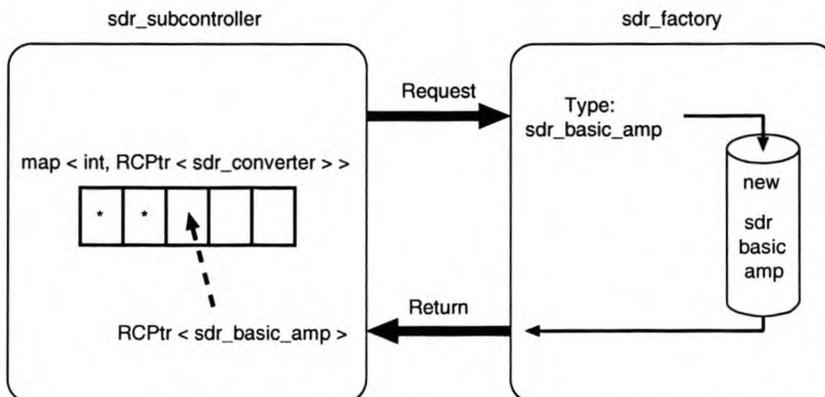


Figure 5.4: Creating a component with the factory

5.3 The subcontroller interface

A software defined radio system will in most cases be of little use if it does not provide a user interface to allow a human operator to interact with the radio. Due to the fact that several hardware platforms may be used in one radio, with a subcontroller for each platform, the subcontroller layer was separated from the user interface layer. Therefore there has to be a well-defined interface between the subcontroller layer and the user interface layer. This interface will be discussed in this section.

5.3.1 Subcontroller attributes

The parameters of a subcontroller are also controlled with a text-based attribute system. An `add_attribute` method similar to the one used in the `sdr_converter` class is used by the constructor to set up attributes when the subcontroller object is constructed. Each attribute has a name, a value (or set of values), a range and a brief description of what the attribute does. Attributes also have an integer access modifier that determines if an attribute may be modified. The subcontroller may have attributes that indicate the version number or the type of converter. These attributes must be constant to have any meaning, and will therefore access to these attributes will be read-only.

5.3.2 The RPC interface

This section describes the implementation of the methods defined in the CORBA interface definition. CORBA will be discussed in chapter 6, and only the interface requirements will be discussed here.

If the user needs a list of the various attributes of the subcontroller it can be obtained with the `controller_attributes` method. It returns a newline-delimited string that contains all the names of the attributes of the subcontroller. The string is delimited with newlines to produce the names in a column if it is displayed on the screen.

To obtain the current value of a specific attribute the `get_controller_attribute` method has to be used. The attribute name is specified with the `attribute_name` parameter and the value of the attribute is returned in a C string. If the user wishes to modify the value of the attribute it can be done with the `change_controller_attribute` method. The attribute to be modified is specified with the `attribute_name` parameter, and the new value of the attribute is specified in the `attribute_value` parameter.

When a component is added to a software-defined radio system it is initialised with default parameters that are determined by the developer of that converter class. These defaults may not be reasonable or practical in all radio systems. The use of a text-based attribute system to manage converter parameters was introduced in section

4.3.1. The `component_attributes` function is used to get a list of attributes of a specific component of the system and acts as a wrapper around the `properties` function of the converter. The component is identified by its integer UID and the list of attributes returned by `properties` is returned in a C string.

The `get_component_attribute` method is supplied to retrieve the current value of an attribute of a converter. The converter is specified with its UID and the attribute name is specified with a string. This method wraps around the `get_attribute` method of the converter class and returns the value of the attribute in a C string. If the user wishes to modify the value(s) of a parameter of a certain converter he can use the `change_component_attribute` method. This method acts as a wrapper around the

Table 5.1: *The subcontroller RPC interface*

Name	Parameters	Return Type	Description
<code>controller_attributes</code>	-	char *	Returns a list of available subcontroller attributes.
<code>get_controller_attribute</code>	const char * <code>attribute_name</code>	char *	Returns the value of the attribute specified with <code>attribute_name</code> .
<code>change_controller_attribute</code>	const char * <code>attribute_name</code> const char * <code>attribute_value</code>	void	Modifies the value of the attribute specified with <code>attribute_name</code> to <code>attribute_value</code> .
<code>component_attributes</code>	CORBA::Long <code>c_id</code>	char *	Returns a list of the attributes of the converter with UID <code>c_id</code> .
<code>get_component_attribute</code>	CORBA::Long <code>c_id</code> const char * <code>attribute_name</code>	char *	Returns the current value of the attribute called <code>attribute_name</code> of the converter with UID <code>c_id</code> .
<code>change_component_attribute</code>	CORBA::Long <code>c_id</code> const char * <code>attribute_name</code> const char * <code>attribute_value</code>	void	Updates the value of the attribute called <code>attribute_name</code> of the converter with UID <code>c_id</code> to the new value <code>attribute_value</code> .
<code>return_component_name</code>	CORBA::Long <code>c_id</code>	char *	Returns the name of the converter with UID <code>c_id</code> .
<code>connect_net_endpoint</code>	CORBA::Long <code>c_id</code>	void	Used to connect two network endpoints together.
<code>add_component</code>	const char * <code>new_component_c</code> const char * <code>data_type</code>	CORBA::Long	Adds a converter of type <code>new_component_c</code> to the system. The data type to be used in the case of a templated class can be specified with <code>data_type</code> .
<code>remove_component</code>	CORBA::Long <code>c_id</code>	void	Deletes the converter with UID <code>c_id</code> .
<code>link_components</code>	CORBA::Long & <code>c_id_out</code> port_number <code>out_port</code> CORBA::Long & <code>c_id_in</code> port_number <code>in_port</code>	void	Used to link the output <code>out_port</code> of the converter with UID <code>c_id_out</code> to the input <code>in_port</code> of the converter with UID <code>c_id_in</code> .
<code>run</code>	-	void	Used to initiate signal processing.
<code>stop</code>	-	void	Used to terminate signal processing.
<code>is_running</code>	-	CORBA::Boolean	Returns true if the subcontroller is running. This is used to make sure that the reference obtained from the naming service is valid.

`set_attribute` method of the converter class. The UID of the converter must be specified along with the name and new value of the attribute.

Sometimes it may be necessary to identify a component at run time, for example to verify that an operation was carried out on the correct converter or to label components in a GUI CAD application. For this purpose the `return_component_name` method is supplied. It returns the string obtained when the `is_a` method of the converter is called.

The special converter implementation classes that are used to stream data across an ethernet connection was discussed in section 4.5. It was mentioned that these objects will not initiate connections when they are instantiated due to the behaviour of the low-level function calls. Another advantage of using this scheme is that it enables the dynamic reconfiguration of the network connections. To connect two network endpoints together after proper parameter setup the `connect_net_endpoint` method must be used. This method simply updates the `connect` attribute of the network converter with the specified UID.

The `add_component` method is used to add components to the system. The type of component to be added is specified with the `new_component` parameter. If the converter class is templated, the type to be used can be passed to the factory in the `data_type` parameter as a string.

If a component is no longer needed it has to be removed from the system to prevent its process method from executing. The `remove_component` method is provided for this purpose. Note that once a component is removed from a subcontroller its UID is no longer valid. UIDs are not reused, so if a component is deleted and a similar one added, it receives a new UID.

When a converter is added to the system it is not connected to any other converters. The `link` method of the `sdr_converter` class must be used to connect the output of a converter to the input of another converter. The `link_components` method is a wrapper that calls the `link` method of a converter to link the output port of that converter to the input port of another converter. The `out_port` of component `c_id_out` is linked to the `in_port` of component `c_id_in`. An output port can also be linked to an input port of the same component to implement a feedback loop.

A method is needed that is responsible for managing the signal processing of the system. This method is aptly named `run`. In its most simple form it simply calls all the process methods of the components under control of the subcontroller. Note that this call will block until the `run` method exits because of an exception or until it is stopped using the `stop` call described below. The call can be made to be non-blocking, but then all return information is lost and any exceptions thrown by the subcontroller or converters are not passed to the main application, causing the subcontroller process

to terminate. This is a CORBA feature and not a SDR design flaw. To stop all signal processing operations and exit from the run method a method called `stop` is provided.

It may be necessary for the CORBA client (the main application) to check if the subcontroller is running. It is especially helpful to check if the reference to the object is valid. It returns true if the subcontroller is running, and if the subcontroller is not running the ORB raises a `CORBA::TRANSIENT` system exception that must be handled by the client.

A summary of the above methods that also show their parameter- and return types is given in table 5.1.

5.4 Conclusion

In this chapter the subcontroller layer was introduced and discussed. The roles of the subcontroller in the overall system was also discussed; these include managing the converter components and the signal flow through the system. Another function of the subcontroller is to provide a remote procedure call interface for the main application to itself and the converter components. The implementation of this remote procedure call interface forms the basis of the next chapter.

Chapter 6

The CORBA Layer

The “best” way to do something in C is not always the same as the “best” way to do it in C++.

— Marshall Cline, C++ FAQ Lite

In keeping with the modular approach to system development, a distributed communications system is needed that allows the various components involved to communicate with each other regardless of platform and programming language. Exploring the mechanisms available for implementing a *remote procedure call* (RPC) architecture is the object of this chapter. The alternatives are briefly discussed and then a thorough overview of CORBA is given in the rest of the chapter.

6.1 The RPC mechanism: text versus binary

The first step towards an RPC implementation is to decide between a text stream based or a binary RPC mechanism. The differences between the two is shortly described in this section.

The traditional Unix way to implement RPC architectures is to use text streams. A text-based RPC implementation is highly portable, but incurs some overhead to parse the strings at both ends. This overhead may be allowable for the slow, vertical control data flow from the control application to the subcontroller. This method is especially effective for implementing RPC functionality in systems that are written with procedural programming languages because of the low level of abstraction these languages provide.

Object-oriented languages, on the other hand, provide the programmer with the ability to abstract the implementation so that the solution can be better defined with respect to the problem. This generally allows complex systems to be modelled in such a way that the solution is easier to understand and therefore easier to implement. Object-oriented programming also allows the interfaces between objects to be abstracted to

a form that is better suited to the solution of the problem. Implementing RPC functionality in systems written with object-oriented programming languages are therefore somewhat more complicated because the interfaces also provide an abstraction for the implementation.

If a text-stream RPC architecture is to be implemented in an object-oriented environment, the developer of the class has to provide methods that allow the object to be serialised and converted to a text stream on the one end and reconstructed at the other end. This can take a considerable amount of time to implement and debug—time that the developer can put to better use. Therefore, for projects that are implemented with an object-oriented language binary RPC methods are a better option.

The *Common Object Request Broker Architecture* (CORBA) offers a very elegant way to overcome cross-platform RPC difficulties, especially in object-oriented environments. It was designed to be as portable as possible, and is platform- and programming language independent. CORBA enables a client written in Borland's Delphi on Microsoft Windows to connect seamlessly to a servant written in C++ running on Linux. It allows the abstraction of object interfaces to be the same as for objects that do not have RPC interfaces. The client will therefore act on a remote object as if it were a local object.

In the rest of this chapter the basics of CORBA will be discussed. The CORBA interface between the subcontroller and the top-level application will also be discussed, as well as some of the implementation details.

6.2 Some CORBA background information

In April 1989 a group of eleven companies founded the *Object Management Group* (OMG) as a non-profit corporation [9]. The OMG was formed to create vendor-independent software standards, of which CORBA was one. Other standards created or managed by the OMG include the *Common Warehouse Metamodel* (CWM) and the *Unified Modelling Language* (UML).

After their founding in 1989, the OMG produced the first CORBA specification, CORBA 1.0, in December 1990 [21]. CORBA 1.0 was not a complete specification, though, because it provided no standard protocol for inter-ORB communication between ORBs of different vendors. Some revisions of CORBA 1 followed, and in December 1994 the OMG adopted the CORBA 2.0 specification.

The CORBA 2.0 specification defined the *general inter-ORB protocol* (GIOP), as well as the *internet inter-ORB protocol* (IIOP, pronounced "eye-op"), that enabled ORBs from different vendors to communicate with one another. GIOP is a transport-independent abstract protocol specification that defines the main protocol stack for all ORB-related communications. Introducing the GIOP protocol was perhaps the most important part

of the specification, as it ensured CORBA would be a viable standard.

The IIOP protocol is an implementation of the GIOP that uses TCP/IP as a transport. The OMG requires all ORB implementations to have an IIOP transport layer in order to be CORBA 2 compliant. This allows the ORB to have a more vendor-independent communication interface.

6.3 Technical aspects of CORBA

The technical aspects of CORBA forms the basis of this section. The interface definition language, used to specify the methods that are available via the CORBA interface, is discussed first. This is followed by a discussion of the interface of the ORB and the object adapter interface. The section ends with a discussion about obtaining object references with the CORBA naming service.

6.3.1 A note on the terms “server” and “servant”

Most distributed systems are modelled on the client-server architecture model. In this model the server is defined as the process that provides a specified service or services to a client process. The client is defined as the process that requests one or more of these services from the server process. These definitions are also valid in the CORBA context, but with a subtle difference. The following definitions are used throughout this document and is taken from the CORBA specification, section 11.2.1 [16]:

- A client is a computational context that makes requests on an object through one of its references.
- A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process.
- A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process.

Put in another way, the difference between a “server” and a “servant” is that a server corresponds to a process, whereas a servant corresponds to an object in the server process. In the SDR context the subcontroller is therefore both a server (it is an independent process) and a servant (it is the object that implements the requests made by the client). The term servant will be used throughout the rest of the chapter to indicate a subcontroller object.

6.3.2 The interface definition language

The data members and member functions of a class determines the interface of that class. All the public members and functions in a class represent the interface that are available for external objects. The ORB needs to be aware of these public methods and members in order to allow remote objects to access them. The *interface definition language* (IDL) is used to specify the methods that a servant will have available for clients to use via CORBA. An example of an interface definition is given below:

```
module hello
{
    typedef string hello_mesg;

    interface corba_hello
    {
        hello_mesg message();
    };
};
```

This definition is saved in a file with a .idl extension. An IDL compiler is then used to generate client stubs and server skeletons from this interface definition in the desired programming language, for example C++.

In the C++ language mapping, CORBA modules map to namespaces and interfaces map to classes. Strings are mapped to C strings and not to STL strings. The above interface will have roughly the following form when compiled with an idl compiler:

```
namespace hello
{
    typedef char * hello_mesg;

    class corba_hello
    {
        virtual hello_mesg message() = 0;
    };
};
```

Note that this is only a general representation, and only of the relevant code; the actual skeleton file will most likely look very different, and will also contain code to marshal and unmarshal commands. This code snippet is only intended to illustrate the mapping between IDL and C++.

All that remains is the implementation of the methods declared in the IDL file. The generated stub and skeleton classes only contain methods to handle CORBA communications between clients and servants, and the implementation of the methods specified in the IDL file must be done by the developer. The implementation of the `corba_hello` class can be done as follows:

```
#include "corba_hello.h"

class hello_impl : public hello::corba_hello
{
    hello_mesg message()
    {
        return "Hello World!";
    }
}
```

The client will now be able to call the `message` method and get “Hello World!” back from the servant.

When a method with a parameter list is defined in IDL, the direction of the parameter has to be specified. This determines whether the ORB will return a parameter from the servant or not. Parameters can be specified as either `in`, `out`, or `inout`. `In` parameters are only sent from the client-side ORB and `out` parameters are only sent from the servant-side ORB. `Inout` parameters, on the other hand, are sent from the client-side ORB and returned from the servant-side ORB. In C and C++ both `out` and `inout` parameters are passed by reference when the function call is made, but the `out` parameters are not marshalled by the client-side ORB.

To define a STL vector-like container in CORBA, the `sequence<>` keyword is used. CORBA sequences, like STL vectors, provide resizable array-like containers but without the dangers of ordinary C-style arrays. CORBA also provides an array type that maps directly to the C/C++ array type. Generally, sequences are preferred over arrays because it has the following advantages:

- Sequences provide bounds checking of subscripts to help prevent illegal memory access
- Sequences are dynamically resizable
- Sequences manage their own memory

CORBA sequences are templated and can therefore be used with any type supported by the CORBA interface definition language.

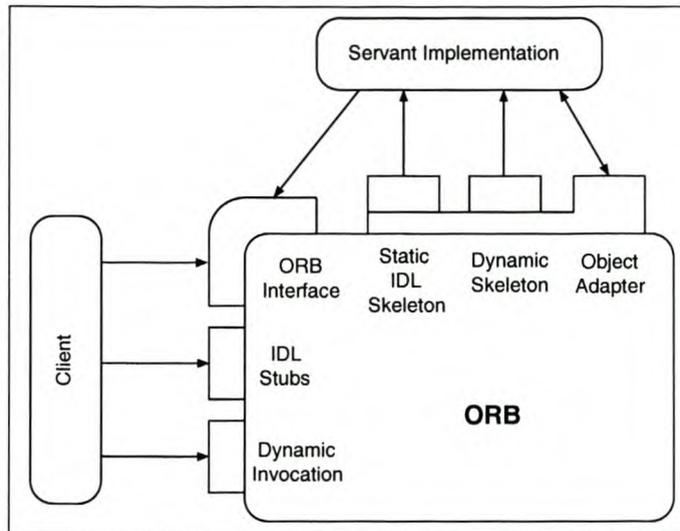


Figure 6.1: *The ORB and its interfaces to servants and clients*

6.3.3 ORB interface

The ORB and its interfaces are illustrated in Figure 6.1. The main ORB interface to the client and servant objects is standardised by the OMG. This ensures that any CORBA client or servant can use any CORBA compliant ORB to interact with one another.

The IDL stub and skeleton interfaces are created from an IDL file, and are therefore implementation-specific.

The *dynamic invocation interface* (DII) and the dynamic skeleton interface are standardised interfaces that are used by a client to discover servant object methods that were not specified in the IDL file that was used to generate the client stubs. This interface is typically used when the servant has been modified after the client has been built. As the DII is beyond the scope of the work presented, it will not be discussed further.

The object adapter interface provides a way for the ORB to communicate with servant objects. Note that the skeleton interfaces connect to the ORB via the object adapter interface. This interface will be discussed in detail in section 6.3.4.

6.3.4 Object adapters

An object adapter is an interface that allows CORBA objects to communicate with the ORB at run time. The ORB uses object adapters to allow clients to make requests to servant objects, and to send the reply of the servant object back to the client. Note that the object adapter is part of the ORB and not of the objects; when a servant object binds to an ORB, it does so via the ORB's object adapters.

The first object adapter that was available was the *basic object adapter* (BOA) [27]. The BOA specification did not define the interfaces between the ORB, BOA and the servant object. Furthermore, because the BOA interface specification is so incomplete, different vendors almost totally integrated it into the ORB, provided their own interface, and the BOA is totally unportable as a result.

With the adoption of the *portable object adapter* (POA) specification, the portability problems that the BOA interface presented was eliminated. The POA provides an extra layer of abstraction between servant objects and the ORB via a well-defined interface. This ensures that the POA is portable across different vendor implementations.

To use this interface, a generic object reference is first bound to the root POA. The reference to the root POA is then narrowed to a generic POA reference, and the generic object reference is assigned to the desired object. Finally, the object must be activated before the POA can send requests to the object. The following code shows all the steps:

```
// Initialise the ORB. The "omniORB" argument is optional and only used
// to specify a specific ORB implementation if more than one ORB is
// installed on your system
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB4");

// Obtain a Root POA reference
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");

// Narrow this reference to a generic POA reference
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

// Create the object and obtain a reference to it
my_class *my_object = new my_class();

// Activate the object
PortableServer::ObjectId_var my_objectid = poa->activate_object(my_object);

// Obtain a reference to the object
obj = my_object->_this();
```

This is almost all that is needed to build a server application. All that is needed further is to create a reference to the CORBA object, and the ORB must be activated, which will be discussed later.

6.3.5 The CORBA naming service

The next step towards creating a CORBA server application is to create a way for clients to address the servant object. There are two ways to do this. The first method is to create

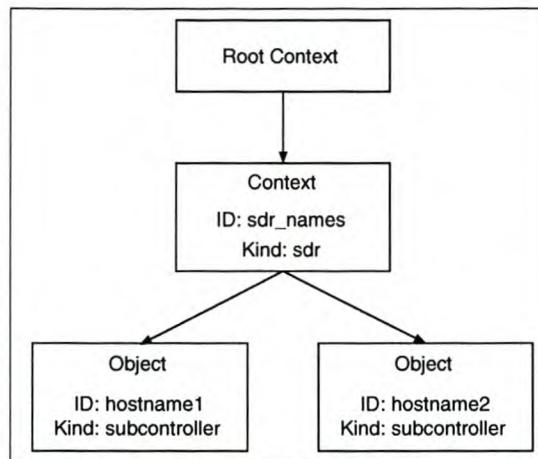


Figure 6.2: *The SDR INS Context Structure*

an *interoperable object reference* (IOR) and convert it to a string. The IOR is the reference that the ORB uses to communicate with a servant object and contains information about the host the servant runs on, what port number the servant listens on and what type of character encoding to use. This string is then passed to the client application as a command-line parameter, or it is read from a file, and the client converts the string back to an object reference. This approach is, however, only viable if both the server and the client runs on the same machine as a string representation of an IOR can be several hundred characters in length and therefore impractical to type on a keyboard.

The second method is much more simple, and only requires a few extra lines of code in both the server and client implementations. As before, an IOR is obtained, but instead of converting it to a string and writing it to the screen or to a file, the IOR is stored in an *interoperable naming service* (INS) server, where it is associated with a string. Once a name is stored in the INS, a client can obtain the desired object's IOR from the INS using the string that the IOR is mapped to. Associating an object with a name in the INS server is called *name binding* [17].

Graphically, the INS can be represented as shown in figure 6.2. It shows the naming service structure used for the subcontroller objects. A *context* is an object that contains a set of unique name bindings and is typically used to group related name bindings together. A context can also contain subcontexts which in turn can contain either name bindings, subcontexts or both. Context- and object names are composed of an "ID" and a "kind" field. The ID field is used to indicate the name of the object and the kind field is used to indicate the type of the object.

The root context is the lowest context that an object or a context can be bound to and is typically only used as a container for subcontexts and not objects. To keep things simple, only one context named "sdr_names.sdr" was bound to the root context. This

context then holds all the different subcontroller objects. All subcontroller objects are then bound to this context with names of the form “*hostname.subcontroller*”. The name of the host that a subcontroller is running on is therefore used as the object name. For uniformity all objects use “subcontroller” in the kind field.

The INS is in many ways similar to the *domain name service* (DNS) used to obtain a numerical IP address of a computer on a network. The DNS maps numerical IP addresses to host- and domain name combinations on a one-to-one basis. If you want an IP address you send a query to the DNS server. The DNS server then resolves the given host name to an IP address and returns the IP address. If you want an IOR of a specific object, you send a query to the naming service server that contains the name of the object as well as the naming context that the object is binded to. The naming service then resolves the object name to an IOR and returns the IOR to the client. Once the client has the IOR of the servant object, it can send requests to the servant via the ORB.

A few static helper functions were created to act as wrappers for useful INS routines, and is provided to hide the implementation details of the name service operation from the developer. The first is used to bind a servant object to the naming service and is called `bind_obj_to_name`. The first argument passed to the function is a reference to the ORB that interacts with the servant. The second argument is a reference to the instance of the servant object. Two string arguments are used to specify the ID and kind of the object respectively. From figure 6.2, for the left object “hostname1” was used for the ID field and “subcontroller” for the kind field. The `bind_obj_to_name` function automatically binds the context `sdr_names.sdr` to the root context if it is not found. If the `sdr_names.sdr` context exists its reference is resolved before the subcontroller object is binded to the it.

The second helper function is used to obtain IORs from the naming service and is called `get_obj_ref`. The first argument passed to the function is a reference to the ORB that interacts with the servant. The two string arguments specify the ID and kind of the object, and must be the same as the arguments passed to `bind_obj_to_name`. The context `sdr_names.sdr` is specified inside the function.

The third helper function, called `unbind_obj`, is used to unbind an IOR from the naming context it was bound to. A subcontroller object will typically use this function to remove its reference from the naming service server before it exits to prevent a client from obtaining a stale reference to the object. The first argument passed to the function is a reference to the ORB that interacts with the servant and the two string arguments are used to specify the IOR that must be unbinded from the naming service server.

The last helper function is called `list_objects` and is used to query the naming service for all the references that is bound to it. This method can be used by the control

Table 6.1: CORBA helper functions

Name	Parameters	Return Type	Description
bind_obj_to_name	CORBA::ORB_ptr orb CORBA::Object_ptr objref std::string c_object_id std::string c_object_kind	void	Binds the object reference objref to the Naming Service using c_object_id as the object ID and c_object_kind as the object kind.
get_obj_ref	CORBA::ORB_ptr orb std::string c_object_id std::string c_object_kind	CORBA::Object_ptr	Obtains an object reference to the object with ID c_object_id and kind c_object_kind.
unbind_obj	CORBA::ORB_ptr orb std::string c_object_id std::string c_object_kind	void	Unbinds the object with ID c_object_id and kind c_object_kind from the naming service.
list_objects	CORBA::ORB_ptr orb std::vector<std::string> objects	void	Lists all the objects bound to the naming service.

application to discover all the available subcontrollers.

A summary of these methods is given in table 6.1.

6.4 Integrating CORBA into the SDR architecture

In this section the CORBA interface between the subcontroller and the main application will be discussed. It is not an exhaustive discussion due to the fact that the RPC interface of the subcontroller has already been discussed, and is only intended to show how CORBA is used to achieve RPC functionality in the system.

6.4.1 Interface definition

The CORBA layer has to provide a remote communication mechanism for the subcontroller interface that was introduced in section 5.3.2. This interface is specified using CORBA IDL and then translated to corresponding C++ code with an IDL compiler. A summary of the methods specified in the CORBA interface definition is given in table 6.2.

A basic call flow diagram of the SDR system is shown in figure 6.3. It shows how a member function call from the control application (client) is invoked on the servant object via the ORB. The client- and server side ORBs are depicted as one logical ORB for simplicity.

The client invokes the operation on the servant by using the CORBA reference of the object like a pointer to the object. The client-side ORB then marshals this request and forwards it to the servant-side ORB. The servant-side ORB then unmarshals the request and invokes the requested method on the servant object. If a return value is requested, the servant-side ORB marshals the return value and forwards it back to the

Table 6.2: CORBA IDL function declarations

Name	Parameters	Return Type	Description
controller_attributes	-	string	Returns a list of subcontroller attributes.
get_controller_attribute	in string attribute_name	string	Returns the value of the attribute specified with attribute_name.
change_controller_attribute	in string attribute_name in string attribute_value	void void	Modifies the value of the attribute specified with attribute_name to attribute_value.
component_attributes	in long c_id	string	Returns the list of attributes of the converter with UID c_id.
get_component_attribute	in long c_id in string attribute_name	string	Returns the value of the attribute named attribute_name of the converter specified with c_id.
change_component_attribute	in long c_id in string attribute_name in string attribute_value	void	Modifies the value of the attribute named attribute_name of the converter specified with c_id to attribute_value.
return_component_name	in long c_id	void	Returns the name of the component with ID c_id.
connect_net_endpoint	in long c_id	void	Used to connect two network endpoints together. The ID c_id is the UID of the network converter.
add_component	in string new_component_c in string data_type	long	Adds a converter of type new_component_c to the subcontroller. The data type can be specified with data_type (for templated classes).
remove_component	in long c_id	void	Removes a converter from the subcontroller.
link_components	in long c_id_out in long out_port in long c_id_in in long in_port	void	Links output port out_port of the converter with ID c_id_out to the input port in_port of the converter with ID c_id_in.
run	-	void	Used to start the processing of samples.
stop	-	void	Used to stop the processing of samples.
is_running	-	boolean	Returns "true" if the subcontroller is busy processing samples.

client-side ORB. The return value is unmarshalled and passed to the client and the call is finished. This whole process is transparent to the client—the client is unaware that the object is in another memory space, and therefore appears to be a “normal” local object.

6.4.2 User exceptions in CORBA

The above methods are all invoked via the CORBA interface, and can raise exceptions like `CORBA::COMM_FAILURE`, `CORBA::SystemException` or user-defined exceptions. To be able to use user-defined exceptions across the CORBA interface, the `raises` keyword is used to tell the IDL compiler to create the relevant code to handle exceptional conditions.

To provide exception support between servants and clients an exception interface

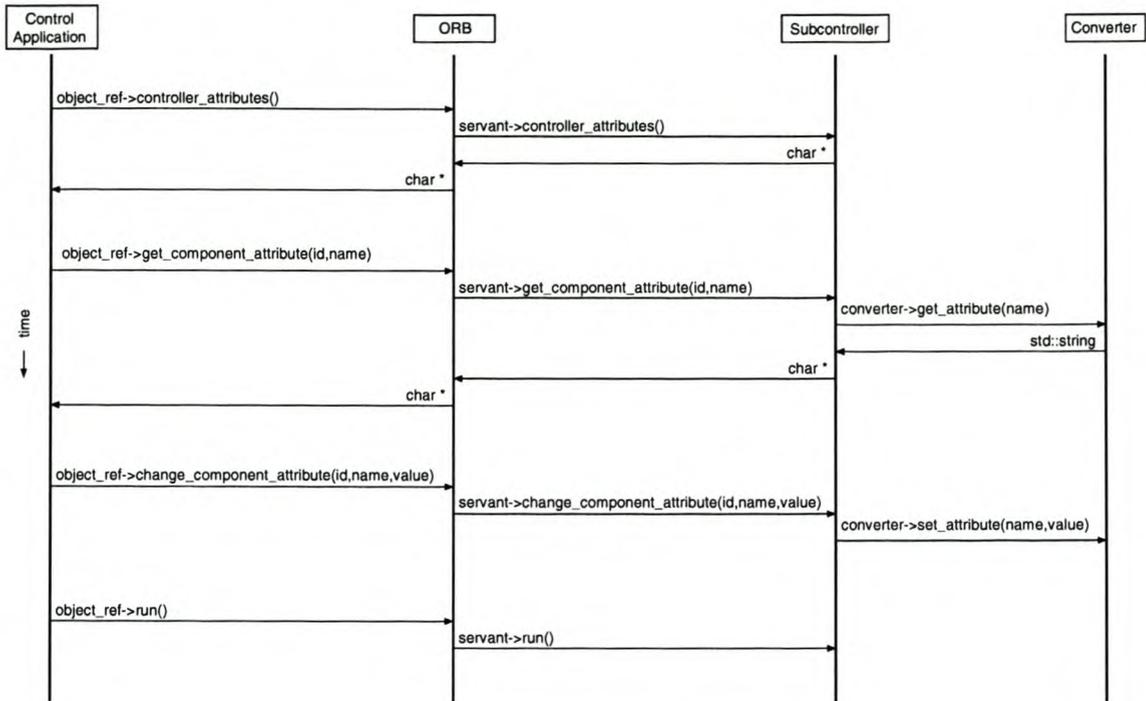


Figure 6.3: Function call flow diagram showing CORBA encapsulation

called `converter_exception`, was defined. This exception class is used to encapsulate exceptions that are generated in the `sdr_converter` class.

```

exception converter_exception
{
    string c_name;
    long e_type;
    string e_desc;
};
  
```

The `converter_exception` class contains three members to describe the exceptional condition. The `component_name` member is used to indicate which component raised the exception, and is typically obtained from the `is_a` function of the converter. The `ex_type` member is a numerical value used to identify the exception raised by the component. The `e_type` value obtained from the exception raised by a converter is used for this member, and is typically one of the values defined in the `sdr_definitions.h` file. The `ex_desc` member is used to provide a verbose description of the exception, and is also taken from the exception raised by a converter.

This class can also be used to indicate exceptional conditions that are not generated by a converter but relates to converter management. Examples of the use of this exception class where an exception is not raised by a converter include, but are not

limited to, the following:

- Requesting a non-existing component type to be added to the system.
- Attempting to perform an operation on a component that is not part of the system.

In these cases the members of the class are assigned appropriate values when they are generated.

6.4.3 The Subcontroller ORB

To enable the use of CORBA each subcontroller needs an ORB to handle communications with the clients. Section 6.3.4 describes how a POA to the object is created, and section 6.3.5 describes how to create an IOR for the object and bind it to the Naming Service. These steps are performed as part of the main routine of the subcontroller executable.

The ORB itself is implemented as a Singleton [7] class in a helper class called `sdr_orb`. This class contains static methods that are used to initialise and shut down the ORB.

A shutdown signal handler is also provided to enable the ORB to shut down gracefully when the subcontroller executable receives a SIGINT (interrupt) or SIGTERM (terminate) signal.¹ The ORB shutdown routine runs in its own thread and is started in the main routine before the ORB is activated. It then waits for either a SIGTERM, SIGINT or a call to the shutdown handler before it unregisters the ORB and shuts it down.

The `sdr_orb` class can be made aware of its INS ID and kind via two private member variables. These values should be used when the IOR of the ORB is binded to the Naming Service to ensure that the correct objects are referenced by the clients.

6.5 Conclusion

The RPC interface used in the SDR architecture was the subject of this chapter. The differences between binary and text-based RPC was first discussed, with the pros and cons of each approach highlighted. Then basic CORBA concepts like the IDL, POA and INS were introduced and discussed, along with how these concepts were integrated into the architecture.

The focus of the next chapter will be on the control application, which makes use of CORBA to access the subcontroller interface defined in the previous chapter.

¹At the moment signal handling is only implemented for *nix platforms.

Chapter 7

The Main Application: SDR Builder

Any system that is made for use by human operators is not very useful if it doesn't have a user interface. This user interface is the part of the system that allows the user to interact with the components of the system. User interfaces can be implemented with numerical keypads, keyboards, pointing devices or special devices used by disabled persons. User interface software then interprets the signals from these devices to allow the application to carry out the requested operation, and usually provides visual feedback to the user via the screen.

The top-level application of the SDR architecture provides interface software to the software radio system. This software is used to interpret user input from the peripheral devices like the keyboard and the mouse, execute the requested task and provide visual feedback to the user. Interface software can be in the form of a *graphical user interface* (GUI) or a simple text-based user interface (TUI). The choice between a GUI or a TUI depends on various factors¹, and for the demonstration implementation a GUI was selected and implemented.

The top-level application interface will be discussed in this chapter along with the GUI implementation.

7.1 Interface requirements

Ideally, a user interface should provide the user with an interface that has a very intuitive, logical structure. This makes the application easy to use and also reduces any frustration the user may experience. A well designed interface also reduces the time spent reading user manuals to understand the application.

A second requirement is that the user interface must be portable across various platforms. This ensures that the minimum effort goes into porting the application to a

¹These factors will be discussed later in the chapter.

new platform if the need arises.

7.2 The SDR Builder user interface

7.2.1 Selecting a user interface library

The choice of a user interface for the system was influenced by a number of factors. The first to consider was that the user interface should fit in with the modular programming approach used in the rest of the system. This means that the user interface should be able to “plug in” to any subcontroller on any platform. This attribute is realised with the CORBA interface.

Another desirable attribute of the user interface is that it should be as portable across various platforms as possible. This means that to port the interface to a new platform, extensive modifications of the source code should not be necessary, and ideally, the source should only have to be recompiled on the new platform to build binary code for that platform.

A third factor to keep in mind is whether the underlying human interface device uses a graphical or textual display system. Usually, a device that has a graphical visual feedback system can also handle text-based feedback, but not vice versa. Unfortunately, this problem does not have a “one size fits all” solution, and two separate implementations are needed.

For text-only environments the ncurses² library offers a very elegant solution. The ncurses library is fully portable across all Unixes, Linux, Microsoft Windows and most other popular platforms. The only drawback is that ncurses has a very steep learning curve, but when mastered it offers a very powerful user interface for text-only environments.

For graphical environments there are quite a few options. WxWindows and the *Gimp Toolkit* (GTK) are both open source GUI APIs that were originally created for Linux, but also offers support for the Windows environment.

Another option is the QT library from Trolltech. This library offers a complete GUI API as well as some general-purpose libraries that need not only be used in a GUI environment. QT is also portable across operating system platforms and it can even be used in embedded environments. The only drawback is that it is only free for open source and free software and it is only available with a commercial license for the Windows environment.

Despite this, QT was chosen as the platform to base the user interface on. The

²The older, non-free curses library is also available

primary reason for this decision was that QT has a very versatile set of libraries and it has a relatively simple API. Another property that makes the QT library very suitable for use in the current context is that it also contains a platform-independent thread library that can be used to spawn multiple threads of a function call. This is needed when the call to the subcontrollers is issued to start processing signals (refer to the run method described in section 5.3.2).

7.2.2 User interface layout

The layout of the SDR Builder application is summarised in Table 7.1. The table lists each available menu item and its contents, as well as sub-menus and their contents.

Table 7.1: *The SDR Builder interface layout*

Menu Item	Contents	Sub-menu contents
File	New System Open System Save System Save System As	
System	Run Suspend Resume Stop	
Components	Add Component Remove Component Link Components Activate Network Link Attributes	Query Component Get Component Attribute Set Component Attribute
Controllers	Add Subcontroller Select A Subcontroller Remove Subcontroller Attributes	Subcontroller Attributes Get Subcontroller Attribute Set Subcontroller Attribute
ORB Setup	Initialise ORB	
Help	About	

A few screenshots are shown to illustrate the design of the GUI. Figure 7.1 shows the main SDR Builder window. Figure 7.2 shows the subcontroller menu and its contents. This menu is used to add, select and remove subcontrollers, and also to query and modify any subcontroller attributes.

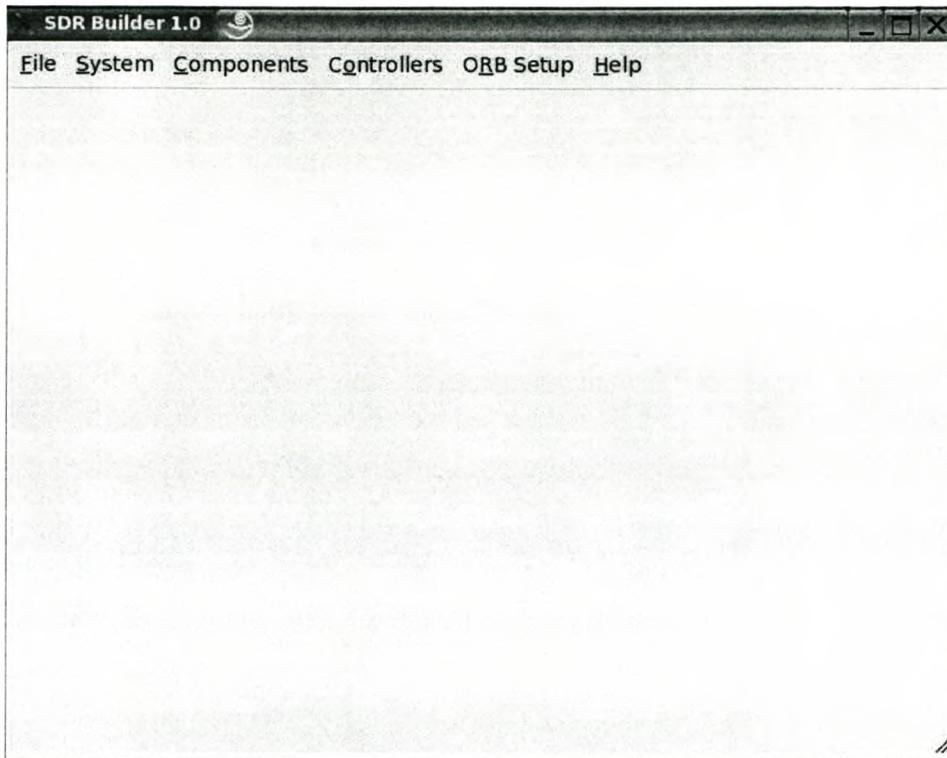


Figure 7.1: *SDR Builder main window*

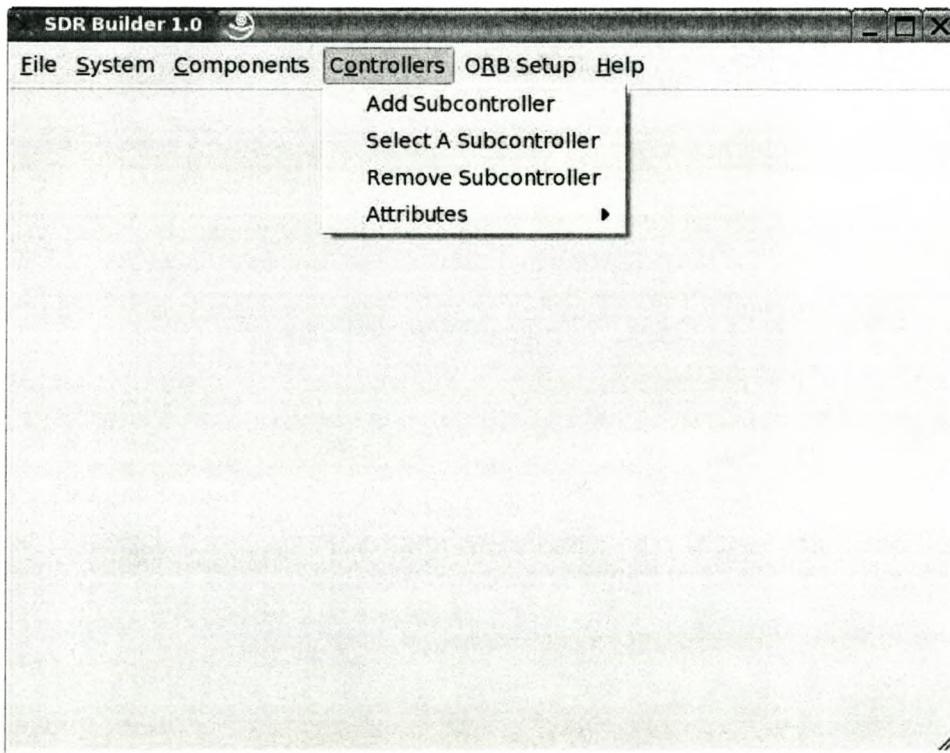


Figure 7.2: *The subcontroller menu*

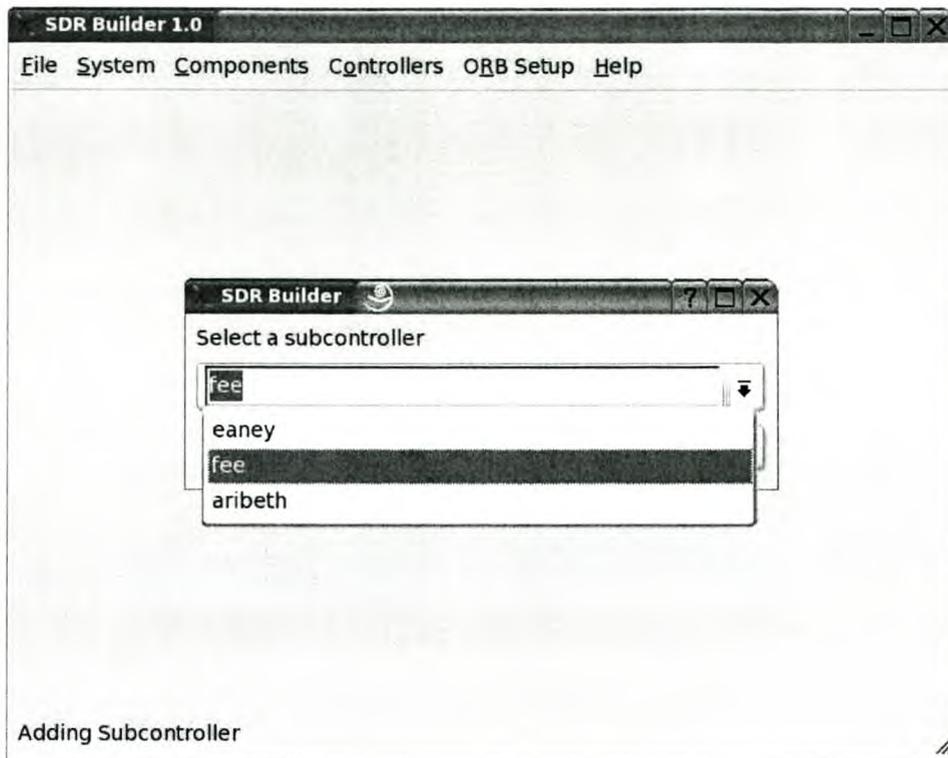


Figure 7.3: Adding a subcontroller to the system

If the user clicked on the “Add subcontroller” menu item, the dialogue box shown in figure 7.3 is shown. This shows a list of active subcontrollers that were obtained from the naming service. The user can select any one from the drop-down list.

Figure 7.4 shows the converter menu and its attribute submenu. Here the user can select to add, remove or link components as well as activate the network interface. The attribute submenu allows the user to query a converter for a list of its attributes as well as get the current value of an attribute. The value of an attribute can also be modified from this menu.

Figure 7.5 shows the pop-up dialogue that allows the user to link two converter components together. The name of the source converter must be entered in the first box, and that specific converters’ UID must be entered in the second box. This is necessary to ensure that, if more than one converter of a certain type is present in the system, the correct one is chosen. The output port number of the converter must be entered in the third box. The name of the destination converter and its UID must be entered in the fourth and fifth boxes, respectively, and the input port that the other converter’s output must be connected to must be entered in the sixth box.

When the user selects the “Query component” item from the converter menu, a dialogue box appears. The user must enter the name of the converter along with its UID in the fields provided. After the user clicks on “OK”, a dialogue similar to figure

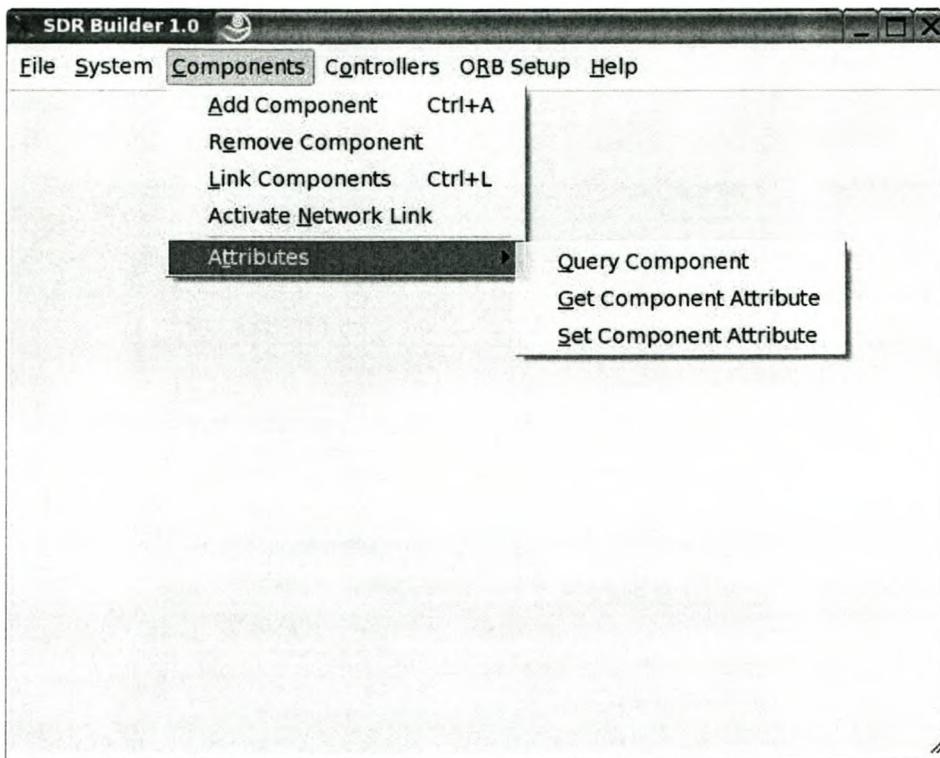


Figure 7.4: *The converter menu and attribute submenu*

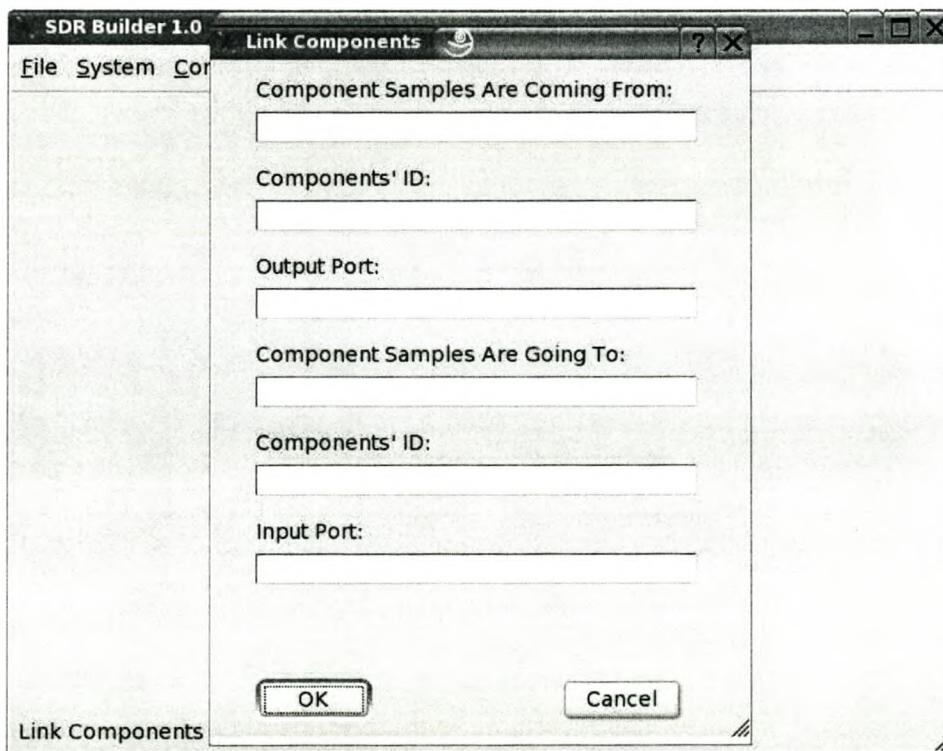


Figure 7.5: *Linking two converters*

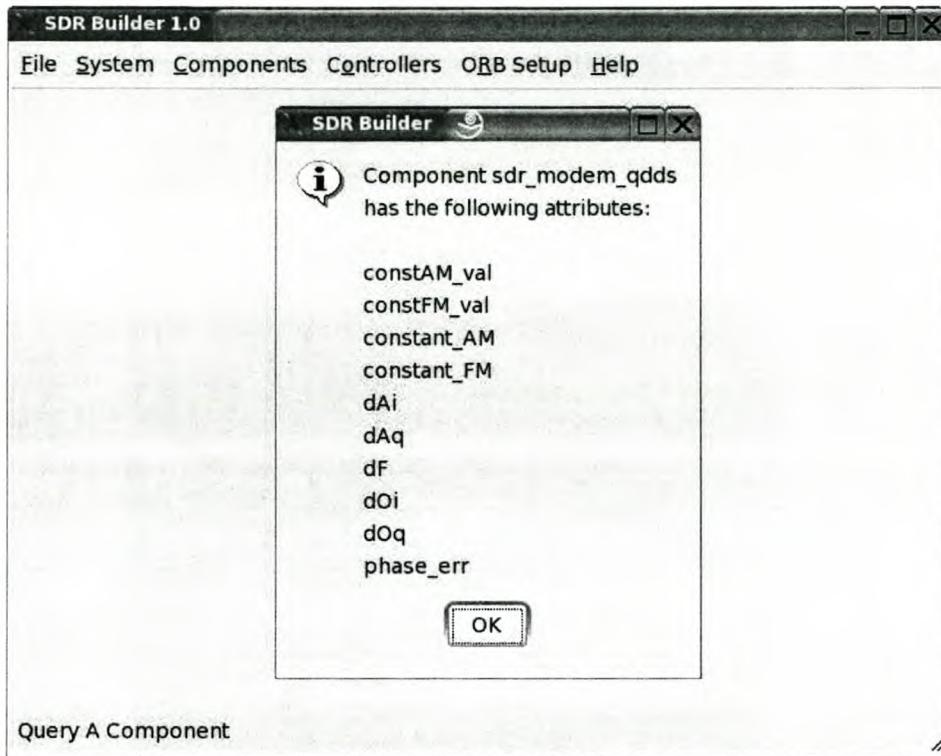


Figure 7.6: *Querying a component for its available attributes*

7.6 will appear. This dialogue simply displays the list of attributes returned by the converter.

To activate the signal processing, the user must select “Run” from the “System” menu. A “Stop” item is also provided to halt all signal processing.

7.2.3 Subcontroller and component management

Effective subcontroller- and component management are key elements to an efficient SDR Builder implementation.

Figure 7.7 shows how components are managed in the SDR Builder. References to the different converters are stored in a STL multimap with QString as key type and integer as value type. A multimap is used because there can be more than one of each type of component in a subcontroller.

Subcontrollers can have different components and each subcontroller also has a unique CORBA reference. The component container mentioned above is stored in a struct, aptly named subcontroller, together with the IOR of the subcontroller. These subcontrollers are stored in a STL map container with a QString as key type. The key doubles as the name identifier of the subcontroller.

A component is identified by its ID, name and the subcontroller in which it resides.

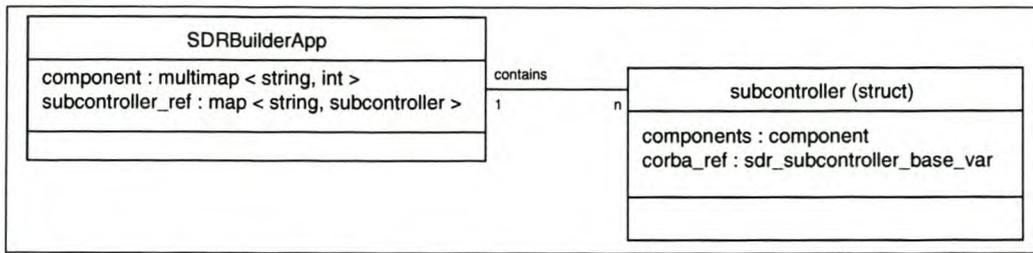


Figure 7.7: *Subcontroller and component management in SDR Builder*

The component UID can be thought of as having the following form: `subcontroller.component_name.component_id`. The subcontroller field must be unique, therefore even if two subcontrollers contain the same component with the same ID, each one will have a unique ID in the SDR Builder. Figure 7.7 also reveals that there is only one SDR Builder instance that contains many subcontroller instances.

7.3 Threads in SDR Builder

It has been mentioned that different threads must be used to call the run methods of the various subcontrollers due to the nature of the run method and the mechanisms used by the CORBA architecture. The difficulty now is that the pthread library available in Linux only supports C functions and not C++ functions. A workaround for this problem is to use a combination of type casting and static functions.

There are, however, several available C++ wrapper classes that can be used as alternatives to this approach. The QT library contains a thread wrapper class called `QThread`, and this class is totally independent from the rest of the QT library. `QThread` can therefore be used in applications or classes that do not make use of the rest of the QT library. To use the `QThread` library, a new class must be created that is a descendent of the `QThread` class. The `QThread` class contains a pure virtual function called `run` that must be implemented in the derived class. The body of the `run` method must contain the code that needs to execute in the thread. The derived class is called `SDRBuilderAppThreads` in the SDR Builder application.

The different threads are only needed to call the run method, and are not needed when the system has been stopped or before the system must process samples. In addition to the main thread, one thread is needed for each subcontroller. To manage the threads, a STL vector container was created that holds pointers to the different instances of `SDRBuilderAppThreads`. When the system must start to process samples, the amount of subcontrollers are determined and instances of `SDRBuilderAppThreads` are spawned with the C++ keyword `new`. `new` returns pointers to these instances, and these pointers are stored in the aforementioned vector. `SDRBuilderAppThreads::run`

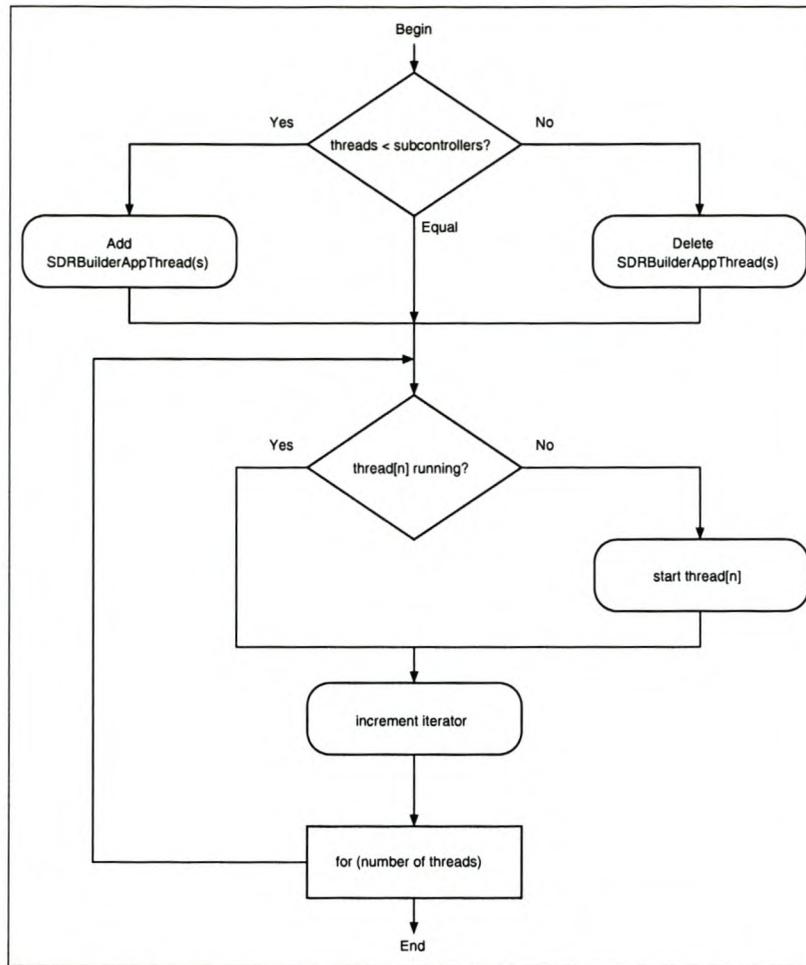


Figure 7.8: Flowchart of the *slotSystemRun* method

is now called for each instance, and the system executes.

The main thread is still free to execute commands in parallel with the child processes, and if a multi-threaded ORB (like omniORB) is used for the CORBA communications, it can even send commands to the subcontrollers *while their run methods are running*. This enables modifying the attributes of components as well as total system reconfiguration³ without halting the system.

7.4 Some last remarks about GUI libraries

The QT library that was used to create SDR Builder has some very attractive features. It provides attractive GUIs on both Linux and Windows, offers platform-independent

³This includes, among others, adding new components, modifying links between components and so on.

threads and even caters for Embedded Linux platforms. QT is unfortunately only available free of charge for X11 if it is used to develop non-commercial applications. If the user interface has to be implemented on Microsoft Windows or Apple Mac OS a commercial QT license is needed.

An alternative to QT can be realised with a combination of two open source libraries. GTK (mentioned at the beginning of the chapter) can be used as a portable GUI library and the Boost thread library for portable threads. Both these libraries are open source libraries that can also be used to create proprietary applications.

Further studies will have to be done to determine if the QT library was in fact the best choice to base the user interface on.

7.5 Conclusion

The rudimentary user interface that was developed for the SDR project was discussed in this chapter. The factors that had to be considered when choosing a user interface library were discussed in sections 7.1 and 7.2.1, as well as a few available options. The graphical user interface that was developed was discussed, as well as some of the implementation details. The drawback of the commercial nature of the QT license was also discussed, along with possible alternatives.

The performance of the architecture will be discussed in the next chapter.

Chapter 8

Evaluation of the Architecture

The proof of the pudding is the eating. — Miguel de Cervantes Saavedra

At first the idea of a reconfigurable radio system sounds quite promising, especially given the diverse communications standards available at the moment. Having a single terminal that is compatible with more than one communications system will be especially handy for people who travel to various parts of the globe.

This idea would, of course, only have merit if the performance of the software radio is comparable with its analogue counterparts. In this chapter various aspects of the system is measured and evaluated to assess whether such a system is feasible.

8.1 Evaluation process overview

Several different tests were performed to evaluate the performance of the system. Firstly, the maximum data throughput was measured as a function of the number of converters in the system. The throughput rate was also measured for individual sample processing and batch processing configurations to assess the efficiency of each approach. Tests were also carried out to measure the effect of frequent attribute modifications on the throughput of the system.

The amount of memory consumed by the system as a function of the number of converters in the system was also investigated. This was done to investigate whether the allocation of memory is a linear function of the number of converters in the system.

The amount of latency introduced by the system was also measured as a function of the number of components in the system. As with the other tests, the effect of individual sample processing versus batch processing was also investigated.

The last set of tests that were carried out was done to evaluate the analogue performance of the system. A simple FM transmitter was implemented for this purpose.

Table 8.1: *The various system configurations used to evaluate the architecture*

	System 1	System 2	System 3	System 4
Hardware configuration				
CPU	AMD Athlon XP 2000+ (1670 MHz)	Intel Celeron 667 MHz	Intel Celeron 1700 MHz	AMD Athlon XP 1900+ (1600 MHz)
RAM	1 GB DDR266	384 MB PC133	512 MB DDR200	256 MB DDR266
Software configuration				
OS	Linux 2.4.25	Linux 2.4.20	Linux 2.4.21	Linux 2.4.20
Compiler	GCC 3.3.3 (Debian)	GCC 3.3 (SuSE)	GCC 3.3.1 (SuSE)	GCC 3.3 (SuSE)
CORBA ORB	omniORB 4.0.3	omniORB 4.0.3	omniORB 4.0.3	omniORB 4.0.3

8.1.1 System configurations

Various systems were used to evaluate the performance of the architecture of the software radio system. This was done in order to investigate how the different hardware configurations affect the performance of the system. The software configurations of these systems were fairly uniform to limit the number of variables present in the system.

Standardised converters were created to evaluate the performance of the system. The first converter that performs a simple *multiply-and-accumulate* (MAC) operation was used to measure the throughput of the system. To measure the latency of the system, a timestamp generating converter was created that obtains a timestamp from the system clock and passes it on to the next converter. This converter is placed at the front of the converter queue. At the end of the converter queue another special converter was added to determine the latency. This converter obtains its own timestamp and computes the latency of the system by subtracting the time value obtained from the first converter from its own time value. This system is illustrated in figure 8.8.

8.1.2 Terminology

In this section the terminology associated with the classification of software defined radio systems that are built upon the architecture described in this document are explained.

In the rest of the chapter, systems will often be classified as “small”, “medium” or “large”. Due to the fact that a single component type was used in these tests, the terms do not necessarily reflect the size of the system in terms of the number of converters in the system. The size of a system is primarily determined by the following factors:

- The number of converters in the system.

The total number of function calls that are not related to signal processing, but

rather to signal transfer, is directly proportional to the number of converters in the system. Therefore, as the number of converters in the system increases the function call overhead also increases.

- The average number of function calls per call to process.

If a system has relatively few converters but all the converters are quite complex, the number of function calls executed during each process cycle will be high. Therefore the complexity of the algorithm implemented in the converter determines the number of signal processing-related function calls.

- The amount of memory consumed.

Processors have a certain amount of high-speed cache memory on-chip. This memory acts as a buffer between the high-speed processor and low-speed “external” memory modules. As the memory occupied by the system increases the amount of memory swapped in and out of the cache will also increase. If a block of memory has to be swapped that is larger than the size of the cache the processor will have to wait for some of the data, increasing processing time significantly.

From the above discussion it is clear that the size classification of a system is a function of both the total number of function calls and memory usage, and not necessarily of the number of converters present in the system. In the following sections the classification will become more apparent.

8.2 Data throughput performance

In this section the throughput performance of the system will be evaluated. The relationship between the throughput and the number of components in the system was investigated and will be discussed.

8.2.1 Purpose of the experiment

The purpose of this experiment was to evaluate the raw data throughput capability of the system as a function of the number of components in the system. This was done to investigate whether the processing time of the system increases linearly as the number of components in the system increases, and also to determine the signal bandwidth capabilities of the system.

The effect of processing samples in batches was also investigated as part of this test. Processing samples in batches generally leads to higher throughput than single sample processing, at the expense of increased latency. This test was performed to determine

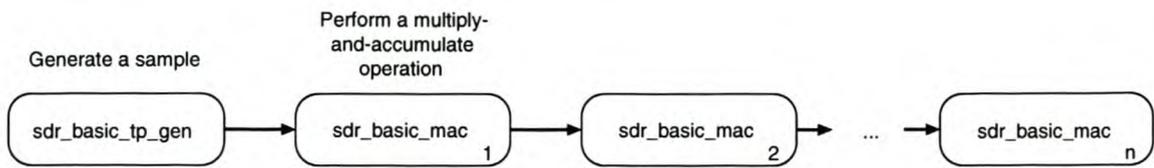


Figure 8.1: *The converters used to measure the throughput of the software radio system*

the relationship between the number of samples in a batch and the throughput of the system.

8.2.2 Experimental setup

System 1 (table 8.1) was used for this experiment. The converter configuration of the system is illustrated in figure 8.1. To determine the throughput, the time taken to process 1 million samples (T_p) was first measured. The throughput was then determined with the following equation:

$$\text{Throughput} = \frac{10^6}{T_p}$$

As mentioned, the effect of processing samples in batches was also investigated. The sample generator, `sdr_basic_tp_gen`, was used to accomplish this. It could be set to generate samples in batches, which the MAC converters would then consume when their process methods are called. For these tests, samples were generated in batches of 10, 100 and 1000 samples.

8.2.3 Results of the experiment

The results of the experiment are plotted in figure 8.2. It shows the time required to process 1 million samples as a function of the number of converters in the system. The graph also shows the results of the batch processing experiments. The throughput of the system, as determined with the above equation, is plotted in figure 8.3.

Of particular interest is the increase in the slope of the graph at about 600 components. This is caused by two factors:

- High function call overhead

If the samples are processed individually, `process` is called once for each sample. This is very inefficient because it requires a large number of function calls for each sample that is processed. This will be discussed in more detail later in this section. Single sample processing also requires that, for each sample, the processor has

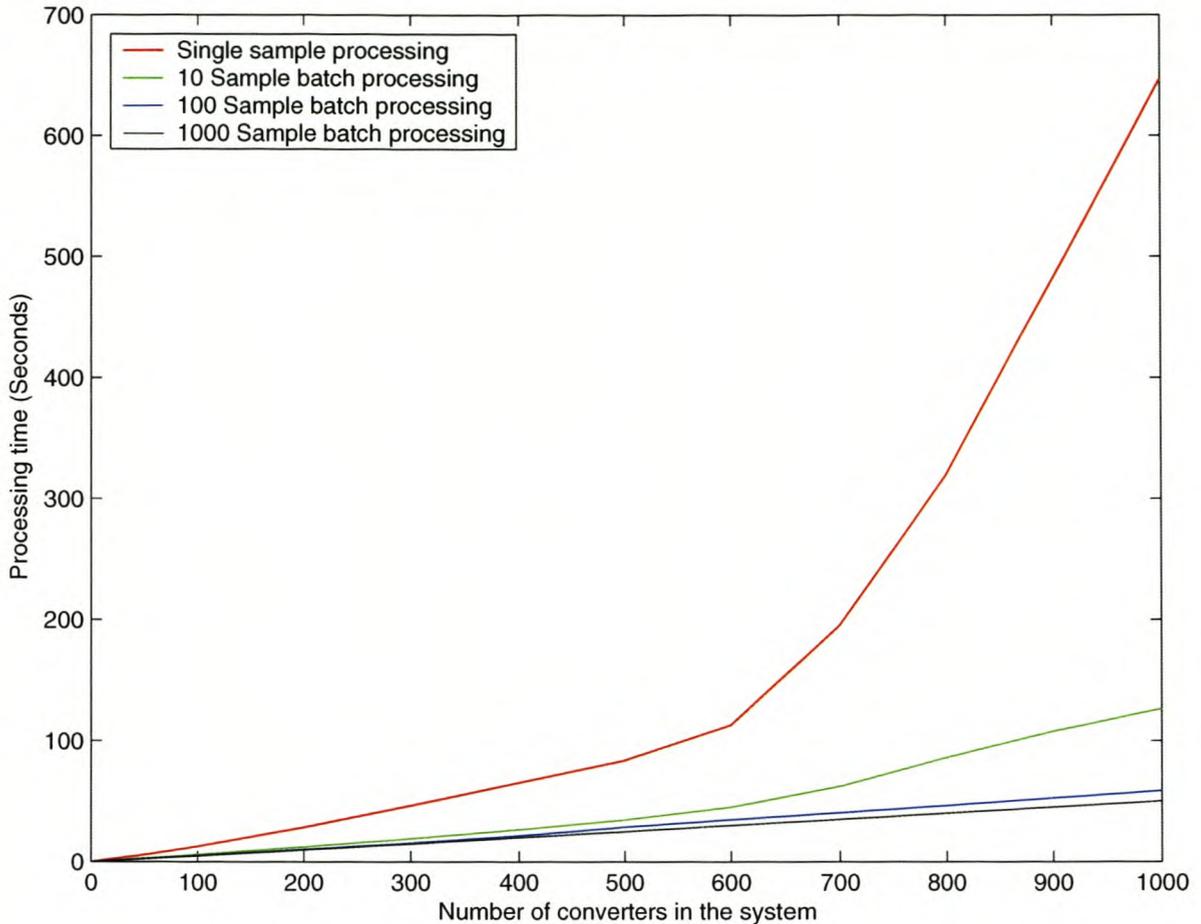


Figure 8.2: Time required to process 1 million samples as a function of the number of components in the system

to load the entire function from main memory into its cache memory if it is not present in the cache.

- Finite cache memory size

Most general-purpose processors have a limited amount of on-board cache memory. The cache acts as an intermediate storage space for the data and instructions that are required by the processor, and is necessary because of the large clock speed difference between the processor core and external memory. This offers considerable processing speed advantages, especially for instructions and data that are used repeatedly. When the total memory used by the components is more than the processor's cache, the processor will (depending on its cache management strategy) replace the "oldest" data in its cache with the latest required data, resulting in a continuous swapping of data to and from the main system memory.

The effect of call overhead with a large number of components is clearly visible for

the case where samples are processed one at a time. Processing samples in batches is clearly much more efficient in very large systems. Processing samples in batches of as little as 10 samples in small systems lead to a 45% reduction in processing time compared to processing samples one at a time (refer to figure 8.2).

The approximate number of function calls as a function of the number of samples can now be determined from the following equation:

$$N_{fc} \approx \frac{N_s * N_C * k}{N_{sb}}$$

Where N_{fc} is the number of function calls, N_s is the number of samples, N_C is the number of converters in the system and N_{sb} is the amount of samples processed in a single call to process. The constant value k is the number of function calls per process and is determined as follows:

- one function call to write per process per output port (N_{wr})
- one function call to read per process per input port (N_{rd})
- one function call to empty per process per input port (N_e)

These functions execute once for each sample in the input buffers, so it must be multiplied by the number of samples processed in a batch. There is also one function call per input buffer to size from the subcontroller to determine the input buffer size as well as one iterator increment operation per converter. empty is also called one time more than the number of samples in a batch. Note that this is only a rough estimate of the number of function calls and only serves to indicate the approximate function call overhead of the system that is not related to signal processing.

As an example, consider the case where samples are processed one at a time for converters that have one input and one output port:

$$\begin{aligned} k &= N_{sb}(N_{wr} + N_{rd} + N_e) + 3 \\ &= 1(1 + 1 + 1) + 3 \\ &= 6 \end{aligned}$$

The number of function calls to process 1 million samples in a system with 100 converters is now approximately

$$\begin{aligned} N_{fc} &\approx \frac{10^6 * 100 * 6}{1} \\ &= 600,000,000 \end{aligned}$$

Now consider the case where samples are processed in batches of 100 samples:

$$k = N_{sb}(N_{wr} + N_{rd} + N_e) + 3$$

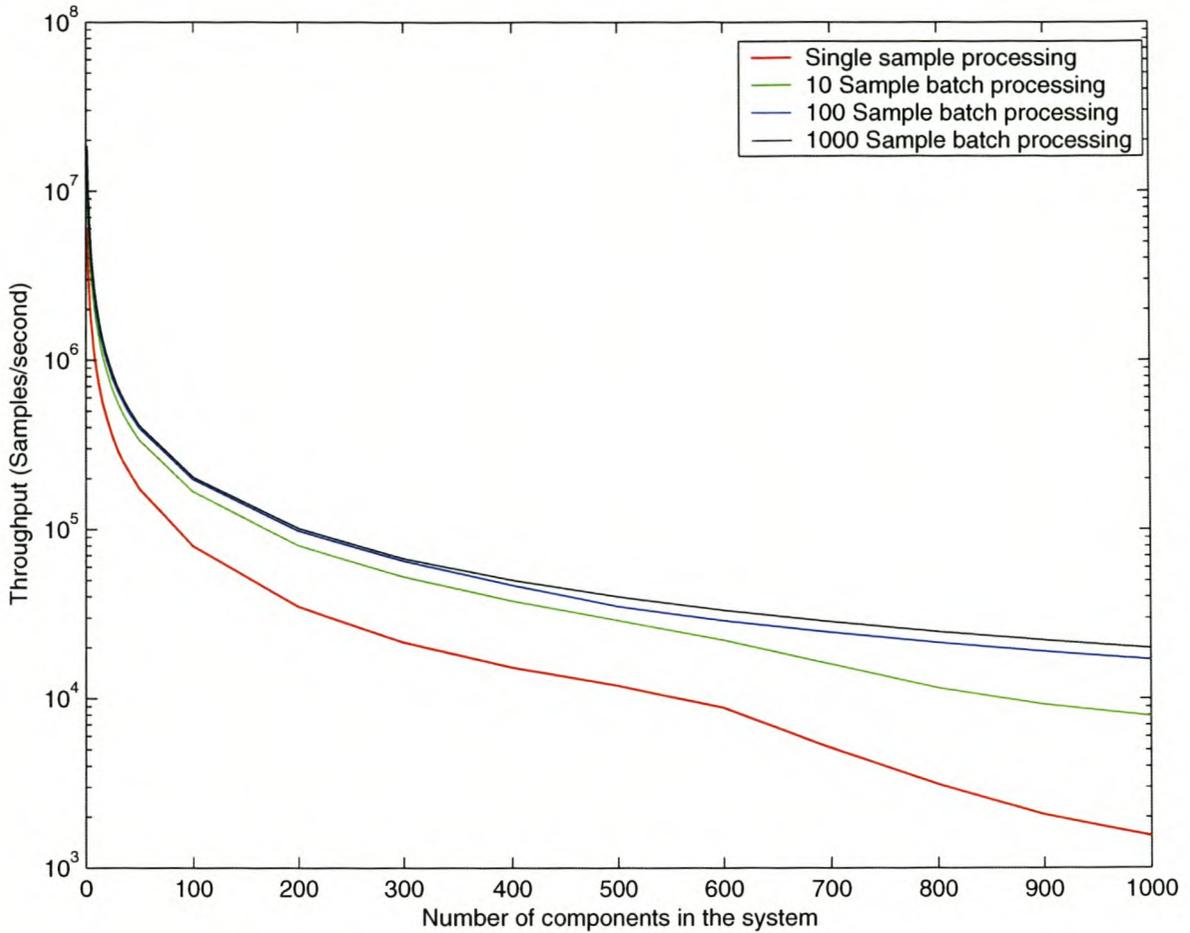


Figure 8.3: Data throughput rate of the test system

$$= 100(1 + 1 + 1) + 3$$

$$= 303$$

$$N_{fc} \approx \frac{10^6 * 100 * 303}{100}$$

$$= 303,000,000$$

which is about half of the amount of function calls in the case of single sample processing.

In figure 8.4 the processing time is plotted for systems 1, 3 and 4. Systems 1 and 4 are fairly equivalent, and the difference is mainly due to the fact that system 1 has a 70 MHz higher clock speed than system 4. The large difference between systems 1 and 4 and system 3 is due to the fact that the latter system has only 128 kB level 2 cache memory while systems 1 and 4 both have 256 kB level 2 cache memory. Systems 1 and 4 can therefore accommodate larger systems without excessive data swapping between the main memory and the cache memory.

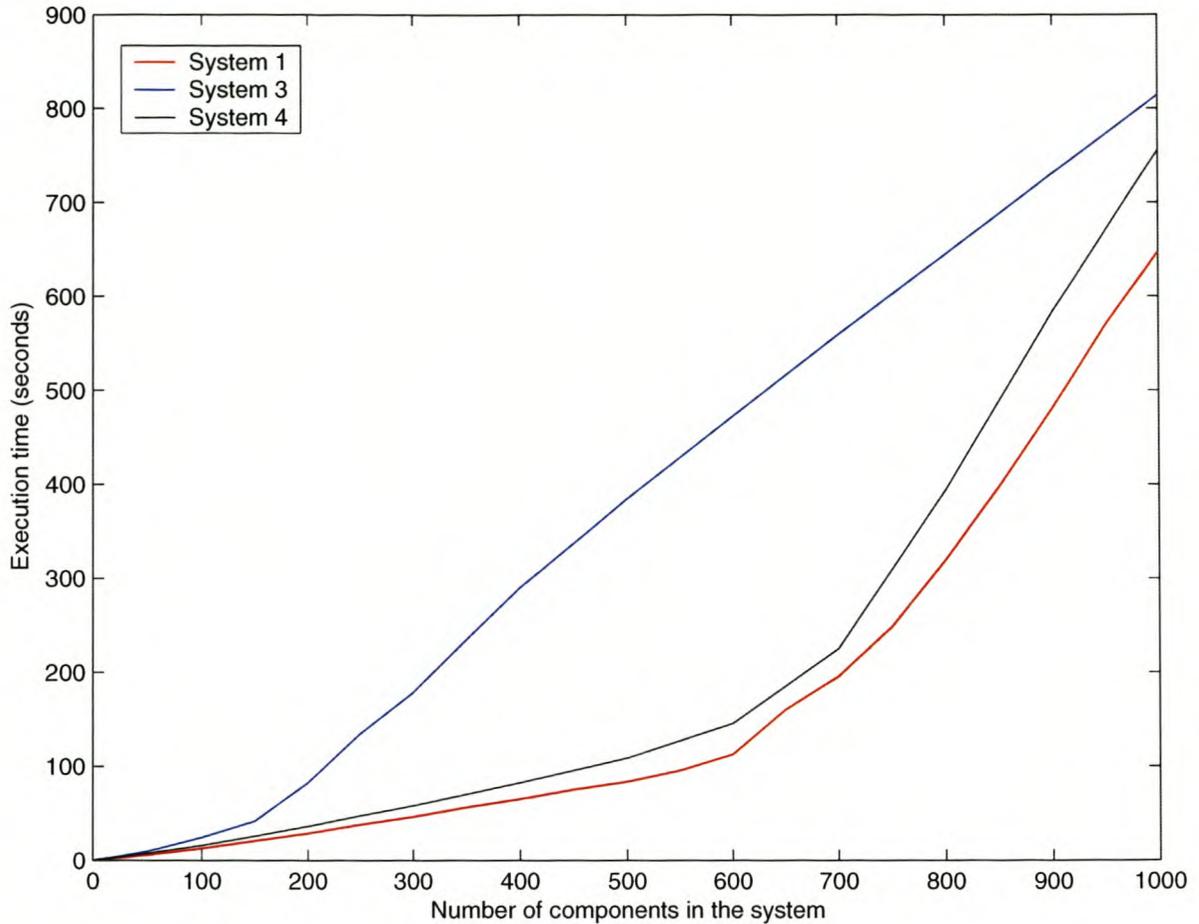


Figure 8.4: Execution times of processing 1 million samples on various systems. Refer to Table 8.1 for details of these systems.

For large systems the graphs of the processing times of the three systems converge to a line with approximately the same slope as that of the graph of system 3. This suggests that the memory access times of the three systems are fairly equivalent.

8.2.4 The effect of attribute modification overhead on data throughput performance

In this section the effect of the attribute mechanism on the data throughput performance is discussed. System 1 was used for these tests and the software radio was configured exactly the same as for the tests in the previous section. Some overhead was induced by periodically modifying the attributes of the components.

The tests were run as follows: one attribute of a converter was modified and then the system waited for a small amount of time before modifying an attribute of the next converter. This process was repeated for as long as the system was busy processing

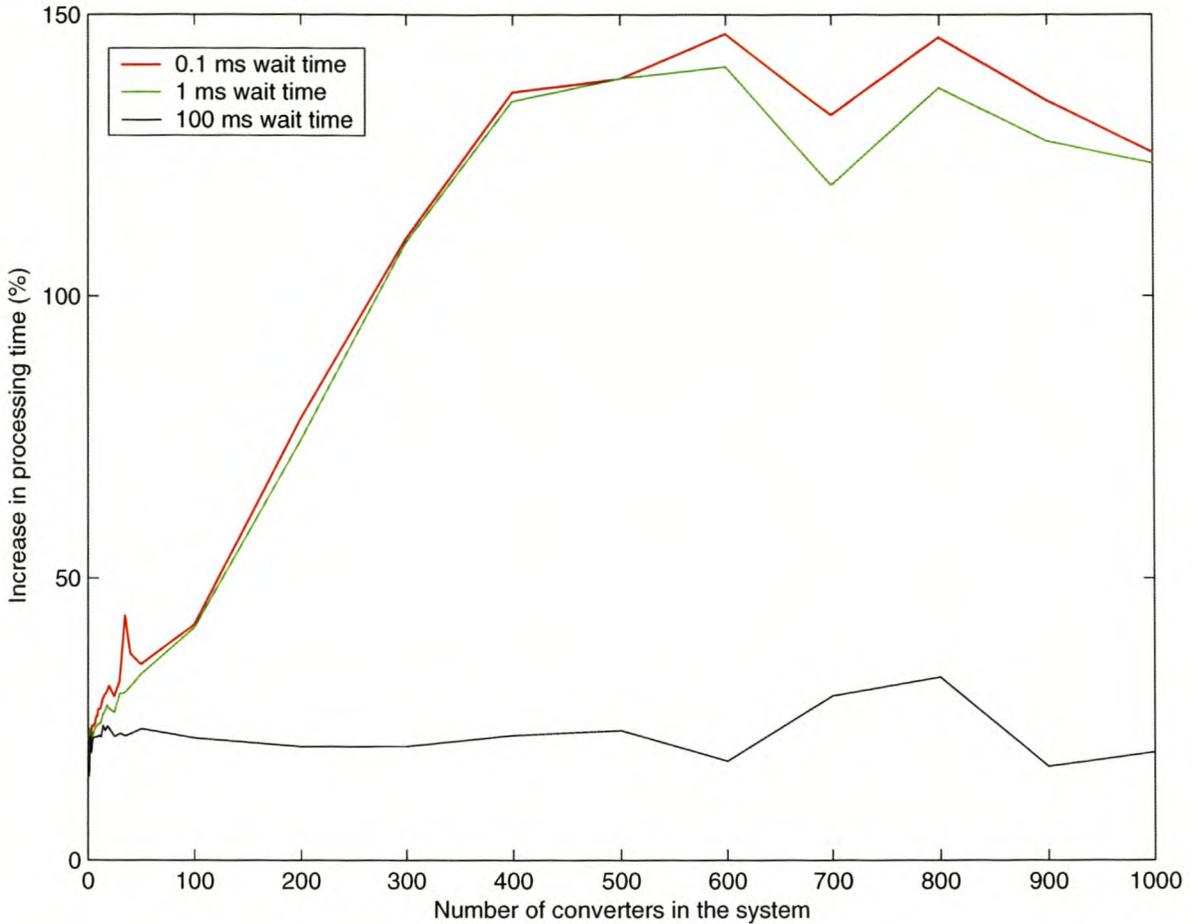


Figure 8.5: Increase in processing times of 1 million samples with attribute modification overhead

samples. The tests were run with waiting times of 100 μ s and 1000 μ s and 100 ms respectively.

The effect of the frequent attribute modifications on processing time can clearly be seen in figure 8.5. This is mostly due to the fact that the attribute modification method executes in a separate thread that is also competing for CPU time. Due to the large number of attribute modifications the CPU has to continuously switch between the signal processing thread and the attribute modification thread, thereby increasing processing time.

A separate set of tests were also carried out to evaluate the effects of single CORBA calls on processing time. This offers a better reflection of how the system would normally be used, that is, attributes are only modified once (or very infrequently). Under these conditions CORBA calls had no noticeable effect on the processing time, indicating that updating converter parameters is done in a very efficient manner.

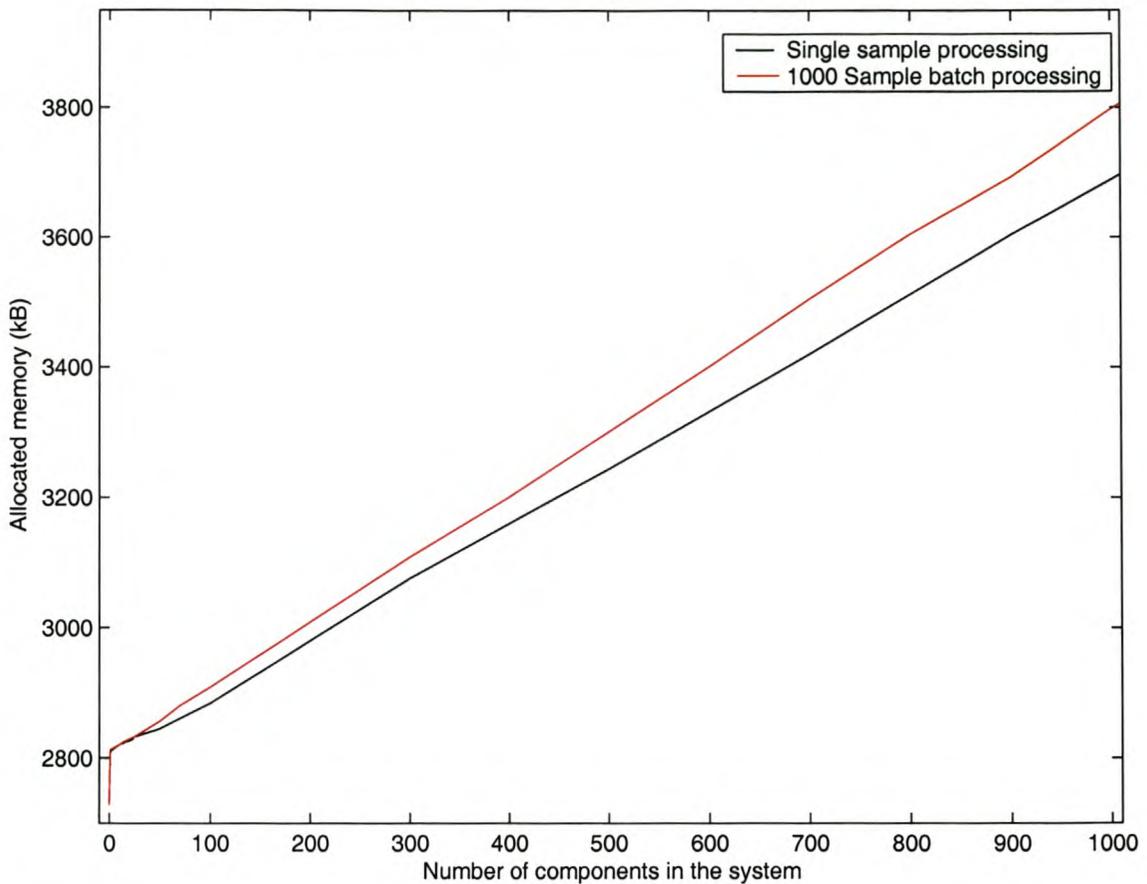


Figure 8.6: Allocated memory as a function of the number of components in the system

8.3 Memory usage

The amount of memory consumed by the system is another important aspect of the system that was investigated. The amount of memory allocated to the system was measured as a function of the number of components in the system.

8.3.1 Purpose of the experiment

This experiment was performed to investigate the total amount of memory allocated to the system as a function of the number of components in the system. It was used to evaluate the memory efficiency of the system and to confirm that the allocation of memory behaves as expected.

Theoretically, the memory used by the subcontroller subsystem should consist of a fixed part that is taken up by the executable. This part is also called the code size or *text resident size* [28]. Note that this part excludes the subcontroller object, the ORB mechanism and the converter components. The operating system has to allocate

additional memory for each component as components are added to the subcontroller. Theoretically, the memory allocated should increase linearly as more components are added to the system because the component objects have fixed sizes. The input buffers are the only components of the system that have variable memory requirements, but with the current subcontroller configuration the size of the input buffers is limited to 8192 samples. The part of memory that contains the stack and all the program data is called the *data resident size*. The sum of the text and data resident sizes present in memory is called the *resident set size*.

8.3.2 Experimental setup

The memory allocation was monitored with a shell script utility that obtains the memory information from the `proc` file system provided by the Linux 2.4 kernel. The allocated memory as reported by the shell script reflects the operating system's view of the memory occupied by the subcontroller. The resident set size was measured because it indicates the total amount of memory allocated to the program that is resident in memory.

8.3.3 Results of the experiment

A plot of the amount of allocated memory as a function of the amount of converters in the system is shown in figure 8.6. It shows that the amount of memory that is allocated to the system increases linearly as the amount of converters in the system increases. The difference between the two curves is due to the fact that the input buffers are larger when the samples are processed in batches than when samples are processed one at a time.

It is also clear from the graph that the initial resident size of the system is approximately 2800 kB. This includes only the ORB and subcontroller objects. The text resident size of the subcontroller executable was measured to be approximately 488 kB, therefore the initial data resident size is approximately 2312 kB. When converters are added to the system, they form part of the data resident size.

8.4 Latency measurements

Another important characteristic of the system is the amount of latency caused by the finite processing time of the processor.

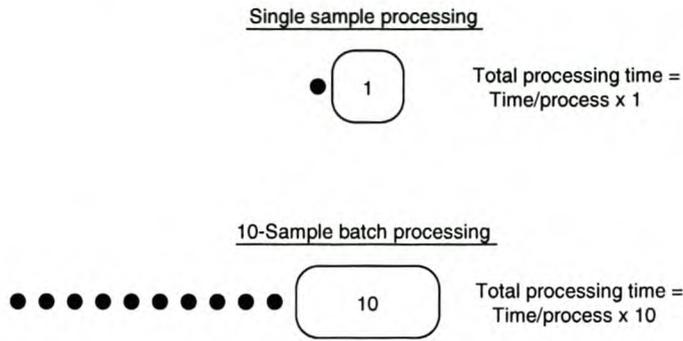


Figure 8.7: Latency in a system due to batch processing

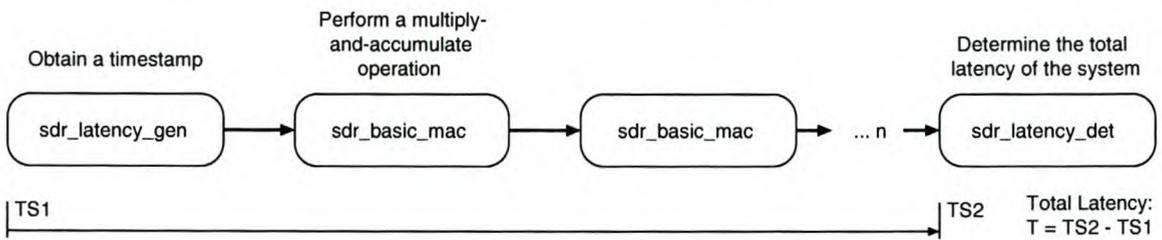


Figure 8.8: The converters used to measure the latency of the software radio system

8.4.1 Purpose of the experiment

It was shown in section 8.2 that the throughput rate can be increased significantly by processing samples in batches. This approach has a drawback—as the size of the batches increase, the amount of latency of the system also increased.

Intuitively it makes sense that the latency should be higher in the systems that employ batch processing. This is due to the fact that the process method of each component will take longer to finish when samples are processed in a batch than when samples are processed individually. Refer to figure 8.7 for a graphical representation of this concept. Note that, although the latency is increased, the total throughput is also increased because the number of function calls are reduced, and this in return also reduces the amount of swapping of executable code between the processor cache and main memory.

8.4.2 Experimental setup

Two specialised converters were created to perform this experiment. The first was a converter that simply obtained a timestamp from the system clock and then this value was propagated through the system. The second converter then also obtained a timestamp from the system and then computed the latency as the difference between the two times. Modified MAC converters were placed between these two converters

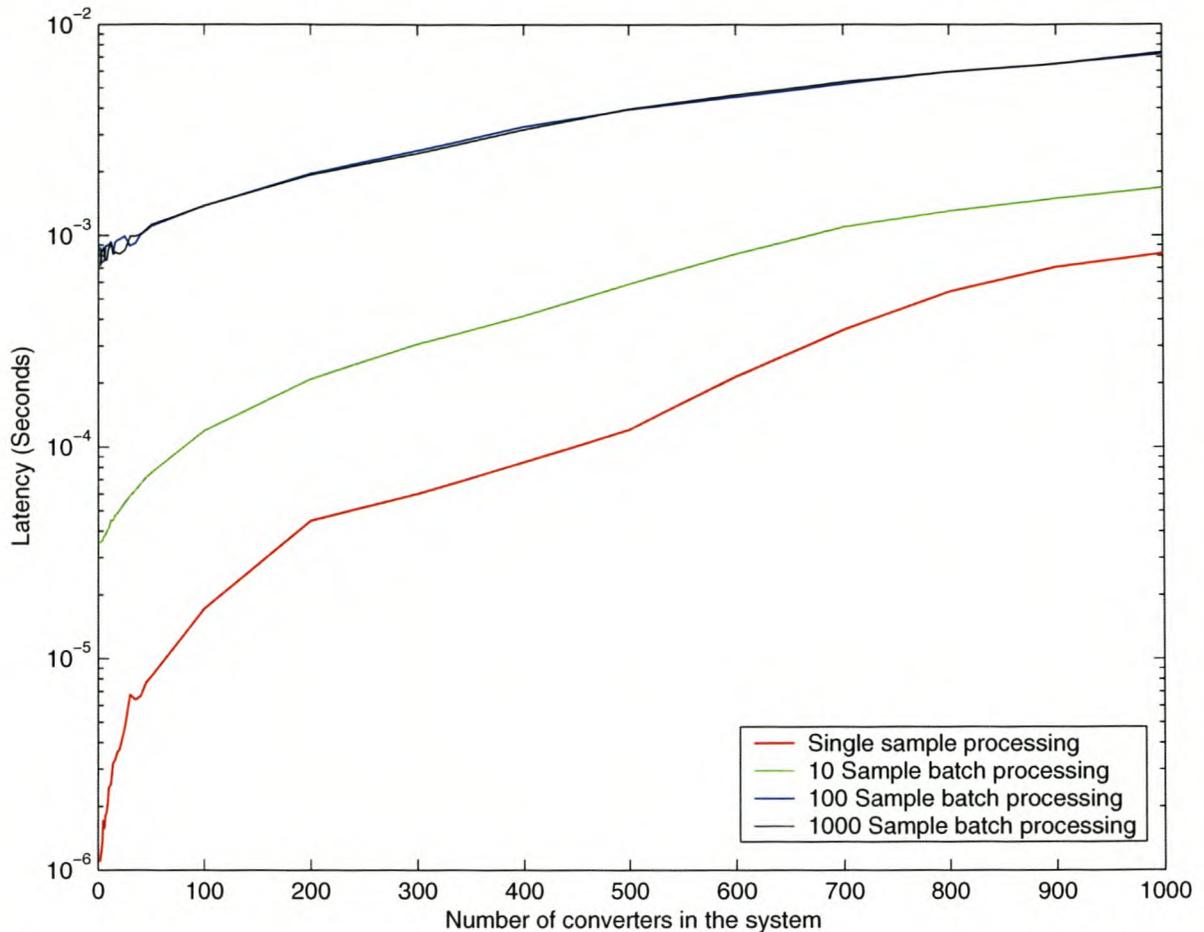


Figure 8.9: *The latency of the system as a function of the number of components*

to act as the components of a real system. The MAC converters were modified to not write out the result of the multiply-and-accumulate operation, but to write out the input value to the output port without modification (after a MAC operation was carried out on the input value). Refer to figure 8.8 for a graphical representation of the system.

The latency was measured for systems with individual sample processing as well as 10-sample, 100-sample and 1000-sample batch processing.

8.4.3 Results of the experiment

Graphs of the latency of such systems as a function of the amount of components in a system is shown in figure 8.9. To get a better idea of the difference in latency between individual sample processing and batch processing, the time axis (vertical) was plotted on a logarithmic scale.

The figure indicates that for small systems the difference between the latency in systems that employ individual sample processing and systems that employ batch processing is about of 3 orders of magnitude. The difference decreases to about 2

orders of magnitude for medium-sized systems and to about 1 order of magnitude for large systems.

8.5 RF performance evaluation

Until now, all the tests that were performed on the system focused on the processing abilities and requirements of the system. Another crucial aspect of the system is its performance in the analogue domain. In this section the conversion from the analogue to the digital domain is evaluated along with the performance of the system in a real-world application.

8.5.1 Purpose of the experiment

The purpose of this experiment was to determine the analogue characteristics of the system. This was done to determine if the system is suitable for use in real-world telecommunications systems.

8.5.2 Experimental setup

System 2 was used to perform the modulation tests. The digital-to-analogue conversion was done with ordinary 16-bit soundcards with a sampling rate of 44.1 kHz. The first set of tests were performed with the on-board soundcard. The on-board sound system uses the C-Media CM8738 audio controller IC.

A quadrature baseband frequency modulator converter was used to frequency modulate the data that was captured with the sound card. This signal was then output to the soundcard and mixed up to 100 MHz with a Rohde & Schwarz vector signal generator (SMIQ). The output of the SMIQ was connected to a spectrum analyser to investigate the spectrum of the resulting RF signal. The FM signal was received and demodulated with a small hand-held FM radio to verify that the system was indeed transmitting a FM signal.

A PCI sound card was then installed in the computer for comparative tests. The second card was a Creative Labs CT4810 with a Creative CT2518 audio controller IC. The same signal source was used for these tests.

8.5.3 Results of the experiment

The spectrum obtained with the C-Media audio controller is shown in figure 8.10. The spectrum shows the modulated message signal around 100 MHz as expected, but the spectrum also shows unwanted images of the signal spaced about 44 kHz apart. This

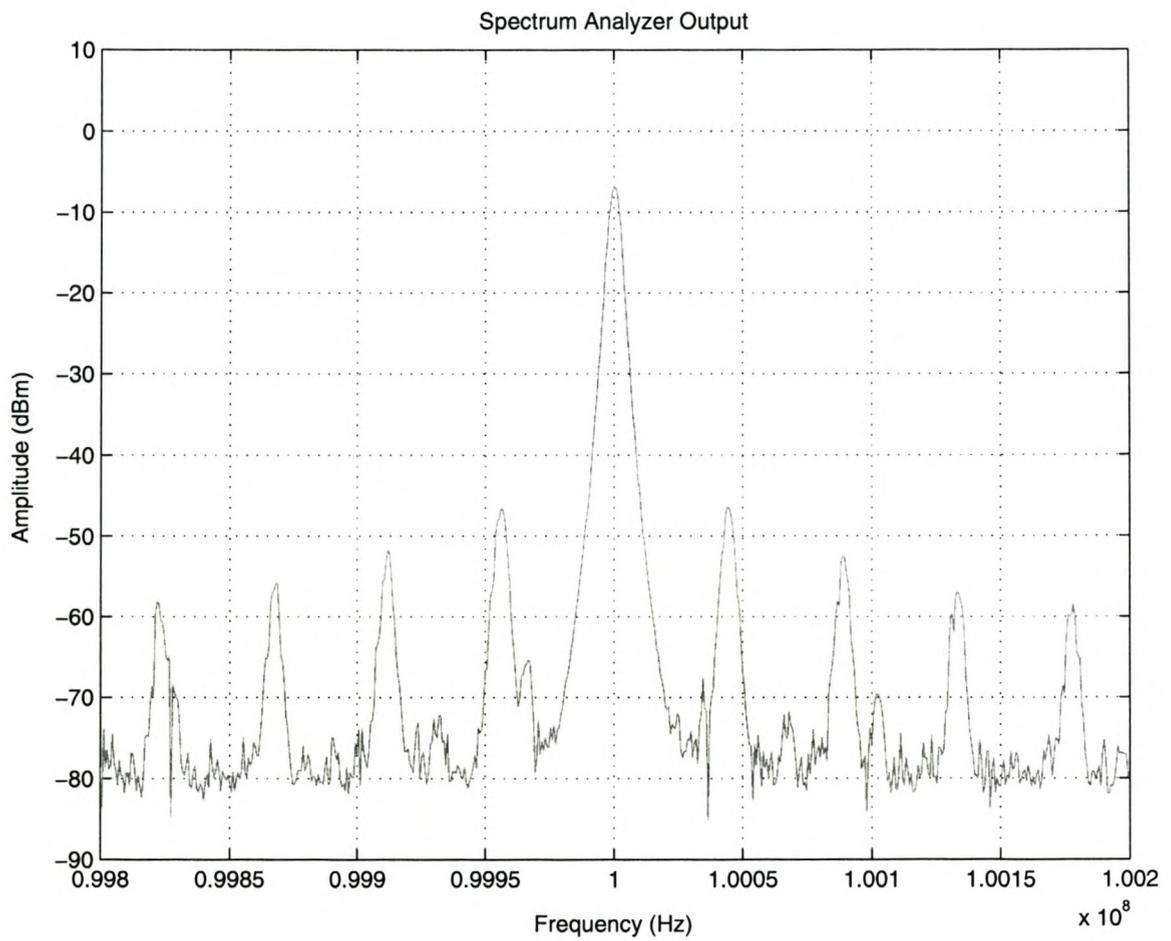


Figure 8.10: Spectrum of the frequency modulated RF signal that was generated with the C-Media audio controller.

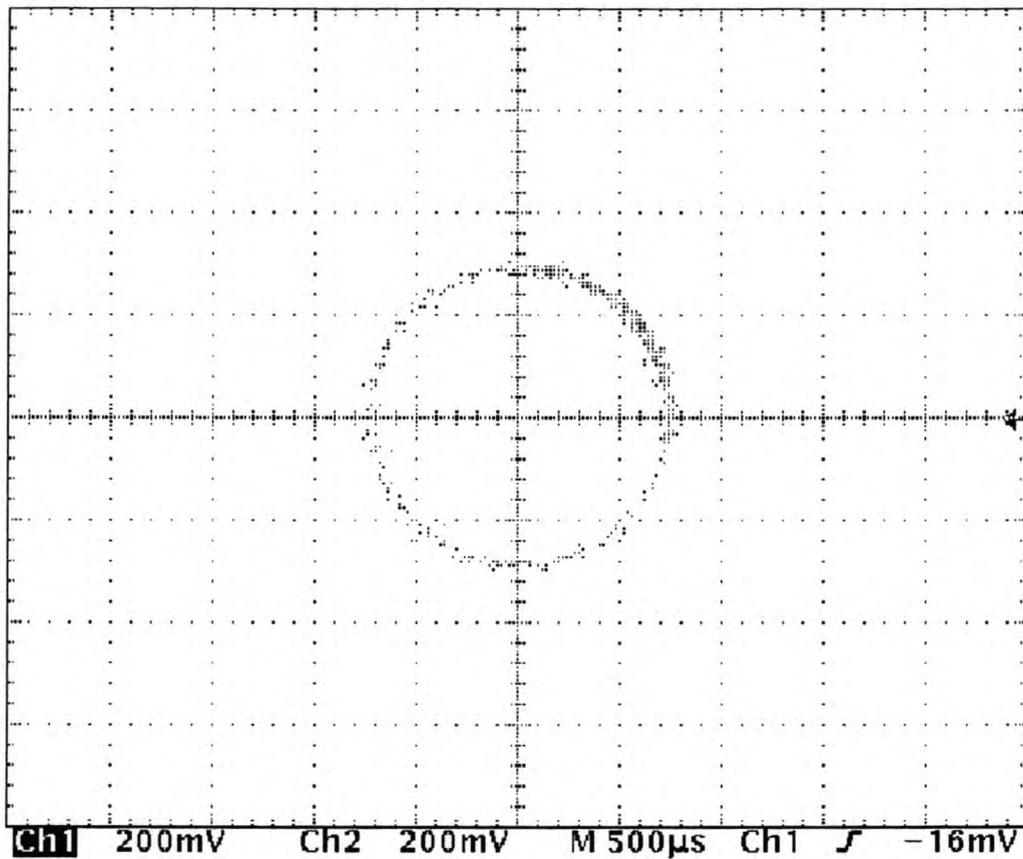


Figure 8.11: Plot of in-phase versus the quadrature component of the baseband FM signal

is due to the fact that the on-board sound system lacks a low-pass filter to suppress higher order harmonics caused by the digital-to-analogue conversion process. From the figure it can be seen that the dynamic range between the desired signal and images of the signal is about 30 dB. This is not a desired effect as it degrades the signal quality and the images may also interfere with adjacent FM signals.

The spectrum obtained with the Creative Labs audio controller is shown in figure 8.12. The spectrum shows the modulated message signal around 100 MHz, with a bandwidth of about 20 kHz. The output does not have any images in the spectrum, indicating that the Creative Labs sound card features a low-pass filter in its output stage. From the figure it can be seen that the *spurious-free dynamic range* (SFDR) of the signal is about 60 dB. This compares very well with the results obtained with first card.

To illustrate the modulation characteristics of the system a single sinusoid with a frequency of about 500 Hz was generated and modulated in software. The in-phase and quadrature components of the signal generated by the soundcard is shown in figure 8.11. The circle indicates that the generated signals are, in fact, phase-shifted by 90 degrees. The spectrum of the baseband FM signal is shown in figure 8.13a.

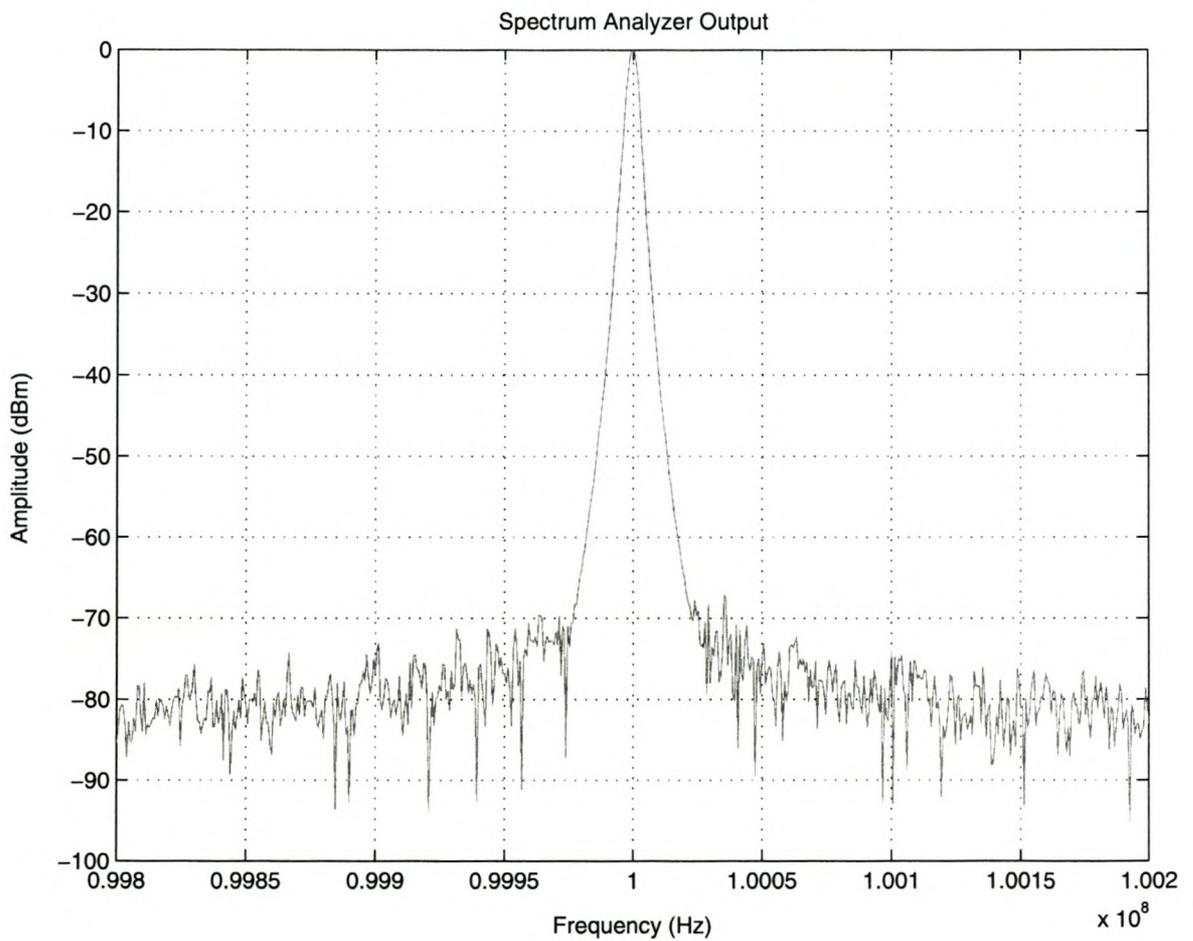


Figure 8.12: Spectrum of the frequency modulated RF signal that was generated with the Creative Labs audio controller

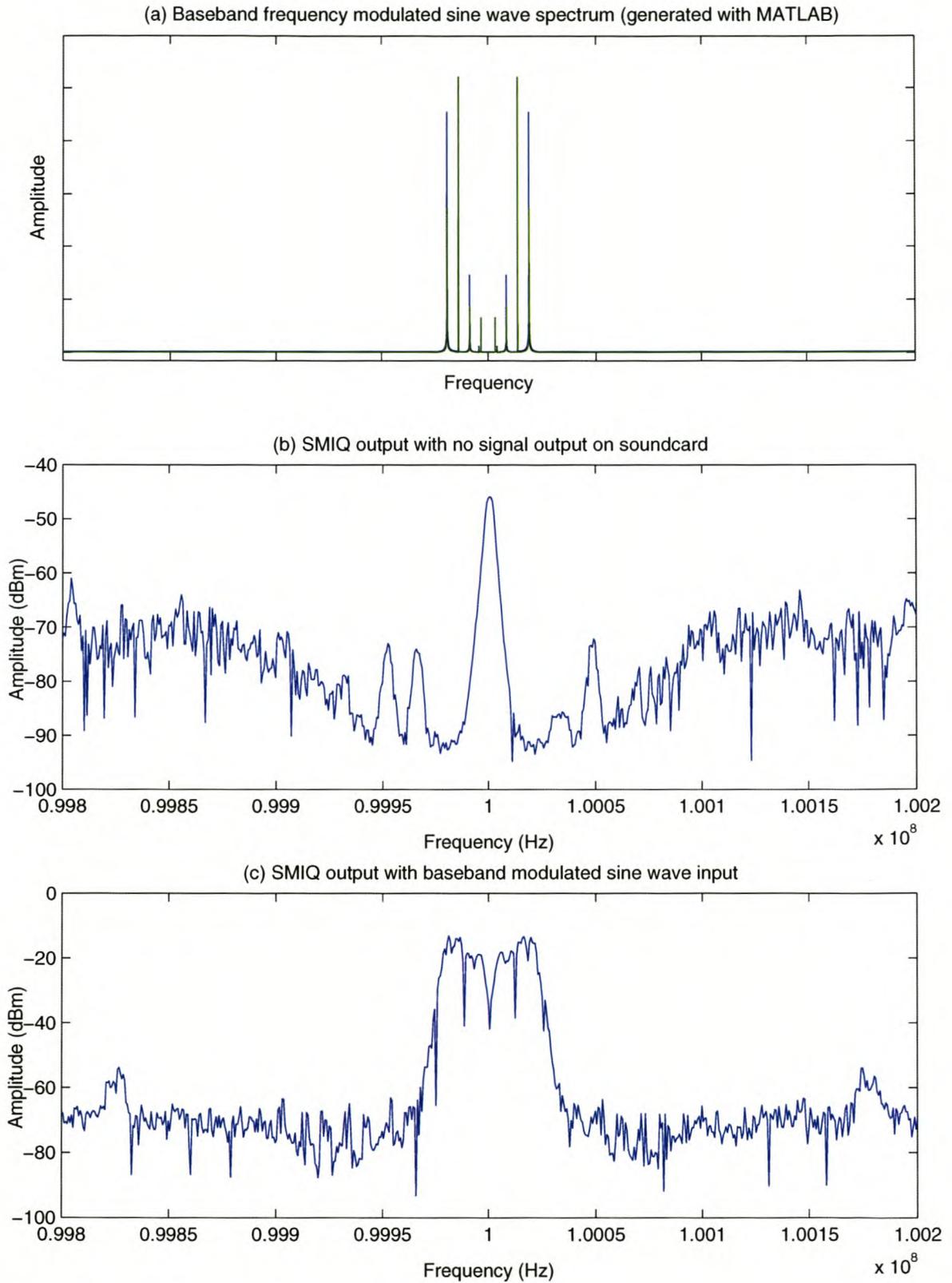


Figure 8.13: Three spectrum plots showing the spectrum of a baseband frequency modulated sine wave signal generated by the Creative Labs audio controller. (a) The frequency modulated baseband signal spectrum. (b) The spectrum of the SMIQ output with no signal output from the sound card. (c) The resulting frequency modulated signal.

The spectrum is as expected from [31]. This signal was then fed to the SMIQ, and the resulting spectrum is shown in figure 8.13c. It is clear from figure 8.13c that the resulting signal is the convolution of the baseband FM signal (figure 8.13a) and the carrier signal (figure 8.13b). This signal has a SFDR of about 40 dB, measured between the peak of the desired signal and the peaks at $f_c \pm 176$ kHz.

8.6 Conclusion

In this chapter the software defined radio system was evaluated to determine the performance of the current architecture. The test results revealed factors such as the data throughput rate, memory allocation and latency as a function of the number of components in the system. A small RF test was also carried out to investigate the performance of the system in the analogue domain.

The rate of data throughput is obviously dependent on the speed of the processor as well as the amount of cache memory of the processor, the amount of RAM in the system as well as the number of processes running on the system, so these tests only serve as a general indication of the performance of the system. These tests indicate important characteristics of the system such as the difference in execution times between different processors as well as the difference between batch processing of samples and processing samples individually.

As with most engineering problems the choice between the individual sample processing approach and the batch processing approach represents a trade-off. If the intended system must be able to handle high data throughput but it can tolerate some latency the best approach is to use batch processing. On the other hand, if a very low latency is required and data throughput is not critical, individual sample processing is the better approach. The choice is obviously also dependent on the size of the software radio system due to the fact that the processing time of large software radio systems is much longer than the processing time of small software radio systems.

The configuration of the system is a further factor that determines whether batch processing can be used; systems that employ feedback, for example, cannot make use of batch processing because the samples that are being fed back will always be lagging by a factor equal to the batch size. Consider the system shown in figure 8.14. Converter A will process the 100 samples that it receives on its first input port. The resulting data will then be passed on to converter B. It will then process the batch and pass the data on to converter C. After converter C has finished processing the data, it will be passed on to the next stage and to the second input port of converter A. In the next process cycle, converter A will process the data in its first input buffer along with the data in its second input buffer from the previous run. The samples that are fed back to converter

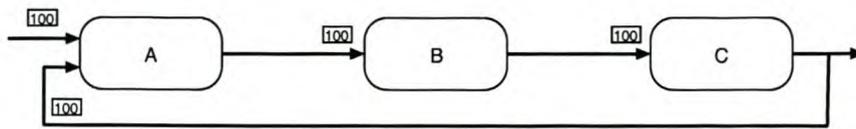


Figure 8.14: Feedback lag in a system that employs batch processing

A is therefore lagging the input by 100 samples.

The latency and throughput of the system is plotted in figure 8.15. The coloured lines each represent a case where fixed numbers of samples was processed in batches. The dotted lines represent fixed numbers of converters in a system. This graph can be used to determine the amount of latency and the throughput for a specific system. It can also be used to determine, for example, the maximum attainable throughput for a fixed amount of latency as a function of the batch size. The figure also shows that there is not much difference between 100 and 1000 sample batch processing. Batches of 100 samples are therefore adequate for achieving high throughput rates, without the large memory requirements of 1000 sample batches.

The classification software radio systems used in this document can be summarised as follows:

- small - approximately equivalent to 100 or fewer MAC converters
- medium - approximately equivalent to between 100 and 500 MAC converters
- large - approximately equivalent to more than 500 MAC converters

where one MAC converter represents one multiplication operation, one addition operation and one assignment operation per sample, not counting sample transfer operations such as reading or writing to buffers. The FM modulator used in the RF tests are classified as a small system because it performs

- four addition operations
- five multiplication operations
- two trigonometric function operations
- one assignment operation

per process cycle. The quadrature baseband frequency modulator is therefore approximately equivalent to about 5-10 MAC converters, depending on the cost of the trigonometric functions it uses.

The RF performance of the system was quite satisfactory given that the digital-to-analogue conversion was performed with a sound card with a sampling frequency of

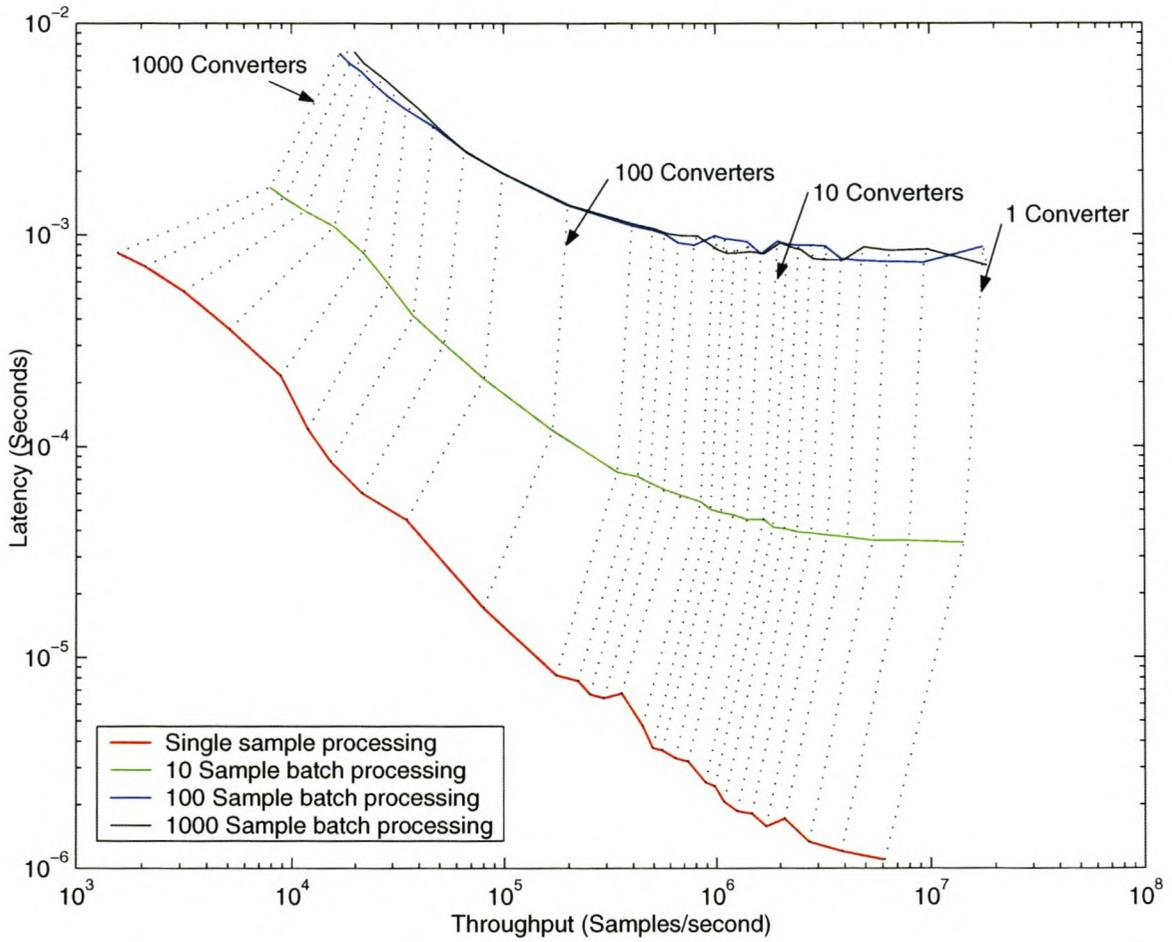


Figure 8.15: Latency of the system versus throughput

44.1 kHz. This low sampling rate mainly influences the frequency deviation of the FM signal and therefore decreases the amplitude of the demodulated message signal. This will not be a problem with a high-speed DAC that can accommodate the higher sampling rate necessary to achieve the correct amount of frequency deviation.

Chapter 9

Conclusion

This chapter provides an overview of the work presented in the previous chapters of the document. It highlights both the strengths and weaknesses of the system in its present form. Areas that require further work are also identified and discussed.

9.1 Overview of the work

This section provides a review of the system presented in the previous chapters. All the aspects of the system presented in the preceding chapters are considered and critically evaluated with respect to the design requirements listed in sections 1.3 and 4.2.

9.1.1 Other software defined radio projects

Several other software-defined radio projects were investigated to gain insight into the methods employed in these projects, and to assess their positive and negative aspects.

The research done by Bose [2] and Mackenzie and others [14] proved that implementing SDR on general-purpose platforms was indeed possible. Bose also implemented time-based scheduling akin to real-time systems on a GPP, thereby showing that execution times could be guaranteed to some extent on these platforms.

The stack-based architecture presented in [14] provides a novel way to reconfigure the functionality of their software radio. It was mentioned that the message blocks employed in this architecture could possibly introduce unwanted parsing overhead that will limit the maximum attainable throughput rate.

The modulation scheme recognition system that was developed by Nolan and others allows their software radio to be automatically reconfigured at run time [15]. This approach eliminates the need for special embedded codes in the received signal to enable the automatic selection of the correct demodulation block.

9.1.2 Hardware platforms

The hardware platforms that are available for performing signal processing were all investigated to provide an insight into the specific applications that each one is suited for. This was done more for general background, but also to identify what had to be catered for when designing the general-purpose platform architecture.

The rapid advances in general-purpose processors has enabled them to be suitable for real-time signal processing, and therefore in software-defined radio applications. The other platforms that were investigated, like FPGAs, DSPs and ASICs are still necessary to implement high-performance systems, though.

9.1.3 The current SDR platform

The well-defined interfaces between the various layers in the system ensures that there is orthogonality between the layers. This means that a layer is not dependent on the specific implementations of the other layers in the system. Any subsystem in the system can therefore be replaced with another implementation of that subsystem, provided the new subsystem adheres to the interface defined for that subsystem. This is good because if the layers were not independent of each other a small change in the implementation of one layer could propagate to all the other layers, thereby necessitating modifications to all the other layers. Clearly this behaviour is not desired.

The CORBA RPC interface enables easy implementation of distributed radio systems. It allows the user interface to reside on one computer while the signal processing is performed on separate computers, thereby allowing distributed signal processing. In the current implementation the CORBA layer is also semi-transparent to the subcontroller developer and the user interface developer, thereby allowing the focus to fall more on the subcontroller and the user interface than on the RPC mechanism.

In its current state the `sdr_converter` class does not allow the user (or developer) to nest subcontroller classes to construct compound converters. This would be especially handy, as it would enable complex converters to be placed in the component library alongside the “primitive” types.

Another shortcoming of the current system is the scheduling mechanism used in the subcontroller. The round robin approach works fairly well, but more specialised scheduling approaches may yield even better performance. A list of possible scheduling strategies, used in computer systems, can be found in [25].

The SDR Builder application provides a user interface for the system that can be used to construct working software radio systems. It has the ability to discover subcontrollers that are running on a local or remote host, it can discover the attributes of the converters that have been added, and these attributes can also be modified while the system is

processing samples. Unfortunately, the user interface does not offer the ability to save the current radio system to non-volatile memory, which severely limits its functionality. It also does not offer graphical feedback as to the current configuration of the system, which is another severe shortcoming. Another feature that the user interface lacks is the ability to discover the subcontroller configuration in the event of a GUI malfunction. At the moment, the system keeps duplicate configuration information in the subcontroller and the GUI, but there is no way to keep this information synchronised.

The main objective of this thesis was to provide a well-defined software architecture to provide a basis for constructing functional software defined radios. This design then had to be evaluated and its capabilities demonstrated with a basic functioning system. The current system meets both these requirements. It provides a reasonably well-designed converter layer, as well as functional subcontroller and control application layers. The performance of the architecture has been measured and a rudimentary frequency modulator was implemented to demonstrate the functioning of the system in the analogue domain. The current architecture does require some more refinement and added functionality, though. The areas that need further development will be shortly discussed in the next section.

9.2 Future Research

A few areas that will require additional research have been identified through the course of the project. One of the most important features that needs to be implemented is a scheduling scheme for the subcontroller. Bose [2] gives a good overview of how the computational variability of a general-purpose platform can be modelled and how to implement a real-time scheduling mechanism on a general-purpose platform. A study can also be made to determine if a multi-threaded subcontroller implementation is feasible.

A second important aspect that needs to be implemented is a dynamic component library system. This system must have the ability to add components to the system's library without the need to recompile or re-link anything as compiling and linking imply that the system needs to be restarted to make use of new functionality.

At the moment the configuration of a radio cannot be saved, in other words, if SDR Builder and the subcontrollers are shut down, the radio has to be reconfigured from scratch. To be able to save the configuration or the state of a radio a format is needed to store a description of the radio in. This will also enable various radio configurations to be saved so that a collection of different software radios can be built up. A strong candidate for this purpose is the *extensible markup language* (XML).

Another issue that was identified during the course of the project is the need for

an easy way to create new converters without having extensive knowledge of the C++ programming language. A description of a converter can be written in a meta-language and then translated to C++. The meta-language compiler must also be able to translate the meta-language description to other languages like C or assembly language.

Bibliography

- [1] BLUG, "Bergen Linux User Group." <http://www.blug.linux.no/rfc1149/>. April 2001.
- [2] BOSE, V., *Design and Implementation of Software Radios Using a General Purpose Processor*. PhD thesis, Massachusetts Institute of Technology, June 1999.
- [3] BRAUN, D., SIVILS, J., SHAPIRO, A., and VERSTEEGH, J., "UML Tutorial." http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm. 2001.
- [4] CUMMINGS, M. and HARUYAMA, S., "FPGA in the Software Radio." *IEEE Communications Magazine*, February 1999, pp. 108–112.
- [5] Freesoft, "OSI Seven-Layer Model." <http://www.freesoft.org/CIE/Topics/15.htm>. April 1997.
- [6] GALYON, E., "C++ vs Java Performance." <http://www.cs.colostate.edu/~cs154/PerfComp/>. 1998.
- [7] GAMMA, E. *et al.*, *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- [8] GEPPERT, L., "The Amazing Vanishing Transistor Act." *IEEE Spectrum*, October 2002, pp. 28–33.
- [9] GROUP, O. M., "About The OMG." <http://www.omg.org/news/about/index.htm>. March 2003.
- [10] HENTSCHEL, T. and FETTWEIS, G., "Sample Rate Conversion for Software Radio." *IEEE Communications Magazine*, August 2000, pp. 142–150.
- [11] HENTSCHEL, T., HENKER, M., and FETTWEIS, G., "The Digital Front-End Of Software Radio Terminals." *IEEE Personal Communications*, August 1999, pp. 40–46.

- [12] IETF, "RFC 1149." <http://www.ietf.org/rfc/rfc1149.txt?number=1149>. April 1990.
- [13] ISO, "Overview." <http://www.iso.ch/iso/en/aboutiso/introduction/index.html#four>. April 2003.
- [14] MACKENZIE, P., DOYLE, L., O'MAHONY, D., and NOLAN, K., "Software Radio on General-Purpose Processors." in *Proceedings of the First Joint IEI/IEE Symposium on Telecommunications Systems Research*, November 2001.
- [15] NOLAN, K., DOYLE, L., O'MAHONY, D., and 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.
- [16] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, December 2001. From <http://www.omg.org/cgi-bin/doc?formal/01-12-01>.
- [17] OBJECT MANAGEMENT GROUP. *Naming Service Specification*, 1.2 edition, September 2002. From <http://www.omg.org/cgi-bin/doc?formal/02-09-02>.
- [18] PLAUGER, P., "Embedded C++: An Overview." <http://www.embedded.com/97/feat9712.htm>. December 1997.
- [19] PUCKER, L., "CommsDesign - Paving Paths to Software Radio Design." http://www.commsdesign.com/csdmag/sections/cover_story/OEG20010521S0118. June 2001.
- [20] READ, J. H., *Software Radio: A Modern Approach To Radio Engineering*. Prentice Hall, 2002.
- [21] ROSENBERGER, J., *Teach Yourself CORBA In 14 Days*. SAMS Publishing, 1998.
- [22] SDR Forum, "SDR Forum FAQs." <http://www.sdrforum.org/faq.html>. January 2004.
- [23] SGI, "Standard Template Library Programmer's Guide." <http://www.sgi.com/tech/stl/>. 1994.
- [24] STROUSTRUP, B., *The C++ Programming Language*. Third edition. Addison Wesley Longman, 1997.
- [25] TANENBAUM, A., *Modern operating systems*. Prentice Hall, 1992.

- [26] Vanu, Inc., "Vanu, Inc." http://www.vanu.com/pressreleases_030512.html. May 2003.
- [27] VOGEL, A. *et al.*, *C++ Programming with CORBA*. Wiley Computer Publishing, 1999.
- [28] WARNER, J., *TOP(1) Manual Page (Linux process lister)*, *SuSE Linux 8.2*, September 2002.
- [29] Xilinx, Inc., "Virtex-E FAQ." http://www.xilinx.com/prs_rls/vtxefaq.htm. September 1999.
- [30] Xilinx, Inc., "Xilinx PAVE Framework Plus ISE Software and Synplicity Synthesis Turbocharge Field Upgrades." http://www.xilinx.com/prs_rls/0183turbo.html. September 2001.
- [31] ZIEMER, R. and TRANTER, W., *Principles of Communications*. Fourth edition. John Wiley & Sons, Inc., 1995.

Appendix A

Class `sdr_converter`: API Overview

A.1 API Reference Documentation

Table A.1 contains a list of all the methods of class `sdr_converter`. It lists all the public and protected methods that can be used in a derived class, as well as all the public and protected pure virtual methods that must be implemented in the derived class.

It is included here as a handy reference for developers. This information is also available in *The University of Stellenbosch Software Defined Radio Project: Converter Application Programming Interface (API) Documentation* by Johan Cronjé, which is available with the SDR library software.

Table A.1: Methods Available in Class `sdr_converter`

Name	Parameters	Return Type	Description
Public Methods			
<code>properties</code>	-	string	This function is used to obtain a list of the attributes of a converter class.
<code>get_attribute</code>	string <code>attribute_name</code>	attribute_struct&	This function is used to obtain the value of an attribute
<code>set_attribute</code>	string <code>attribute_name</code> string <code>attribute_value</code>	-	This function is used to modify the value of an attribute
<code>read_input_port</code>	int <code>port_number</code>	RCPtr<sdr_buffer_base>	Returns a pointer to the input buffer <code>port_number</code> .
<code>write_to_buffer</code> (templated)	int <code>port_number</code> T <code>sample</code>	-	This function is used to write the <code>sample T</code> to the buffer of input port <code>port_number</code> .
<code>ready</code>	int <code>port_number</code>	bool	Checks if there are any samples in input buffer <code>port_number</code> .
<code>buffer_size</code>	int <code>port_number</code>	int	Returns the size of the input buffer <code>port_number</code> .

<code>link</code>	<code>int output_port_number</code> <code>RCPtr < sdr_converter > destination_module,</code> <code>int destination_portnumber</code>	-	Links the output port <code>output_port_number</code> to the input port <code>destination_portnumber</code> of the converter module <code>destination_module</code> .
Public Pure Virtual Methods			
<code>process</code>	-	-	This method defines the functionality of the converter and must be implemented in all derived classes.
<code>is_a</code>	-	<code>const char*</code>	This method is used to identify the converter at run time.
Protected Methods			
<code>add_port</code>	<code>int porttype</code>	<code>int</code>	This function is used to add ports to the converter.
<code>write_output_port</code> (templated)	<code>int port_number</code> <code>T sample</code>	-	write the sample <code>T</code> to the output port <code>port_number</code> .
<code>add_attribute</code>	<code>string attribute_name</code> <code>string attribute_value</code> <code>string attribute_limit</code> <code>string attribute_description</code> <code>int access</code>	-	This function is used to add attributes to the attribute table of the converter
Protected Pure Virtual Methods			
<code>apply</code>	<code>string attribute_name</code> <code>string attribute_value</code>	<code>int</code>	This function is used by <code>set_attribute</code> to change the values of the member variables. It must be defined in the class implementation

Appendix B

The Unified Modelling Language

The *Unified Modelling Language* (UML) is a very versatile tool used for OOA. It was created by Rational Software Corporation in 1996, and is currently managed by the Object Management Group (refer to section 6.2).

In order to facilitate understanding the design methodology used in this document, a very short overview of UML is given here.

B.1 UML Class Diagrams

UML Class Diagrams are used to model the structure and contents of classes and is also used to show how classes are related to each other [3]. A basic UML class diagram is shown on the left in Figure B.1. It shows the various fields that can be described by class diagrams.

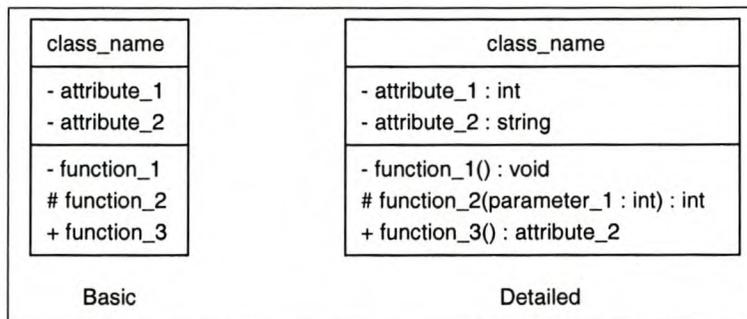


Figure B.1: *Two UML Class Diagrams Showing Attribute And Function Types*

The top block in the class diagrams illustrated in Figure B.1 holds the class name. This is the name that the class will have in the source code. The middle block lists all the data members that the class contains. All the private, protected and public data members are listed here. The bottom block lists all the member functions that the class contains. All private, protected and public functions are listed here, as with the data members.

The diagram on the right side shows a more detailed class diagram. In this type of class diagram the types of the data members and member functions are also specified. Unlike some programming languages (like C++), the return types of a function are listed *after* the function name and parameters.

The -, + and # characters are known as visibility modifiers [21]. These symbols are used to specify the visibility of data members and member functions from outside the class. A list of available modifier symbols is given in Table B.1.

Symbol	Description
+	This is used to indicate a public data member or function
#	This is used to indicate a protected data member or function
-	This is used to indicate a private data member or function
/	This is used to indicate that a data member is dependant on at least one other data member.
\$	Used to indicate that this member is accessible without a class instance; in C++ this translates to static data members and functions

Table B.1: UML visibility modifiers

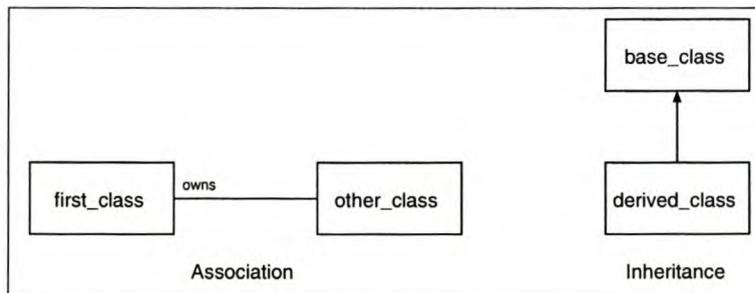


Figure B.2: Two UML class diagrams showing association and inheritance

From Table B.1 it is clear that class `class_name` contains two private data members, one private member function, one protected member function and one public member function.

The last types of UML class diagrams that will be discussed here are the association and inheritance diagrams. An association diagram shows the relationship between two classes and an inheritance diagram shows if a class is derived from another class. Examples of these diagrams are show in Figure B.2

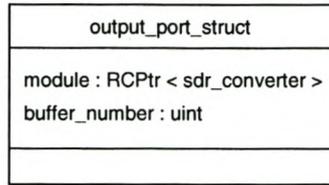


Figure B.3: A Detailed UML Class Diagram Of The `output_port` Structure

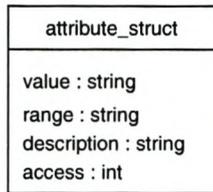


Figure B.4: A Detailed UML Class Diagram Of The `attribute_struct` Structure

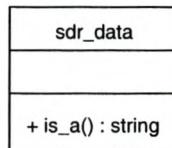


Figure B.5: A Detailed UML Class Diagram Of The `sdr_data` Class

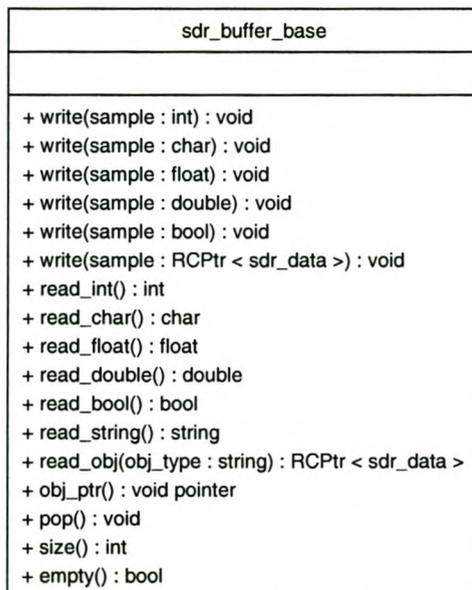


Figure B.6: A Detailed UML Class Diagram Of The `sdr_buffer_base` Class

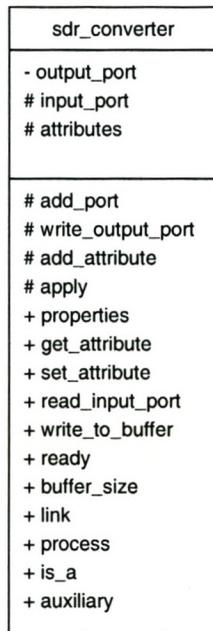


Figure B.7: A Detailed UML Class Diagram Of The `sdr_converter` Class