Cape Peninsula University of Technology
Digital Knowledge

Peninsula Technikon Theses & Dissertations | Theses & Dissertations

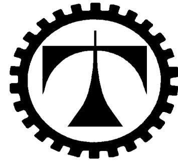
1-1-2003

A hardware emulator testbed for a software-defined radio

Jason Witkowsky
Peninsula Technikon

Recommended Citation
Witkowsky, Jason, "A hardware emulator testbed for a software-defined radio" (2003). *Peninsula Technikon Theses & Dissertations.* Paper 2.
http://dk.cput.ac.za/td_ptech/2

This Text is brought to you for free and open access by the Theses & Dissertations at Digital Knowledge. It has been accepted for inclusion in Peninsula Technikon Theses & Dissertations by an authorized administrator of Digital Knowledge. For more information, please contact barendsc@cput.ac.za.

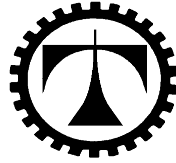


PENINSULA TECHNIKON
FACULTY OF ENGINEERING

A HARDWARE EMULATOR TESTBED FOR A SOFTWARE-DEFINED RADIO

JASON WITKOWSKY

PENINSULA TECHNIKON



Department of Electrical Engineering

Faculty of Engineering

A HARDWARE EMULATOR TESTBED FOR A SOFTWARE-DEFINED RADIO

by

Jason Witkowsky

Submitted in fulfillment of the requirement for the

Masters degree of Technology (MTech): Electrical Engineering

Under the supervision of

Robert Gordon Key (internal)
Gert-Jan van Rooyen (external)

DECEMBER 2003

A HARDWARE EMULATOR TESTBED FOR A SOFTWARE-DEFINED RADIO

Jason Witkowsky

December 2003

DECLARATION

I, Jason Witkowsky, hereby declare that the thesis presented here is my own work and the opinions contained herein are my own and do not necessarily reflect those of the Technikon. All references used have been accurately reported.

Name: Jason Witkowsky

Signature:

Date:

Acknowledgments

I would like to express my appreciation to the following people for their contribution and support during my research:

- My study leader, Gert-Jan van Rooyen, for his continued guidance and infectious enthusiasm throughout the research. His leadership and expertise during the time of my research proved both invaluable and extremely rewarding.
- Mr. Shahien Behardien for his tremendous help and guidance, especially in the former stages of the research project.
- Mr. Tony Abrahams, for his invaluable input at various stages of my research.
- Mr Robert Key, for his continued support and willingness to help.
- The staff and students at the Peninsula Technikon.
- Everybody at Stellenbosch University who assisted me in any way, especially the members of the SDR group.
- Dr. Cornel van Niekerk of Stellenbosch University for his expertise and assistance in the RF lab.
- Mr. Justin Slack for taking the time out to proof-read this thesis.
- My sponsors, the National Research Foundation (NRF), for funding the project.
- Finally, and certainly not least, thank you to the invaluable support of my family and friends who, throughout various setbacks, continued to support and believe in me.

A Hardware Emulator Testbed for a Software-defined Radio

by

Jason Witkowsky

Summary

Contemporary software-defined radio (SDR) is continuously changing and challenging the way traditional RF systems operate. Having more of a radio system's operation in software enables further flexibility through the use of software manipulation. Due to practical limitations, however, it is not always feasible to have the entire radio system's operations performed using software. Practical limitations, therefore, require that a SDR employs some form of RF front-end in order to interface the antenna signals and the signals prior to the data converters.

As technology grows in support of SDR development, this hardware interface is becoming increasingly smaller. The problem with the rapid rate at which SDR developments are occurring is that RF hardware needs to change accordingly. Therefore, the RF hardware front-end can be seen as a non-standardised piece of equipment. To the designer, this means having to prototype in hardware in order to experiment with various types of SDR hardware front-ends.

One of a SDR's main attractions is the inherent property of software testability. Taking this fact into account, this thesis investigates the design and operation of a basic software-driven RF front-end emulator for a SDR. Basic prototype software models are identified and developed in order to test their performance within the emulator. The focus of the thesis, however, is geared toward the development of a software architecture that enables a high degree of interchangeability amongst the underlying modelled components.

In the case of a SDR, the advantage of prototyping in software is in predicting the behaviour of a system prior to having to perform any physical developments. This property of software testability in the emulator can only fully be appreciated if a bench-mark system is used to evaluate the overall performance of the emulator. Therefore, a physical hardware setup is performed in order to test the basic aspects of the emulators operation. This evaluation is not meant as an exhaustive analysis of the emulator, but aims to highlight the overall performance of the emulated system against a typical physical system setup.

Contents

Nomenclature **x**

1 Introduction **1**

1.1 Requirement for a software-defined radio hardware emulator 1

1.2 The scope of the thesis 5

1.3 Thesis outline 5

2 Software-defined radio overview **7**

2.1 Background 7

2.2 Challenges 9

2.3 SDR architectures 9

2.4 Component identification 11

2.5 Implementation 14

2.6 Measures of performance for a SDR 15

2.7 Summary 16

3 Component theory and modelling considerations **17**

3.1 Introduction 17

3.2 Components 17

3.3 Noise 18

3.3.1 Noise modelling considerations 19

3.4 Data converters 20

3.4.1 DAC/ADC classification and architectures 21

3.4.2 Ideal conversion 23

3.4.3 Non-ideal converter errors 25

3.4.4 Data converter modelling considerations 28

3.5 Amplifiers 37

3.5.1 Saturation and clipping 38

3.5.2 1-dB compression point 39

3.5.3 Harmonic distortion 40

3.5.4 Intermodulation distortion 42

3.5.5 Amplifier modelling considerations . . . 44
3.6 Mixers . . . 48
3.6.1 Conversion gain and loss . . . 51
3.6.2 Noise figure . . . 52
3.6.3 Conversion compression point . . . 52
3.6.4 Harmonic intermodulation products . . . 52
3.6.5 Interport isolation . . . 53
3.6.6 Quadrature mixers . . . 54
3.6.7 Mixer modelling considerations . . . 58
3.6.8 Quadrature mixer modelling considerations . . . 65
3.7 Filters . . . 68
3.7.1 Filter classification . . . 68
3.7.2 Filter modelling considerations . . . 69

4 Architecture design and model implementation 74
4.1 General design considerations . . . 74
4.2 Architecture objectives . . . 75
4.3 Architecture design . . . 75
4.3.1 High-level design considerations . . . 75
4.3.2 System topology . . . 78
4.4 Sample rate considerations . . . 82
4.5 Model transfer considerations . . . 82
4.5.1 Data converter models . . . 83
4.5.2 Amplifier model . . . 85
4.5.3 Mixer model . . . 85
4.5.4 Quadrature mixer model . . . 85
4.5.5 Filter model . . . 88
4.6 A front-end application design . . . 89
4.6.1 Add and remove components . . . 89
4.6.2 List components . . . 90
4.6.3 Link components . . . 91
4.6.4 List component links . . . 92
4.6.5 Run emulation . . . 93
4.7 Generating and monitoring samples . . . 94

5 Emulator evaluation — a sample system 96
5.1 Hardware system setup . . . 96
5.1.1 System overview . . . 96
5.1.2 Measurements . . . 98

5.2 Emulated system setup . . . 105
5.2.1 System overview . . . 105
5.2.2 Emulated system outcomes . . . 112
5.3 Conclusions and summary of results . . . 124

6 Conclusion and recommendations 127
6.1 Background and Aim . . . 127
6.2 Achievements of thesis . . . 128
6.2.1 The development of an abstract base class . . . 128
6.2.2 The development of sample modules . . . 128
6.2.3 Software architecture development . . . 128
6.2.4 Verification using a physical setup . . . 129
6.2.5 Post-processing and visualisation tools . . . 129
6.3 Application of the results . . . 129
6.4 Future work . . . 130
6.5 Publications in connection with thesis . . . 131

A Source code 136
A.1 MATLAB simulation models . . . 136
A.1.1 Data converter simulation models . . . 136
A.1.2 Amplifier simulation models . . . 136
A.1.3 Mixer simulation models . . . 137
A.1.4 Filter simulation models . . . 137
A.1.5 Post-processing routines . . . 137
A.2 C++ code listing . . . 161
A.2.1 Architecture source code . . . 161

List of Figures

1.1 Illustration of a connection between the SDR and hardware emulator architectures . . . 3

2.1 An ideal software-defined radio receiver. The equivalent SDR transmitter would employ a similar configuration, with a reverse signal path flow. . . . 10
2.2 A typical SDR block diagram . . . 11
2.3 Illustration of three typical SDR receiver hardware front-end configurations 12
2.4 Functional block diagram illustration of the envisaged SDR hardware emulator . . . 13
2.5 A flexibility and performance summary of ASICs, FPGAs and DSPs 14

3.1 Illustration of component modelling . . . 18
3.2 Block diagram showing how noise is modelled in the system . . . 19
3.3 Illustration of the addition of white gaussian noise to a signal . . . 20
3.4 Illustration of a 1-bit R-2R DAC . . . 22
3.5 Ideal DAC transfer function . . . 24
3.6 Ideal ADC transfer function . . . 25
3.7 An example of common DAC INL curves . . . 27
3.8 Illustration of spurious free dynamic range . . . 28
3.9 Illustration of a typical DAC glitch on an output transition . . . 29
3.10 3-bit quantisation model output . . . 30
3.11 Transfer function of an ideal and pre-distorted conversion . . . 31
3.12 Negative glitch impulse illustration . . . 32
3.13 DAC output response showing typical glitch characteristics . . . 33
3.14 Illustration of the different types of glitch area measurements . . . 34
3.15 The deterministic property of glitch impulse . . . 35
3.16 Harmonics due to upsampling . . . 36
3.17 Effect of filtering glitch transients . . . 37
3.18 The requirement for upsampling . . . 38
3.19 The cascade of models to form a more complex model . . . 39
3.20 Charateristics of an amplifier output . . . 39

3.21 Saturated sinusoidal waveform in the time and frequency domain 40
3.22 Frequency response of an amplifier with a second-order nonlinearity to a single-tone input 41
3.23 Frequency plot of signal passed through an amplifier with a second-order nonlinearity 41
3.24 Frequency plot of signal passed through an amplifier with a third-order nonlinearity 43
3.25 Quadrature Taylor series amplifier model 45
3.26 Class-A amplifier I and Q channel response 46
3.27 Flowchart of saturation algorithm 49
3.28 A double balanced mixer circuit 50
3.29 Noise Figure versus LO power for a generic diode double-balanced mixer . 53
3.30 Illustration of a quadrature mixer block diagram 55
3.31 Illustration of a typical intermodulation table 59
3.32 A mixer modelled using two Taylor amplifiers 60
3.33 Cascaded mixer model to improve the accuracy of IM component representation 62
3.34 Output of the mixer model using a single Taylor amplifier pair 63
3.35 The IMT used to verify the operation of the mixer model 64
3.36 Spectral output of the mixer model using the sum of two Taylor amplifier pairs 64
3.37 Ideal quadrature mixer output 66
3.38 Spurious component measurement on the output of a quadrature mixer . . 66
3.39 Spectral output of quadrature mixer model 67
3.40 Output of quadrature mixer showing artifacts due to phase imbalance . . . 68
3.41 Direct form I filter implementation 71
3.42 Example of a second-order polynomial structure 72
3.43 Impulse response of a quadratic filter prototype model 72

4.1 Illustration of the commonality between SDR hardware components 76
4.2 Strategy-based shallow hierarchy illustration 77
4.3 A generic component illustrating the input and output interfaces 78
4.4 Illustration of the emulator's control-based interface which is used to monitor the status of the buffers 79
4.5 Example of how links are setup in the emulator architecture 81
4.6 Outputs for selected discrete-time frequency values 87
4.7 An illustration of how modelled components are polled in the emulator architecture 94

5.1 Block diagram of the hardware system setup . . . 97
5.2 Quadrature-mixed soundcard output with no input data . . . 98
5.3 Quadrature-mixed soundcard output with I and Q data inputs applied . . 99
5.4 Amplitude compensated output showing a decrease in the adjacent side-band component . . . 101
5.5 Determination of soundcard's filter transfer function . . . 102
5.6 Output with 1% carrier leakthrough added . . . 103
5.7 Output with 1% amplitude imbalance . . . 103
5.8 Output with 3° of phase imbalance added . . . 104
5.9 Combination of the three quadrature mixer impairments . . . 105
5.10 A screen shot of the user front-end application showing the various options. 106
5.11 The setup of the emulator evaluation system . . . 109
5.12 Spectrum of the 10 kHz input cosine waveform . . . 114
5.13 Scaled output after quantisation . . . 115
5.14 Frequency plot for the M3 monitor file . . . 116
5.15 Effects of the sound cards LPF . . . 117
5.16 Output of the quadrature mixer model . . . 118
5.17 Final spectral output of the emulated system . . . 119
5.18 Output of the emulator's quadrature mixer with 1% carrier leakthrough . . 121
5.19 Output of the quadrature mixer model showing the effects of amplitude imbalance in the frequency domain . . . 122
5.20 Output of the quadrature mixer model, showing the effects of 3° of phase imbalance added . . . 123
5.21 Frequency plot showing the combined quadrature effects in the quadrature mixer model . . . 125

List of Tables

3.1 Calculation of the third-order IM terms 44
3.2 Typical performance figures for a passive mixer 53

5.1 Suggested window choices based on signal frequency content 113
5.2 Comparison of hardware and emulation measurements 121
5.3 Comparison of hardware and emulation measurements with 1% amplitude imbalance added 123
5.4 Comparison of hardware and emulation measurements with 3° phase imbalance added 124
5.5 Comparison of hardware and emulation measurements with the combined effect of all three quadrature impairments 125

Nomenclature

Acronyms

ADA	Analogue-to-Digital and Digital-to-Analogue Converter
ADC	Analogue-to-Digital Converter
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
COTS	Commercial-off-the-Shelf
DAC	Digital-to-Analogue Converter
DC	Direct Current
DNL	Differential Nonlinearity
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
IF	Intermediate Frequency
IM	Intermodulation
IMT	Intermodulation Table
INL	Integral Nonlinearity
LO	Local Oscillator
LPF	Low-Pass Filter
NF	Noise Figure
RF	Radio Frequency
SDR	Software-Defined Radio
SDRF	Software-Defined Radio Forum
SFDR	Spurious-Free Dynamic Range
SQNR	Signal Quantisation Noise Ratio
SVD	Singular Value Decomposition
USR	Ultimate Software Radio

Variables

f	Continuous frequency
f_c	Discrete frequency
f_{IMP}	Intermodulation product for two-tone signals
f_{out}	IM components for single or multi-tone signals
$h_0...h_3$	Harmonics related to Taylor amplifier
$H(s)$	Analogue filter transfer function
$H(z)$	Digital filter transfer function
$I_{(\mathrm{t})}$	In-phase quadrature mixer input signal
k	Boltzmann's constant
K	Taylor amplifier gain coefficient
L_{min}	Conversion loss in double-balanced mixer
m	Gain variable used in emulated amplifier model
N_p	Value of calculated noise power for emulator AWGN model
$Q_{(\mathrm{t})}$	Quadrature-phase quadrature mixer input signal
SFDR_α	Spurious free dynamic range associated with phase error
SFDR_β	Spurious free dynamic range associated with amplitude error
SFDR_ρ	Spurious free dynamic range associated with offset error
T	Temperature in degrees Kelvin
v_I	Taylor amplifier I-channel voltage
$v_{\mathrm{i(t)}}$	Amplifier input voltage
v_N	Open-circuit noise voltage [rms]
$v_{\mathrm{out}(t)}$	Output of ideal memoryless linear amplifier
v_Q	Taylor amplifier Q-channel voltage
w_n	Output of AWGN model
y_n	Output of the addition of AWGN

Chapter 1

Introduction

1.1 Requirement for a software-defined radio hardware emulator

Radio communications technology has been redefined by digital signal processing.

Tremendous developments within the wireless industry as well as the availability of high-performance, inexpensive, low-power signal processing and reconfigurable logic devices have led to explosive advances in software-defined radio (SDR). In general, a SDR is a radio communications platform on which the majority of the signal processing is performed in the digital domain.

Performing the duties of traditionally analogue hardware in software alleviates maintenance and calibration duties. Another plus is the consistency of performance in reproducing digital circuitry as apposed to its analogue counterpart. One of the primary advantages of performing the system functionality in the software domain is the vast flexibility associated with software processing. The benefit of software upgrades to a communications system can now prevent communications systems from becoming outdated by allowing the system to become easily adaptable to changes. Such changes could include, in the case of cellular base stations, the introduction of a new wireless standard. This is a possibility since there are a vast amount of cellular wireless air-interface standards both present and emerging.

The vision to try and unify these standards on a single communications device with minimal complexity and cost has caused widespread interest in software-defined radio technology to grow rapidly, especially in terms of industry awareness. The benefits of software radio in the context of cellular base stations have also caused it to be a focal point of a number of technical papers (Arnott et al. 1998, Salkintzis et al. 1999, Turletti & Tennenhouse 1999, Zangi & Koilpillai 1999). These benefits range from channel reduction in cellular base stations (Turletti & Tennenhouse 1999), migration to software for system-specific parameters (rendering more flexibility in the system) as well as "future-proofing"

the design for emerging standards. With a cellular handset using software radio technology to enable it to conform to multiple standards across the globe, true seamless global roaming could become a reality. Cellular base stations should be able to avoid costly hardware upgrades as more of the functionality is executed in software. This would also enable over-the-air, on-the-fly reconfiguration of cellular communications infrastructure. In fact, documents have already been developed detailing specific protocol requirements for downloading software to a SDR. These documents are available via the SDR Forum website (SDRF 2002). Continuous SDR developments can be tracked on the website since the Software Defined Radio Forum is an organization aimed at promoting SDR technology on a global scale.

SDR overcomes the drawbacks of analogue signal processing — but there are limitations. This limitation lies predominantly in the bridge between the analogue and digital realm: the analogue-to-digital and digital-to-analogue (ADA) converter. Research has shown that the ADA is the key enabling component of a SDR (Walden 1999*b*, Salkintzis et al. 1999). What this ADA limitation means is that some form of RF front-end will inevitably be present in order to yield practical SDR designs.

A SDR hardware platform in practice is a non-standardised piece of equipment because there is no set point at which the digitisation stage should occur. Added to this is the continuous need to move the software side as close to the antenna as possible to ensure maximum flexibility. Literature has shown and confirmed that a range of hardware configurations exist for a SDR setup and this results in the existence of a number of SDR hardware architectures, each with its own merits (Gunn et al. 1999).

Another key advantage of SDR design is that of software testability. SDR accelerates the development of new communications systems because much of a system's functionality and performance is verifiable in software. Simple software modifications ensure easy experimentation in the software side of a SDR design. To make similar hardware changes, however, is not always this easy and could mean physically re-altering the design.

Studies have shown that complete SDR testbeds have been developed specifically to meet the requirement of end-to-end testing of complete SDR systems — at the cost of employing COTS (commercial-off-the-shelf) components (Reichhart et al. 1999).

The hardware emulator proposed in this thesis forms part of a greater project that aims to develop a library of components for a SDR. Computer workstations are used for initial project development for both SDR and hardware emulation development. The cost-effective approach of using workstations has been shown to be a very effective means for SDR design and experimentation (Bose et al. 1999). The library of components, for both the SDR and the hardware emulator, is developed within a C++ object-oriented environment — this ensures that integrating the two is simply a matter of software linkage. The envisaged concept of the hardware emulator and its relationship with the SDR

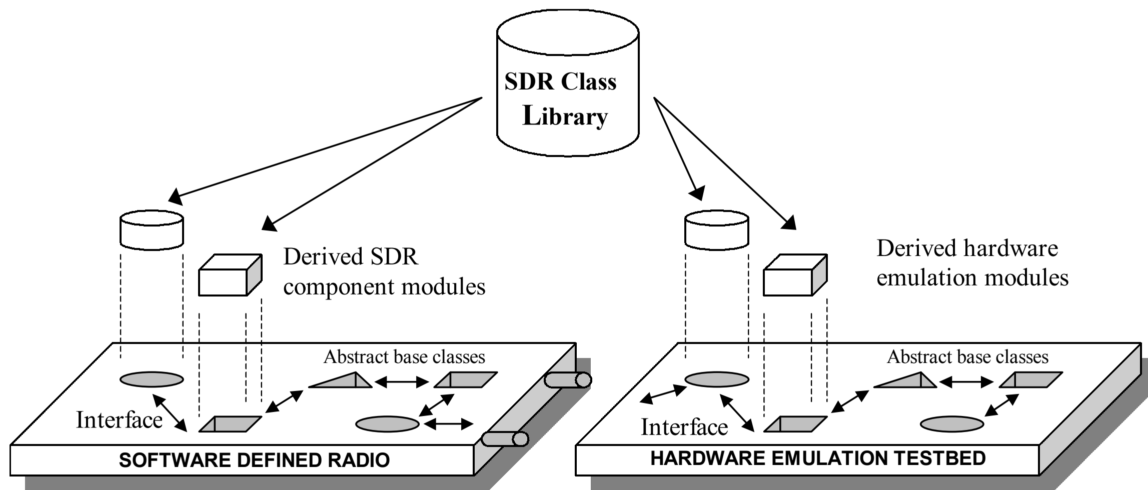


Figure 1.1: *Illustration of a connection between the SDR and hardware emulator architectures*

software architecture is shown in figure 1.1 (Witkowsky & Van Rooyen 2002).

Of course, there are advanced computer simulation tools that are available in order to address the issue of testing and verifying the performance of a system prior to any actual physical development. These simulations rely on algorithms that specify the behavior of a particular system. Complex system simulations of complete end-to-end designs are often used to test systems in their entirety (Rhode & Whitaker 2001). This requires the designer to develop a simulation model for the entire system. The advantage of a SDR in this regard is that the actual processing is performed in software already — it is only the hardware front-end that effectively requires modelling. In this regard, a simulation can be utilised to analyse and make predictions about a particular RF front-end. One of the thesis objectives, however, is to have the capability to integrate the SDR software architecture and modelled RF front-end on the same platform — for this purpose, a hardware emulator fits the criteria.

The overall objective then of the dissertation is to extend the software-testability of a SDR to the remaining hardware components (the hardware front-end) of a SDR. This intention ultimately brought about the requirement for a hardware emulator to facilitate the overall SDR development process. Emulation is essentially an attempt to mimic the functionality of a device or system and the benefits of a SDR hardware emulator can be summarised as follows:

- The complete end-to-end testing of a SDR prior to any hardware construction.
- The testing of a variety of hardware front-end configurations in software.
- The capability to investigate the effectiveness of software compensation algorithms

used to address the effects of hardware component imperfections.

- Emulating a hardware front-end gives the designer the ability to deliberately introduce hardware component inaccuracies. Intentionally adding inaccuracies could be used to test the system's overall performance for worst case scenarios.

- Data analysis and visualisation is made easy in software. Data streams, therefore, at various points in the emulated front-end can easily be made accessible for investigation purposes.

It is important to note that the models designed in the emulator need not be exact replicas of the internal system or component design, but could include different ways of implementing the functionality. Stated otherwise, the models at a higher level could just be functional blocks that represent a particular component. The internal functionality, however, can be implemented using various techniques in order to mimic the operations of the device.

In general, an emulator would be used as a substitute for a system (in this case, a hardware front-end) such that the software-processing segment of the SDR must be indifferent to whether it is connected to actual hardware, or to a hardware emulator. In particular, it must be possible to model specific analogue hardware components, and emulate their performance in software, before actual hardware development. The hardware emulation modules should integrate seamlessly with the SDR component modules (but should also be able to run stand-alone), and allow the system designers to fully test their application without the requirement for any actual analogue hardware development. By referring again to Figure 1.1, the user should, at a higher level, have the ability to access both a library of SDR components as well as a library of hardware-emulated components in order to perform a complete system analysis. It must also be possible to provide raw waveform data to applications such as MATLAB, in order to evaluate system performance.

In summary, the objectives of the study are to develop:

1. A collection of abstract base classes that define generic hardware components and their interfaces.

2. A collection of sample modules descended from these abstract base classes, that emulate the performance of actual hardware components.

3. A software architecture in which these models can effectively operate and communicate. The design of the architecture must also facilitate a high degree of interchangeability for the components.

4. Physical circuits to assess the performance of the emulator.

5. A set of tools that provide post-processing and data visualisation capabilities for the emulated waveforms.

1.2 The scope of the thesis

The following deliverables were identified in order to address the research problem:

1. An in-depth literature study on software-defined radio, its possible applications and typical architectures.
2. Identification and thorough research on hardware components that are typically used in the front-end of SDR systems. The research must focus on these components' expected non-idealities, and how to model them.
3. A study on the modelling and emulation of hardware components when used within the emulator architecture, including the development of sample models for system testing.
4. The design and development of the hardware emulators software architecture.
5. The design and study of a physical hardware setup against which the emulator results will be compared.
6. The capturing of emulation results in order to perform comparisons for a specified hardware front-end architecture.
7. Conclusions that compare the emulation results to those of the physical hardware setup, and provide a thorough and scientific assessment of the performance of the SDR hardware emulator testbed.

1.3 Thesis outline

A thorough literature research is the primary objective of Chapter 2 which starts by looking over the history of software-defined radio (SDR) and highlights the benefits of the technology. It is in presenting the challenges within SDR that Chapter 2 illustrates the problem of non-standardised RF front-ends. This finding states the motivation for the development of the hardware emulator testbed. Typical architectures are also examined in order to gain some insight into how a general SDR is composed, both at the block diagram and component level.

The architectures looked at in Chapter 2 highlight an important section of the study — the hardware components required for modelling. Chapter 3 aims to provide the theoretical background on the identified components in order to gain a thorough understanding

of the non-idealities presented by each component. The knowledge of the components' behaviour is then used to develop appropriate prototype models. These models are developed in MATLAB in order to assess the performance of the technique or algorithm used to emulated the non-ideal components' behaviour.

The effectiveness of the models can only truly be exploited if the architecture used to house the components is well designed. Chapter 4 illustrates the design considerations taken in order to implement an architecture that optimally accommodates the modelled components. Various considerations such as buffer control and the linking of components are dealt with. The ultimate objective of the architecture is to allow a high degree of interchangeability amongst components in order to emulate a variety of hardware front-ends. Chapter 4 takes a detailed look into how this objective is achieved.

For an emulated system to be considered effective and useful, some form of benchmark is needed to comparatively assess the hardware emulator's performance. This benchmark comes in the form of a practical hardware setup, and Chapter 5 takes an in-depth look at the results presented by both the practical and emulated system. These results are then scrutinised in order to provide a thorough scientific report on the outcomes. The outcomes are then used to gauge the overall performance of the emulator.

Lastly, Chapter 6 is used to present the final product of the thesis — to give conclusions based on the outcomes of the research results. Suggestions for future work, in order to improve on further developing the hardware front-end emulator, are also included.

Chapter 2

Software-defined radio overview

The hardware emulator presented in this thesis is designed specifically for a SDR and it is for this reason that an in-depth literate study on SDR will be given. The study will begin by looking briefly at the history of SDR and by providing a deeper understanding of what a SDR actually is. In practical software-defined radios, some form of RF front-end hardware is often used between the antenna and the software processing segment. The reason for this hardware front-end is mainly due to the requirement for extensive analogue signal processing. These and other limitations are further presented in Chapter 3.

Chapter 3 also focuses on typical SDR architectures in order to obtain a better understanding of the various hardware front-end setups that are employed. The SDR architectures should also reveal what components are used in the various types of hardware front-ends. This finding is important since it is these very components that will be modelled in the emulator.

2.1 Background

The intermediate phase between where SDR technology began and where it stands today has showcased a number of developments. The name Joseph Mitola is synonymous with software radio, a term he coined in 1991 (Mitola III 2000). The term software radio was used by Mitola (who in fact wrote the first paper on software radio in 1992) to describe a type of radio that was reconfigurable in software (Mitola III 2000).

Past developments of SDR emerged when the requirement for a flexible (multi-band, multi-mode) radio system for military purposes was pursued. In 1992, this resulted in the development of a program entitled SPEAKeasy (Lackey & Upmal 1995), a military-based program to demonstrate and fully utilise the concept of SDR. The result of the study was the SPEAKeasy software reprogrammable radio that allows software changes of radio functionality and operates in the 2 MHz to 2 GHz frequency band. In order to address such a wide frequency range, the radio set used different interchangeable RF front-ends

for different frequency bands.

The definition of a SDR is not a straightforward one. Much controversy surrounds the level of reconfigurability needed for a radio set to be labeled as a SDR. In fact, the Software Defined Radio Form (SDRF) uses a set of tiers to describe a class of radio with regard to its facilities in order to demystify the SDR semantics (SDRF 2002). The range begins at tier 0 which is a hardware radio in which physical changes are required to alter the characteristics of the radio. The range extends up to tier 4 which describes the ultimate software radio (USR), a fictitious model used merely for comparative reasons, as a radio that has no operating frequency limits, no external antenna, just a single connector to interface to the outside world, and also a radio that can rapidly switch between different interface formats. A software-defined radio (tier 2) describes a class of radio in which some form of front-end hardware exists between the antenna and the software processing. The use of software processing in a SDR includes, but is not limited to, modulation scheme alterations, channel selectivity and communications security functions (SDRF 2002). The SDRF formally defines a SDR as (SDRF 2002):

> A collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals.

SDR is thus an implementation technique that facilitates the software embodiment of traditional analogue circuit functionality. In a certain sense, SDR could be thought of as an emulation of the traditional analogue component functionality, since the digital portion of a SDR is continuously being pushed closer to the antenna.

The direct benefits of moving the digitisation stage as close to the antenna as possible and having the remaining processing tasks software reconfigurable can be summarised as follows (Mitola III 1995, Mitola III 1999*b*, Buracchini 2000, SDRF 2002):

- A flexible architecture is created that has the capacity to support current and evolving commercial air-interface standards.
- Over-the-air software downloads can be performed to enable system upgrades.
- Advanced error-correction schemes can be employed in software to ensure data integrity.
- Minimal RF problems are encountered since the analogue section gets smaller and performs fewer or no waveform-specific tasks
- Cost benefits are gained due to minimal RF hardware expenditure.
- Full advantage can be taken of rapidly improving signal processing and data converter technology.

2.2 Challenges

In order to take full advantage of the benefits mentioned above, certain restrictions need to be addressed. One of the primary aims of a SDR is attempting to move the digitisation stage as close to the antenna as possible and replacing dedicated hardware with flexible signal processing devices (such as DSPs) (Buracchini 2000). As stated earlier, the primary advantage associated with the migration of system functionality into the software realm is the immense flexibility obtainable with SDR technology. This idea of a more software-driven system opens up a host of possibilities previously too complex to obtain within traditional hardware-constrained radio systems.

In practice there are several technical challenges facing the designer in order to exploit the immense flexibility that SDR has to offer (Buracchini 2000). These challenges arise as a result of the drive toward the capabilities that the ideal SDR (Fig 2.1) has to offer — a system that digitises directly at the antenna, with all subsequent processing performed in software (Buracchini 2000). Research has shown that ADA converter technology development is one of the key enabling elements to fully exploit the benefits that SDR has to offer (Walden 1999*b*, Salkintzis et al. 1999). Part of this challenge is a trade-off decision between converter sampling rates and resolution, since in general, an increase in one results in a decrease of the other (Walden 1999*a*). Some of the other more general technical challenges facing designers of a SDR are (Mitola III 1999*b*, Mitola III 1995, Buracchini 2000, Mitola III 2000):

- Addressing changing data rates across interprocessor interfaces.
- Estimating processing requirements of reprogrammable devices.
- Selecting a suitable flexible implementation platform (software architecture).
- Minimising power consumption (especially in portable devices).
- Selecting suitable RF and IF processing platforms.

2.3 SDR architectures

SDR architecture analysis is a well-documented area of research and contains various perspectives ranging from COTS implementations (Reichhart et al. 1999) to mathematical interpretations (Mitola III 1999*a*). An insight into a basic SDR architecture at the block diagram level will be given and, for the purpose of this thesis, only three typical RF front-end architectural configurations for a single channel SDR design will be investigated.

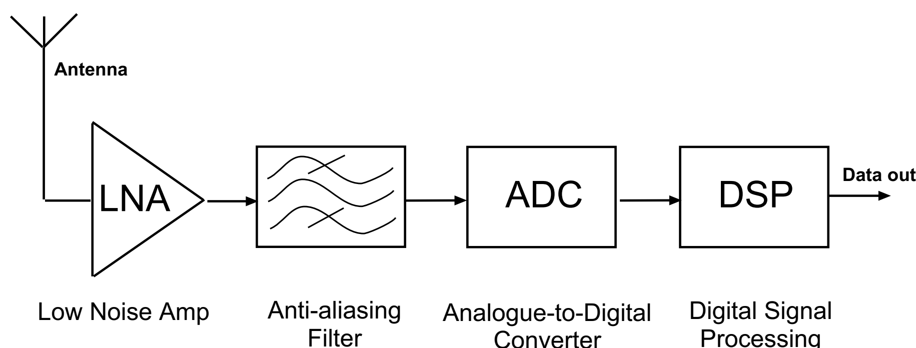


Figure 2.1: *An ideal software-defined radio receiver. The equivalent SDR transmitter would employ a similar configuration, with a reverse signal path flow.*

Figure 2.2 shows a typical block diagram of a SDR architecture and represents both a transmitter and a receiver (Reed 2002).

For the SDR receiver stages of Figure 2.2, the RF signal is received via the antenna[1] and in order to keep the bandwidth to the ADC practical, a hardware front-end (typically in the form of a superheterodyne receiver) is employed. The hardware front-end usually performs some form of filtering, amplification and mixing to frequency-translate the RF to an IF signal. Ultimately, the application in which the radio is used determines the number of these stages required (variations of these types of stages will be illustrated later). The next stage is the ADC which digitises at the IF frequency — this sampled IF creates spectral replicas which can be placed near the baseband and thus combining the processes of frequency translation and digitisation (Reed 2002). This combined process of frequency translation and digitisation is performed using a technique called undersampling and the reader is referred to (Kester 1997) for a tutorial on undersampling. In order to properly interface the output of the ADC to the digital processing hardware, channelization (channel selection using filtering) and sample rate conversion is performed. The subsequent processing can be performed on a variety of platforms and section 2.5 looks at the trade-offs of three of the most commonly used hardware processing platforms.

Recall that Figure 2.2 shows both a receiver and a transmitter. Up to now only the functions of the receiver are explained. A SDR transmitter, according to figure 2.2, begins it operations at the software processing platform. As with the receiver, the SDR transmitter's signal could also require sampling rate conversion. A DAC is then used to generate the analogue equivalent of the digital input signal. Finally, as with the receiver, RF hardware stages are used to up-mix the output of the DAC to the desired RF frequency

[1]Figure 2.2 shows this as a smart antenna, which if used, minimises noise, interference and multipath signal problems because of the antenna's high directivity characteristics (Reed 2002)

before the signal is transmitted by the antenna.

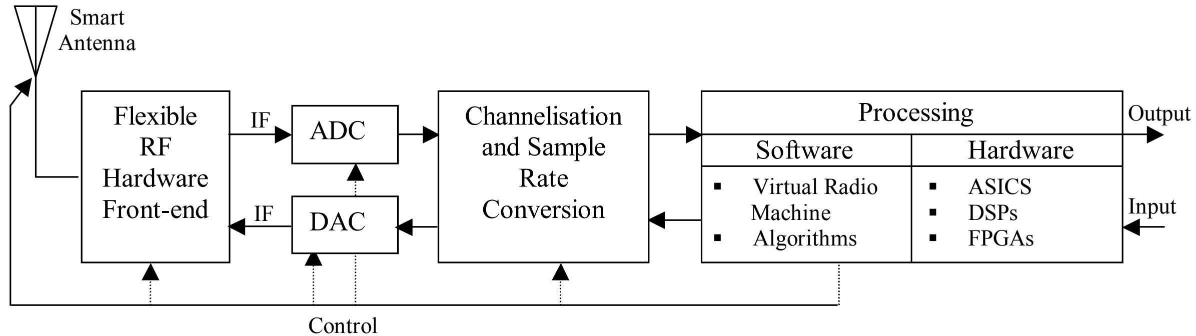


Figure 2.2: *A typical SDR block diagram (adapted from (Reed 2002))*

The analogue in-phase and quadrature (I and Q respectively) superheterodyne receiver, a common RF receiver configuration for a software radio, is presented in figure 2.3 (i) (Gunn et al. 1999). Variations to this configuration include passband alternatives in which the quadrature demodulation is performed digitally, using a single ADC for IF sampling, as shown in Figure 2.3 (ii). A second variation (Figure 2.3 (iii)), in which the RF signal is directly converted into baseband using IQ downconversion, affords additional flexibility, but is more challenging to practically implement than the previous two configurations.

2.4 Component identification

Figure 2.3 highlights some of the possible architectures that could be encountered in SDR hardware front-ends. This by no means includes all possible architectures, but refers to some configurations that are commonly employed in single-channel SDRs. By observation of the various hardware architectures in figure 2.3, the following key components of the RF front-end will be modelled:

- Data converters (both ADCs and DACs)
- Amplifiers
- Mixers (both standard and quadrature)
- Filters
- Antennas and channels

It is important to note that in the software processing block in Figure 2.3, the partitioning happens according to the functions assigned to a specific software processing task. The

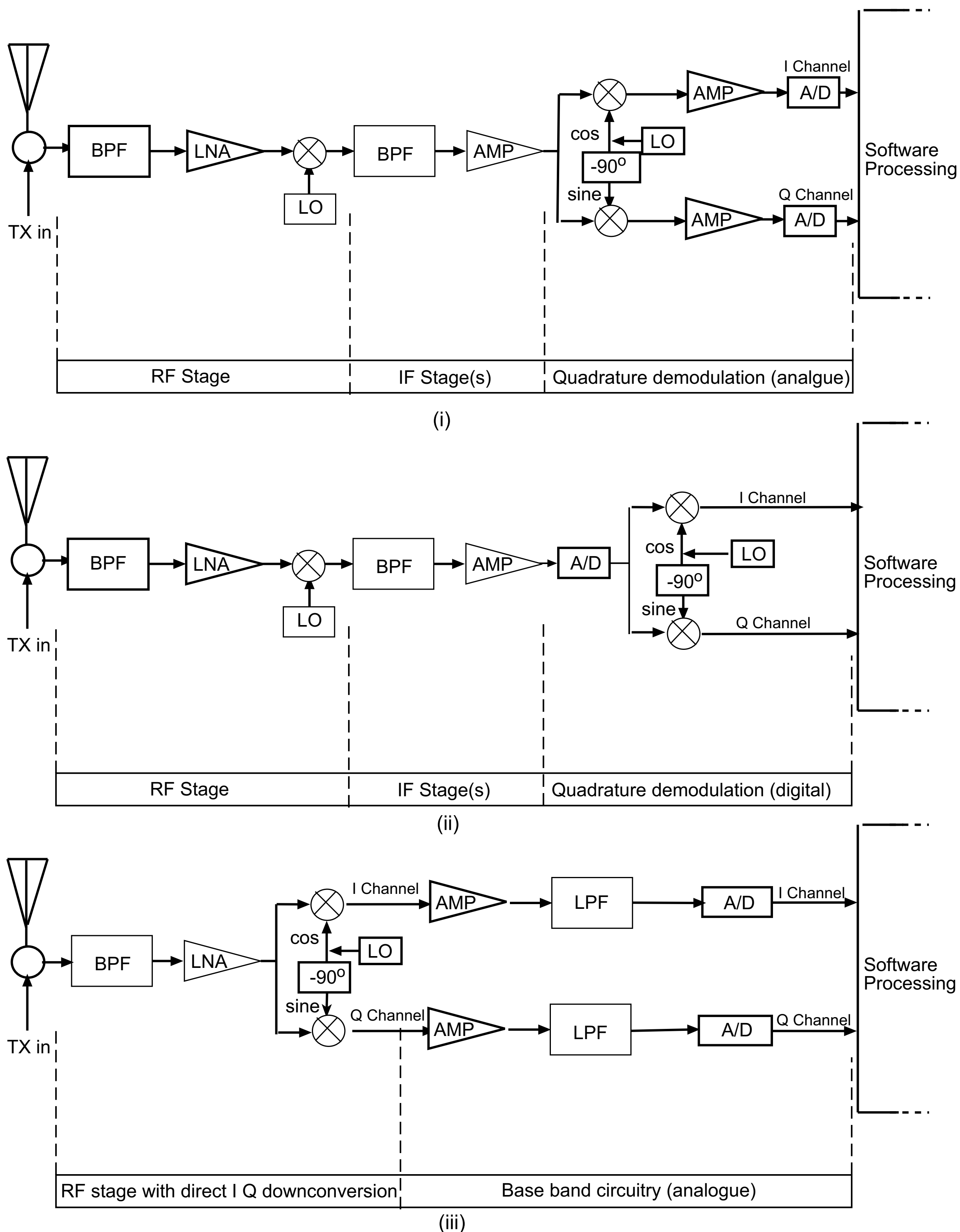


Figure 2.3: *Illustration of three typical SDR hardware receiver front-end configurations. A typical SDR hardware front-end transmitter (at the block diagram level) would employ similar components as the receiver. (adapted from (Gunn et al. 1999))*

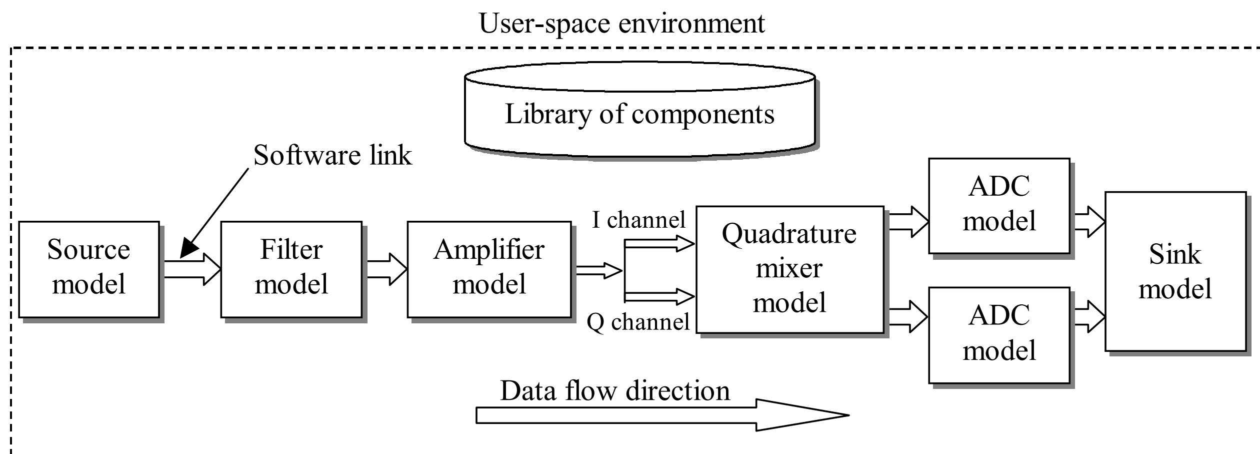


Figure 2.4: *Functional block diagram illustration of the envisaged SDR hardware emulator*

type of software processing function used ultimately depends on the application for which the radio is required and varies between designs.

Figure 2.4 shows a functional block diagram (a receiver in this case) in order to get a conceptual idea of the envisaged outcome of this thesis at this stage. Ultimately, the user should have a library of modelled hardware components at their disposal. These components could be placed in any sequence in order to model various types of hardware front-ends, such as those illustrated in Figure 2.3. Conventional hardware systems require physical interfaces to link components, but in the emulator the component interfaces are implemented as software links. Chapter 4 focuses on the design of these software links. The source and sink models represent points where the data samples are accumulated and distributed respectively. For example, the source model could be a file with samples that represent data received at the antenna. In a complete end-to-end SDR system, the sink model of Figure 2.4 represents the software architecture of a SDR. Not having a SDR software architecture present, however, does not limit the use of the emulator and if this is the case, the sink module could just represent a software object that receives and stores incoming samples. The samples could, for example, be used later for analysis in an application program such as MATLAB.

The actual effects presented by the components in the diagrams of figure 2.3 will be looked at in detail in Chapter 3 in order to effectively model their component behavior for the emulated design.

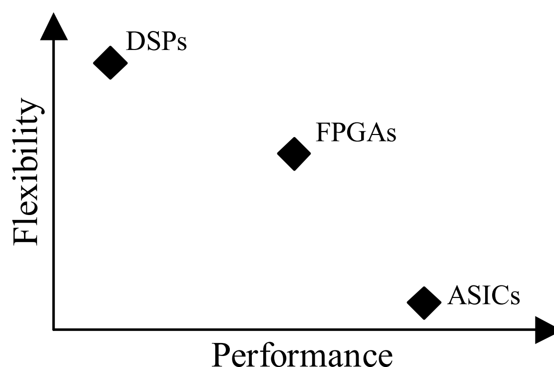


Figure 2.5: *A flexibility and performance summary of ASICs, FPGAs and DSPs (adapted from (Bose et al. 1999))*

2.5 Implementation

The digital portion of a SDR can be implemented on a variety of signal processing platforms with each platform having its own merits.

Some of the these major platforms include digital signal processors (DSPs), application-specific integrated circuits (ASICs) as well as field programmable gate arrays (FPGAs). ASICs and FPGAs are normally assigned tasks that rarely require frequent adjustment in the field (Pucker, L. 2001). To effectively utilise DSP, ASIC and FPGA technology in a software radio design, the performance vs. flexibility trade-offs for each technology must be understood, and their relationships are shown in figure 2.5 (Ahlquist 1999). Figure 2.5 indicates that if processor speed is imperative, then an ASIC should be used at the cost of reduced flexibility. If, on the other hand, flexibility is a much greater concern than processor speed, then a DSP would more suited to the task. An FPGA lies somewhere in the middle and would generally be used if an balanced speed-flexibility trade-off is needed.

Aside from the performance-flexibility trade-off, other key selection criteria considerations include addressing

- the level of integration required in order to accommodate several functions on a single chip
- the amount of development time needed to implement and test a software radio for a specific signal processing device
- power consumption requirements

The points listed above indicate that selecting a signal processing platform from the devices mentioned can be quite time consuming. An alternative approach to using dedicated devices, such as DSPs, for signal processing requirements, is to make use of a

general-purpose workstation to perform digital signal processing — such a device is called a virtual radio[2] (Bose et al. 1999).

A virtual radio is defined as a communications platform that performs all the signal processing in user space on a workstation (Bose et al. 1999). The primary advantage of this approach is the capability to easily experiment at a higher level with new processing methods and algorithms. Some of the other advantages of employing the workstation approach include:

- Fast deployment times (upgrades to software can be done with great efficiency)
- Facilitates integration with other applications (e.g. a software radio and hardware emulator on the same platform)
- Cost benefits (using a workstation that is already available)
- Use of preferred high-level language and therefore not restricted to (in certain cases) device specific programming languages.

One of the biggest challenges in implementing such a virtual radio is addressing the various problems of interfacing to the outside world (Bose et al. 1999). For the purpose of this thesis, this interfacing dilemma is not an issue, since no connection is needed to external hardware outside the workstation.

The virtual workstation approach, therefore, will be used in this thesis to facilitate rapid, low-cost system development.

2.6 Measures of performance for a SDR

Although a SDR is architecturally different from its analogue counterpart, the standard measures of performance used to evaluate a typical analogue radio still apply. A SDR on the outside, after all, should function like a normal radio — it is what happens on the inside of a SDR that gets done in a different manner. The context in which the SDR is used will determine which performance measurements are most relevant. For example, in the case of audio, amplitude variations with frequency as well nonlinear distortion effects are important, but when considering processing a digital data stream, the number of symbols per second, fraction of errors and the specific waveform symbols are more relevant (Rhode & Whitaker 2001). For the hardware emulator testbed, data streams at various stages in the chain of emulation modules can easily be obtained. This allows essential examination

[2]Considering that this is the approach taken in this thesis (as well as in the software radio group in general), the reader is referred to the article by Pucker (Pucker, L. 2001) if a further discussion of the merits of DSPs, FPGAs and ASICs with regard to SDR design is required.

of the various waveforms by using appropriate workstation application software, such as MATLAB, for data analysis and visualisation.

2.7 Summary

SDR technology poses many design challenges, but offers significant benefits in return — the most important of which is unparalleled system flexibility. Because analogue-digital-analogue (ADA) conversion technology is, for all practical reasons, still limited, RF front-end hardware is often required in a SDR. Even for the ideal SDR, an ADA is a piece of hardware that will always be present. Therefore, in order to facilitate the design and testing of different front-ends, an effective hardware emulation system was proposed. This Chapter was used to illustrate, as well as to gain a complete understanding of, the broader context in which the emulator will be used. An important outcome of Chapter 2 was the study of certain hardware front-ends in order to identify the typical front-end components that could be encountered.

Already at this stage, however, we are able to identify the typical components that should be generically modelled just by observation of figure 2.3. For both transmitters and receivers, these components are identified to be analogue-to-digital and digital-to-analogue converters, filters, amplifiers and mixers.

This outcome is important since it is these very components that need to be modelled for the emulated system. The study of how these components are modelled based on their individual behavioral imperfections, is taken up for further investigation in the following Chapter.

Chapter 3

Component theory and modelling considerations

3.1 Introduction

In order to develop an emulated system for the components in the configurations identified in Chapter 2, it is important to have an understanding of the way in which the hardware components function. The aim of chapter 3 is to present the background theory for each component of the system identified in chapter 2. The theory will include studying and identifying the non-idealities of each identified component. Prototype models are developed in MATLAB in order to fully test and verify the algorithms used to model the components. This is the preliminary process prior to implementing the final models in C++. The prototype models developed for the hardware components can be either complete models or building blocks to form a complete model as shown in figure 3.1. The outcomes of the models' operation are presented in this chapter. In summary, the objectives are:

- Reiterating the identified hardware front-end components required for modelling
- A theoretical study on the components and their non-ideal behaviour
- The modelling consideration taken for each component
- The development of a model prototype for each component

3.2 Components

Recapping from chapter 2, the components identified for modelling were:

- Data converters (both ADCs and DACs)

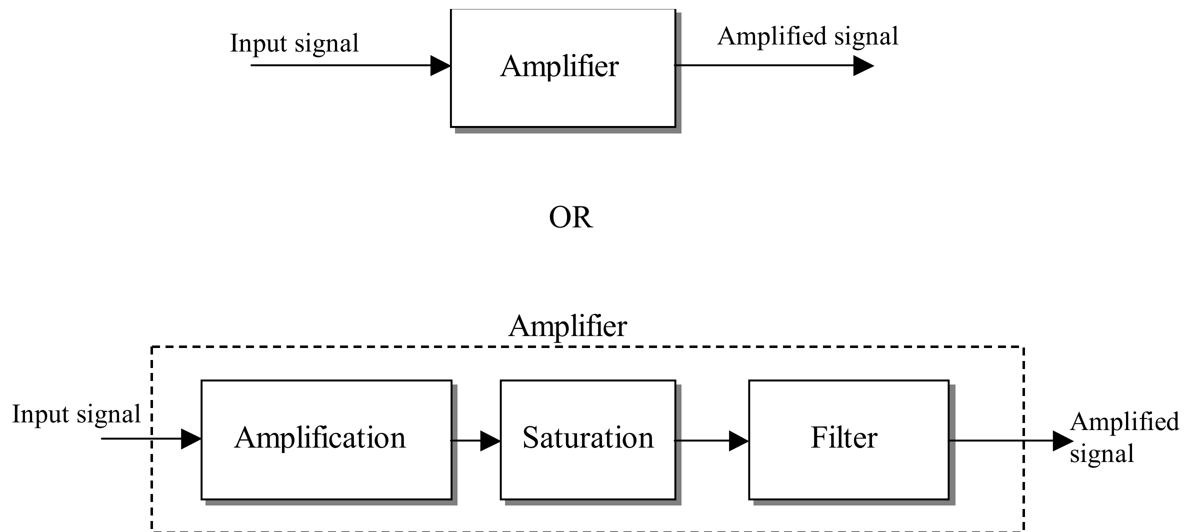


Figure 3.1: *Components can be modelled as complete units or modelled as sub-unit cascades, as this amplifier example illustrates*

- Amplifiers
- Mixers (both standard and quadrature)
- Filters
- Antennas and channels

These components will be studied in this particular order but before this study commences, there is an aspect of all communications system that must be considered — noise. Even in the most well-designed system, noise is an element that will always exist in the system. For the emulated system to accurately represented real-world system behaviour, noise is a concern that must be taken into account.

3.3 Noise

Noise is an ever-present factor in all practical electronic systems. It is therefore important to take into account the effect that noise has on the signal for the various front-end components. There are a variety of internal types of noises generated within electronic circuitry of which the most predominant include thermal noise, shot noise and flicker noise (Horowitz & Hill 1989).

Internal noise is the result of the random action of charge carriers within electronic components (Ziemer & Tranter 1995). In order to keep the noise model in the emulator as practical as possible, only the effects of a primary source of noise, thermal noise, will be considered.

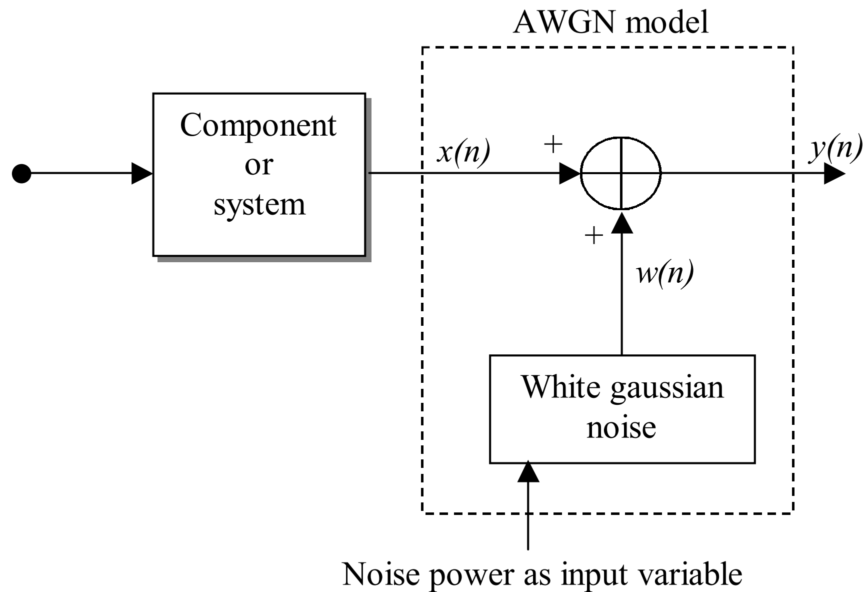


Figure 3.2: *Block diagram showing how noise is modelled in the system*

Thermal noise (also known as Johnson and white noise) occurs as the result of the random motion of charge carriers in a conducting or semiconducting medium (Ziemer & Tranter 1995). The equation that describes the open-circuit (rms) noise voltage generated by a resistance is given as (Horowitz & Hill 1989):

$$V_{\mathrm{N(rms)}} = \sqrt{4kTRB} \tag{3.1}$$

where k is Boltzmann's constant ($1.38 \cdot 10^{23}$ J/K) and T is the temperature in degrees Kelvin. R is the resistance of the device and B is the system bandwidth. Equation 3.1 shows the thermal noise voltage's dependence on resistance, temperature and bandwidth. In order to keep thermal noise to a minimum, these variables must be kept as low as possible.

Thermal noise can be considered as an additive white gaussian noise (AWGN) process since thermal noise has a uniform power spectral density and a gaussian distribution (Ziemer & Tranter 1995, Horowitz & Hill 1989).

3.3.1 Noise modelling considerations

Noise in the system is modelled as AWGN in order to represent thermal noise. This concept of modelling noise is illustrated in figure 3.2.

MATLAB's white gaussian noise function was used in the model prototypes. The effect of adding 5mW of noise power to an input signal with unity amplitude is shown in figure 3.3. This basic principle of generating gaussian distributed random numbers was

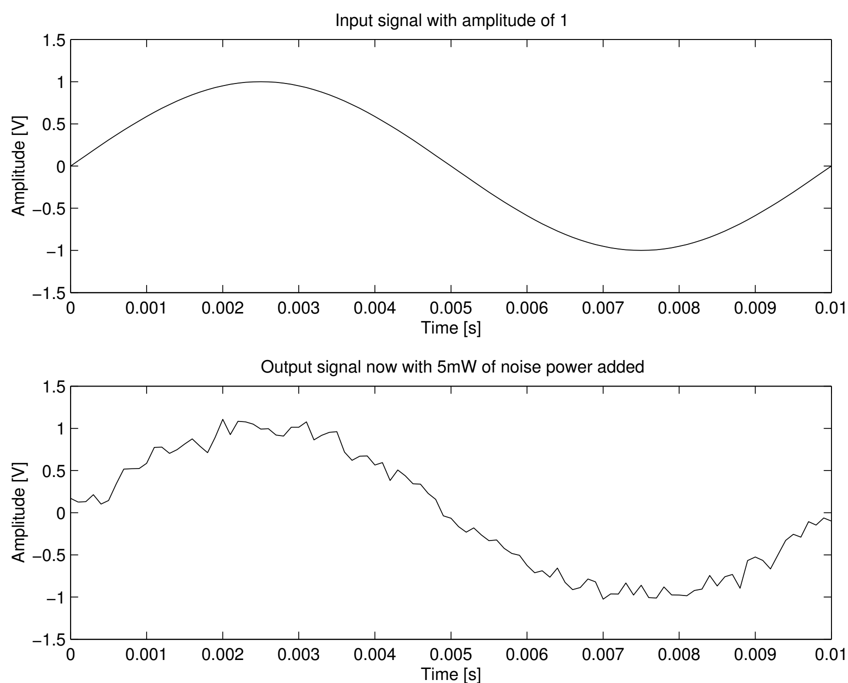


Figure 3.3: *Illustration of the addition of white gaussian noise to a signal*

later adapted and used in the C++ models. The output of the AWGN model can be described simply as:

$$y(n) = x(n) + w(n) \tag{3.2}$$

where $y(n)$ is the output of the model which is the sum of the input, $x(n)$, with the output of the wgn generator, $w(n)$. In order to have a working noise model in the final C++ implementation, an equivalent noise model implementation was written in C++. The actual syntax used in the C++ noise model to execute equation 3.2 is:

```
y = x + gasdev(&idum)*sqrt(noise_power)
```

The variable '`x`' represents the input signal sample. The `gasdev()` function (Press et al. 1992) generates a gaussian random number that is multiplied with the square root of the '`noise_power`' (to convert the noise power to a noise voltage) specified by the user. Noise is specified as a linear power level (in Watts), but the front-end application gives the user the option to specify this as a dB or a dBm value.

3.4 Data converters

Analogue-to-digital and digital-to-analogue (ADA) converters are fundamental building blocks in mixed-signal circuits and are seen as the key components in a SDR. It is impractical to try and cover every single type of converter architecture that exists today.

Instead, key fundamental architectures are examined. Some of the more complex architectures used today employ some, or a combination, of the principles listed below. There exists a host of theory for the architectures presented here and for this reason it is stated up-front what the objective of the study is.

The aim of evaluating the various ADAs is to examine and outline how the architectural makeup and operational principles contribute to the non-idealities present in ADA converters.

3.4.1 DAC/ADC classification and architectures

DAC classification and typical architectures

Typical DACs make use of either the a resistor string, ratioed current source or capacitor array architecture (Chen 1995). Their respective architectural compositions will be studied next in order to get an understanding of their distortion effects. This will in turn facilitate a better understanding of how to model a generic non-ideal data converter.

Resistor-string DACs: As the name implies, this DAC architecture uses a resistor string to generate a number of reference voltage levels. The advantage of this type of DAC is its simplicity. The major drawback of this type of architecture is the output resistance's dependency on the digital input code (van Rooyen 2000). White thermal noise, contributed by the resistors, is the primary noise source of this type of topology.

Current-ratioed DACs: Current-ratioed DACs are quite often used as stand-alone devices. There are two types of current-ratioed DACs, namely, the *weighted-current DAC* and an *R-2R DAC*. The R-2R DAC will be highlighted because its design addresses the shortcomings of the weighted-current DAC (Chen 1995).

The R-2R ladder DAC's resistor divider arrangement overcomes the large resistor problem presented by the binary-weighted DAC since only two types of resistor values are required. Figure 3.4 shows an example of a 1-bit R-2R ladder current steering DAC (van Rooyen 2000) that comprises a series R-valued resistor and shunt 2R resistors. The advantage of this R-2R ladder configuration is that if more bits are required it is just a matter of appending additional R-2R networks. The operation is based on the binary division of current flowing down the ladder. The analogue current source error determines how linear the DAC will be (Chen 1995). This analogue current source error occurs as a result of non-ideal switching effects of the transistors used to alternate between the bit states (van Rooyen 2000). The current noise is the primary source of error for a current-ratioed DAC and arises as a result of the shot noise in the current sources (Chen 1995).

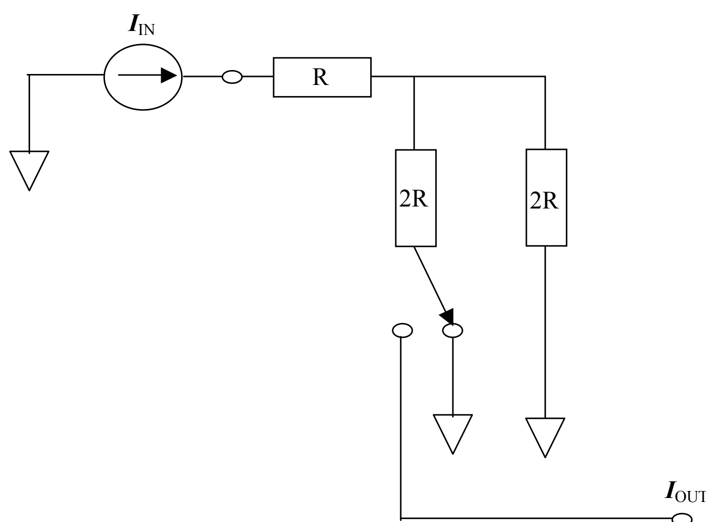


Figure 3.4: *Illustration of a 1-bit R-2R DAC (adapted from (van Rooyen 2000))*

Capacitor-array DACs: The principle of operation for a capacitor-array DAC is essentially based on the substitution of the resistor-string DAC using an accurate binary-weighted capacitor array (Chen 1995). The primary noise source is that of switching noise, referred to as kT/C noise.

ADC classification and architectures

ADCs can be classified as either Nyquist rate converters or oversampled converters.

Nyquist rate converters: Nyquist rate converters sample the input signal at just above the minimum sampling rate (Nyquist frequency). Nyquist converters can be further categorised according to the number of clock cycles needed to complete an analogue-to-digital conversion cycle. Thus, high speed, medium speed and high resolution converters will be classified as 1-clock, n-clock and 2^n clock converters respectively (where n is the number of bits of resolution).

Oversampling delta-sigma data converters: One of the important points to note about Nyquist converters is that their circuitry requires very accurate components in order to provide a high overall degree of resolution. This is not the case with oversampling converters. As the name implies, oversampling converters work on the principle that roughly sampling (low quantisation) the analogue input several times higher than the Nyquist rate and then feeding back the error signal in order to average out the quantisation noise (Candy & Temes 1992). The operational principle of oversampling converters favor the IC manufacturing processing in that it is cheaper and easier to manufacture a fast

digital IC than to produce an accurate analogue IC. The theory of oversampled delta-sigma data converters is covered extensively in a number of texts[1] and its complete study is beyond the scope of this thesis.

3.4.2 Ideal conversion

When developing an ADA model for emulation purposes, the ideal performance should be considered first in order to have a good reference. It is the various sources of error that cause the divergence from the ideal characteristics and this will be looked at additionally.

Ideal converter transfer function

The theoretical ideal transfer function for an ADC is a straight line (i.e. disregarding quantisation). The practical ideal transfer function for an ADC with quantisation, however, is a regularly-spaced staircase as will be illustrated for an ADC in the following section. A DAC theoretical ideal transfer function would also be a straight line with an infinite number of steps. Practically, however, the DAC transfer function is a series of points that are superimposed on the ideal straight line.

Ideal DAC transfer function: A DAC represents a finite number of discrete digital input codes by a corresponding number of discrete analogue output values. The transfer function of a DAC is a range of discrete points as illustrated in figure 3.5 (Texas Instruments 1995). The step height between consecutive samples in an ideal DAC transfer function is 1 LSB.

Ideal ADC transfer function: An ideal ADC, as illustrated in figure 3.6 (Texas Instruments 1995), uniquely represents all analogue input values by a limited number of digital output codes defined by 2^N, where N is the number of bits. Quantisation error is introduced because the continuous analogue scale is now being represented by a discrete number of points. This quantisation error will be looked at next in more detail as a separate entity to show its noise contribution to the converted signal. The width of one step is defined as 1 LSB.

Quantisation noise

Trying to represent (quantise) an infinite range of values (analogue domain) using a finite number of bits (digital domain) leads to a rounding off inaccuracy called quantisation error. This loss of information leads to an error contributing to the overall noise floor

[1] See the bibliography of (Candy & Temes 1992) for a comprehensive list of relevant references

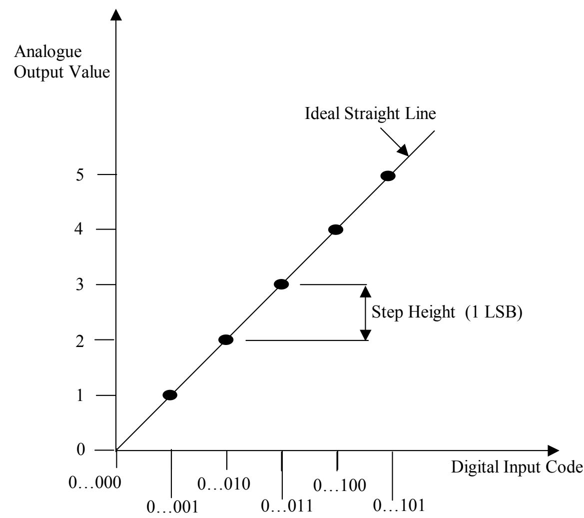


Figure 3.5: *Ideal DAC transfer function (Texas Instruments 1995)*

of the signal. In an ideal ADA, it is the only source of error present. Using a sine wave input to derive a theoretical maximum signal to quantisation noise error gives (van Rooyen 2000):

$$\text{SQNR} = 6.02N + 1.76 \text{ dB} \tag{3.3}$$

Equation 3.3 shows that, in order to better the SQNR of the system, more bits are required and that for every one bit added to the system, the SQNR is improved by approximately 6 dB.

Sample-and-hold distortion

The process of sampling is a fundamental requirement for both ADCs and DACs. For an ideal DAC, the concept of a pulse train of scaled impulses is used to represent the output of an of an ideal sampler to an arbitrary signal (van Rooyen 2000). In practical DAC's, this ideal pulse train is convolved with a normalised rectangular pulse (van Rooyen 2000). An important artifact of sample-and-hold circuits that must be considered in practical systems is the resultant gain response curve: a low pass filter response. If the system is not designed with this effect in mind, the output could experience attenuation near half the sampling frequency (van Rooyen 2000). However, although this type of effect is not explicitly taken into account for modelling purposes. If the DACs are modelled to have a sample-and-hold output, the sample-and-hold distortion will automatically be observed.

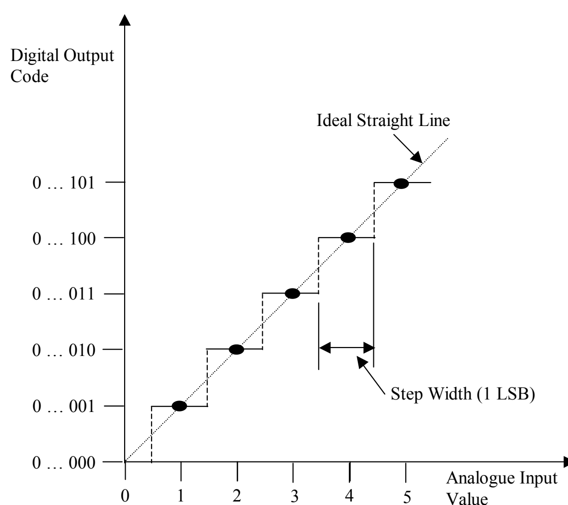


Figure 3.6: *Ideal ADC transfer function (Texas Instruments 1995)*

3.4.3 Non-ideal converter errors

A converter's performance can be evaluated according to either its static or dynamic error performance. The following definitions are often used to describe the performance of a given data converter (van Rooyen 2000, Texas Instruments 1995).

Static errors

Except for oversampling converters, the linearity of conventional converters are very dependent on the accuracy of the reference voltages and currents used. Internal resistance imbalances are a main contributor to the static (DC) error of a converter.

Offset error: This can be seen as the difference between the nominal and actual offset points of a converter. For a ADC, this means that for an analogue input of zero, the corresponding bit sequences should be zero. Similarly, a DAC output should read zero for a digital input of zero. Any non-zero values for either converter means that some offset error is present. Offset error is expressed as a percentage of the reference voltage.

Gain error: With the full-scale input voltage applied to the ADC (resulting in ones in the digital output code) or a digital code of all ones applied to a DAC, gain error is defined as the amount of deviation between the ideal transfer function and the measured transfer function (with the offset error removed). This is, in essence, a measure of the amount of gain drift of a given device. As with offset error, gain error is expressed as a percentage of the reference voltage.

Monotonicity: For a given device to be monotonic, an increase in the input must result in a corresponding output rise. For some low-performance ADA devices, monotonicity is not necessarily guaranteed. Inaccuracies in the binary weighting of a DAC causes non-monotonic errors and this can result in missing codes in an ADA. A monotonicity error of less than 1 LSB means that the converter is guaranteed to be monotonic.

DNL (Differential nonlinearity error): DNL error is the difference between a single step width (for an ADC) or step height (for a DAC) and the ideal value of 1 LSB. This means that there is no DNL error if the step height or width is exactly 1 LSB for a DAC and ADC respectively — something very desirable in practical data converters.

INL (Integral nonlinearity error): The integral nonlinearity is the divergence of the values on the actual transfer function from an ideal straight line. This straight line is usually drawn between the endpoints of the transfer function once the gain and offset errors have been nullified. For an ADC the deviations are measured at the transitions for consecutive steps, and for the DAC they are measured at each step. INL is actually therefore the cumulative effect of all the DNL errors, after the gain and offset errors have been removed.

Analogue component imbalances can result in the nonlinear INL response curve shown in Figure 3.7. INL is measured as the maximum positive and negative swing with reference to the ideal straight line. The shape of the INL curve can have a direct influence on the order of the harmonic spurs in the output signal (van Rooyen 2000). Some of the more common curve shapes are shown in figure 3.7 for which a bow shaped and S-shaped curve produce even-order and odd-order harmonics respectively (van Rooyen 2000).

Total unadjusted error: This is sometimes referred to as the *absolute accuracy* of an ADC or DAC. The measurement takes into account the effects of all the above-mentioned static errors. With reference to the ideal transfer function, the total unadjusted error is usually specified as a percentage of the full-scale voltage or current.

Dynamic errors

Spurious-free dynamic range (SFDR): SFDR is often used to describe the spectral quality of a generated signal. SFDR is a measure of the power difference between the fundamental signal and the magnitude of the largest undesired frequency component as illustrated in Figure 3.8. SFDR will often be used as a unit of measure for the hardware emulator to evaluate the quality of the output waveforms in the frequency domain. Other dynamic error performance measurements not listed here include, but are not limited to,

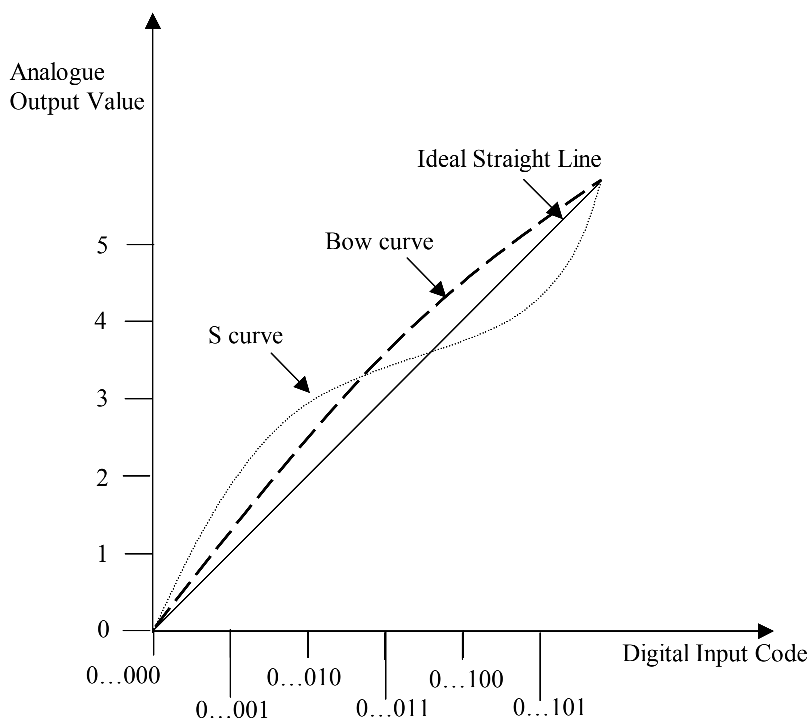


Figure 3.7: *An example of common DAC INL curves (adapted from (Crook & Cushing 1998))*

signal to noise and distortion ratio (SINAD) and total harmonic distortion (THD). The reader is referred to Chen (Chen 1995) for more insight into these concepts.

Glitch impulse: The study of glitch impulse relates to a DAC only. High-speed DACs, such as those used for software radio applications, require very accurate waveform reconstruction in order to keep the SFDR as low as possible. Glitch impulse, the consequence of transient switching effects, can have a serious limitation on the overall performance of a converter if not treated correctly (van Rooyen 2000). One source of glitch error is poorly synchronized bit switching times — the small switching time differences when the input bit codes drive the switching transistors (van Rooyen 2000, Garcia & LaJeunesse 1995). This results in momentary output current error which inturn shows up in the transient response as glitch impulse as illustrated in figure 3.9. Glitch impulse, expressed in pV/s, can be seen to be increasingly problematic during the DAC's major carry transitions. This is due to the switching of the most significant bit (MSB) and this small switch timing error can momentarily cause a large error in the current or voltage output. Although techniques have been employed to make the glitch impulse error constant over the entire output range (Garcia & LaJeunesse 1995), it is still an ever-present error contributor.

Charge injection from analogue complementary metal-oxide semiconductor (CMOS)

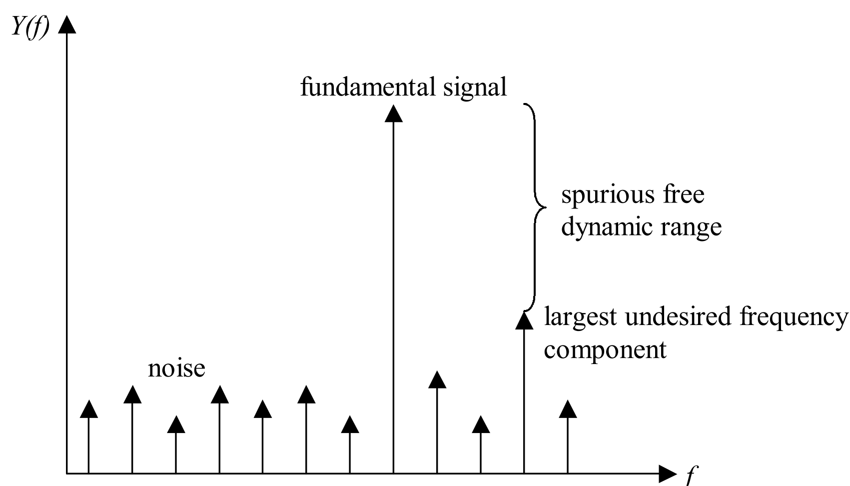


Figure 3.8: *The power difference between the desired signal and the largest unwanted component is known as the spurious free dynamic range (SFDR) (adapted from (van Rooyen 2000))*

switches (Eichenberger & Guggenbuhl 1991) is a second contributor of glitch impulse error. Although deglitching techniques exist to combat the effect of glitch impulse error, it is important to model the glitch impulse phenomena in order to observe the effect it has on the overall system — the technique used to achieve this is explained next.

3.4.4 Data converter modelling considerations

The ADA is a key component in a SDR system. The various architectures as well as the parameters that are important in defining the performance of an ADA were outlined above. In summary, the data converter theory covered indicates that:

- Even in an ideal ADA, there will always be quantisation noise.
- Component imperfections are one of the main reasons for a nonlinear conversion transfer function.
- Passive components used in most architectures contribute to an increase in thermal noise.
- Glitch impulse is one of the biggest performance restriction factors in DACs.

Although it would be quite possible to mathematically describe and model most of the possible sources of specific ADA architectures mentioned earlier (MALOBERTI, F. et al 1992), it goes beyond the scope of the thesis. Instead, the aim of the models presented here

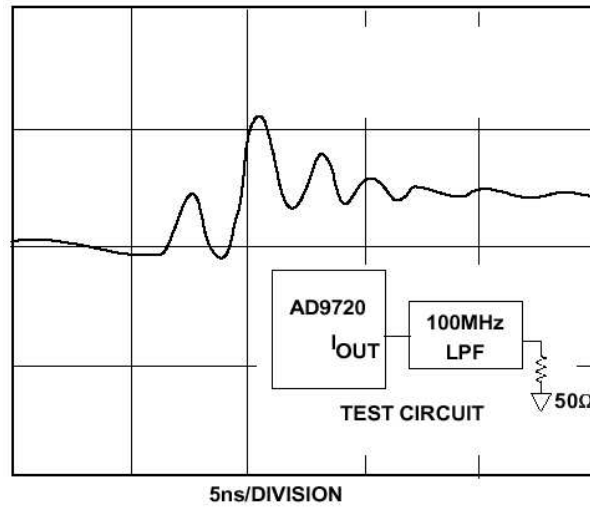


Figure 3.9: *Illustration of a typical DAC glitch on an output transition, taken from the Analog Devices AD9720 data sheet.*

is to try to develop a generic method that emulates the characteristics of the nonidealities generated by ADAs.

In order to develop a model for representing a generic ADA, the model was broken down into the following building blocks:

- quantisation
- INL predistortion
- glitch impulse

Quantisation

A quantisation function was developed in order to represent the rounding-off error created within an ADA. The syntax used in MATLAB to implement quantisation is:

```
y = sign(x).*(ceil(abs(x)*(q-1)/(2) - 1 + eps) + 0.5)/(q-1)*2;
```

The code above represents a linear (uniformly spaced) quantisation process. The quantisation code produces output levels symmetrical around, and not including, zero as shown for a 3-bit quantised sinusoidal waveform in figure 3.10. The quantisation code represents a normalised function and therefore the (bipolar) input signal must be in the range of ±1 in order for the code to operate correctly. The output swing corresponds to the total input range, which is also ±1. The output of the DAC can, therefore, be scaled to any desired level by cascading the DAC output to the input of an ideal amplifier. The quantisation model will be used to introduce quantisation error in both an ADC and a DAC.

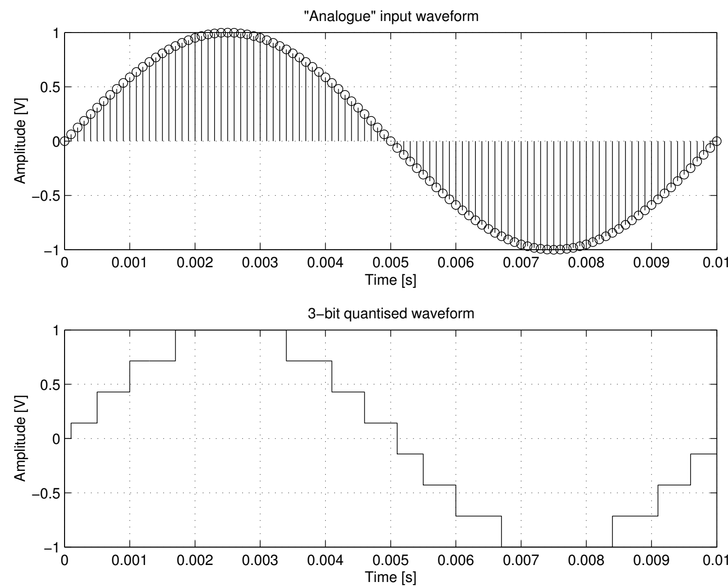


Figure 3.10: *3-bit quantisation model output*

INL predistortion

INL error, the cumulative effect of the deviations from the ideal conversion transfer function of an ADA occur because of imperfect component behaviour within the various ADA architectures. A method of describing a non-ideal curve in order to represent a nonlinear transfer function generally for an ADA could, therefore, be utilised.

One solution would be to use a lookup table. A lookup table would hold distorted output values that correspond to each ideal input value. Of course, the drawback of this approach can immediately be noted — a large computational memory requirement that increases with the resolution of the system. For example, a 16-bit system corresponds to a lookup table with 65536 entries! Apart from the computational memory considerations, entering these values by hand could be quite time-consuming.

A more elegant and less memory-intensive approach would be to employ a curve-fitting algorithm, such as the cubic spline interpolation function used by the INL predistortion model. This is achieved by specifying the INL curve shape in the form of a few key co-ordinate markers taken from the ADA transfer function. These points specify the shape of the nonlinear transfer function as shown in figure 3.11. The cubic spline interpolation function is used to calculate the intermediate points. This process is implemented by the `inl.m` model that can has the following syntax:

```
INLOut = inl(IN,x1,y1)
```

where the arrays `x1` and `x1` specify the x and y marker co-ordinate points respectively. The output, `INLOut`, is the distorted output of the input signal, `IN`, based on the values

of `x1` and `y1`.

Figure 3.11 illustrates a s-shaped curve, specified using only five curve markers shown as small circles on the nonlinear output curve. The MATLAB model requires that the user specifies as many markers (x and y co-ordinate values) as deemed necessary so that the interpolated INL curve is a good approximation of the measured INL. The two endpoints of the INL curve are always specified, and if these are the only markers used, then a straight line (zero INL) results. For a bow-shaped curve (recall figure 3.7), only one additional marker is required. The exact point at which these markers lie will vary, depending on the required degree of deviation from a straight line transfer function. More complex-shaped INL curves might require more than two markers, although this should be unlikely.

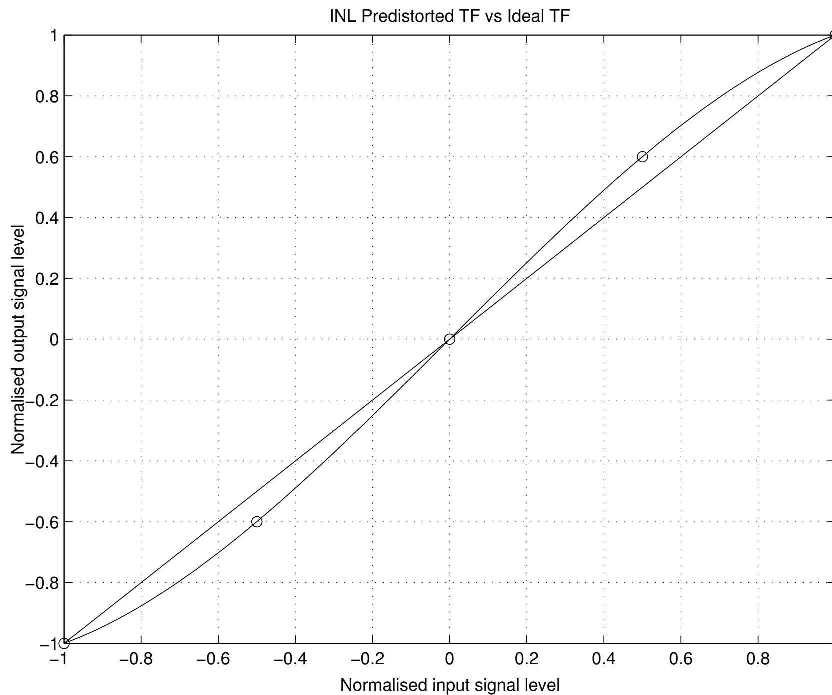


Figure 3.11: *Transfer function of an ideal and pre-distorted conversion*

Glitch impulse

Glitch impulse is a deterministic error in that for any given output transition, a fixed error will be associated with it. Therefore, glitches are code-dependent errors, and to calculate the exact amount of glitch for every possible code transition based on an actual DAC circuit model would be prohibitive for the model suitable for this thesis. Instead, certain assumptions are made in order to yield a simplified basis on with the glitch model is based, namely:

1. The glitch magnitude depends only on the last and new quantisation level.

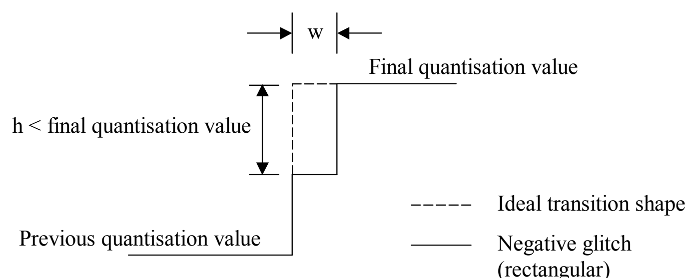


Figure 3.12: *Illustration of negative glitch impulse using a rectangular approximation.*

2. The underlying mechanism producing glitch (switching effects, charge injection) are not important if the effect that the glitch has on the signal is the consideration taken into account

3. Glitches have a characteristic shape that can be described using a small number of parameters, namely, a glitch width and height value.

These three points warrant further explanation. In reality, practical glitches depend on the last and new input code. Therefore, because glitches are code-dependent, they are not directly related to the quantisation level. Therefore, the first point is a reasonable assumption, that does not compromise the accuracy of the model in any way since the glitch impulse would appear to be pseudo-random. Since it is a behavioral glitch model being considered, the second point is a valid assumption. The last point assumes a very simplistic glitch shape. In reality, glitches have a much more complex shape as illustrated by figure 3.13. Most data sheets, however, only take the first major transient into account when specifying the level of glitch. In order to calculate this level, the glitch is assumed to be triangular in shape, specified only by a width and height value (Garcia & LaJeunesse 1995) as shown in figure 3.14 (i). Considering that this is the process used to measure glitch, it is therefore a reasonable assumption that the modelled glitch output can be specified by a width and height value. This height and width value in the glitch model therefore describes a rectangular shaped glitch output as illustrated in figure 3.14 (ii). Figure 3.15 shows the output of the glitch model with the assumption that after the required low pass filtering in a DAC, the transient effects will show, as illustrated by figure 3.17.

Because of the nature of the switching mechanisms for the generation of various code transitions in DACs, both positive and negative glitch impulse errors can occur — this effect is considered and accommodated for in the glitch model algorithm. An illustration of negative glitch is shown in figure 3.12. The glitch impulse error is therefore modelled as a square approximation using both positive and negative height values.

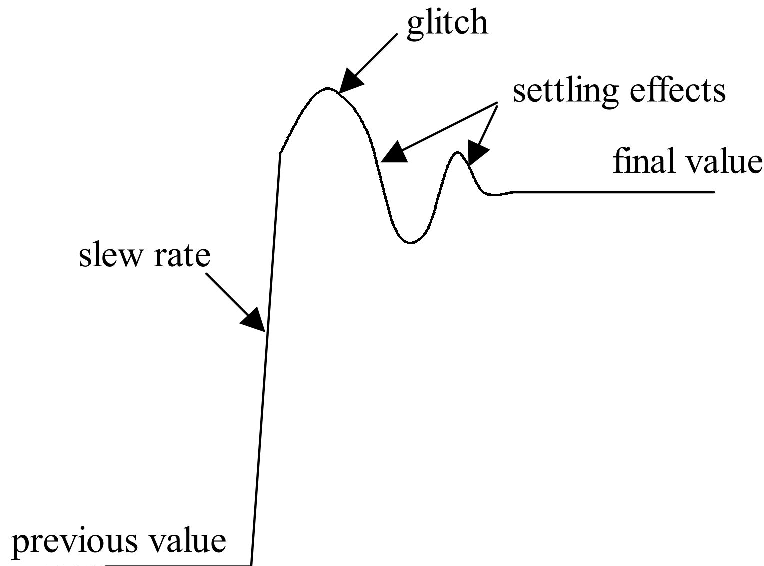


Figure 3.13: *DAC output response showing typical glitch characteristics (adapted from (Garcia & LaJeunesse 1995)).*

In order to generate the random values used to specify the height and width of the glitch, a hashing function is used which takes in three parameters:

1. The last quantisation value
2. The new quantisation value
3. A seed value

The actual syntax for the hashing function used to generate the glitch height and width values is given as (van Rooyen 2000)

```
[w(n), h(n)] = hash(last, x(n), seed)
```

where the variable `last`, which can be thought of as `x(n-1)`, and `x(n)` is the previous and current output values respectively. The result is a uniformally distributed width and height value of the associated glitch as a number between 0 and 1, with 0 representing no glitch and 1 representing maximum glitch width or height. The hashing function is a deterministic function in that given the same values for last, current and seed value, it will always return the same width and height (i.e. the same glitch shape). This corresponds to the deterministic nature of DAC glitches in that they are always the same for the same transitions. The hashing function also has the convenient property that the way glitches were allocated to these output transitions seem random — this corresponds directly to the unpredictable glitch shape in a specific DAC.

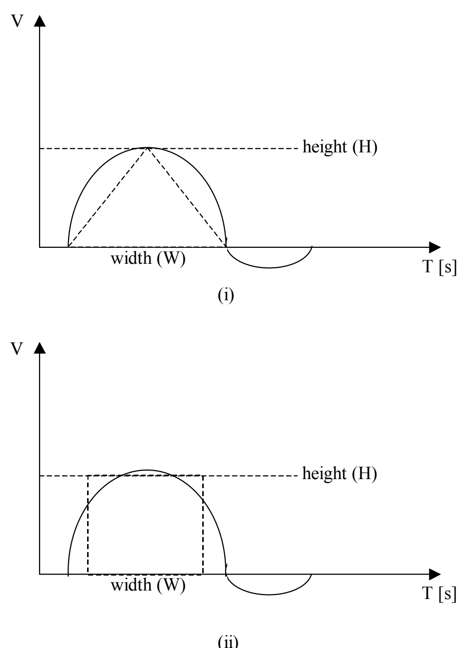


Figure 3.14: *Glitch area specified by a height and width value as triangular dimensions (i) and rectangular dimensions (ii).*

The `seed` value is included so that different DACs in an application can give different responses, and therefore unique results. This might be useful for monitoring the effects of mis-matched DACs such as those employed in a quadrature signal systems. The seed value can also be altered to obtain different output glitch values for different emulation runs.

The level of glitch impulse error that can occur on the output of a real DAC can vary between devices, depending on factors such as the sampling rate and architectural makeup of the device. A DAC that suffers glitch has a worst-case glitch of 1 MSB, which corresponds to half the total output DAC voltage swing in either direction. The glitch model assumes an input range of ± 1 and can, therefore, add a maximum glitch level of ± 1. Therefore, different seed values must be specified in order to represent different DACs in a system.

In practical DACs, glitch impulse is an analogue-observed effect. Therefore, the glitch impulse will only occur for a fraction of the DAC output. The output glitch effect can easily be observed in practice using an oscilloscope on the output of a DAC. In the digital domain, however, glitch impulse translates to a momentary effect that occurs between consecutive samples. The problem now is that there are no “filler” samples in-between in order to represent this glitch impulse. In order to address this problem, the input sampling rate is increased by an integer factor. With the aid of figure 3.18, this concept

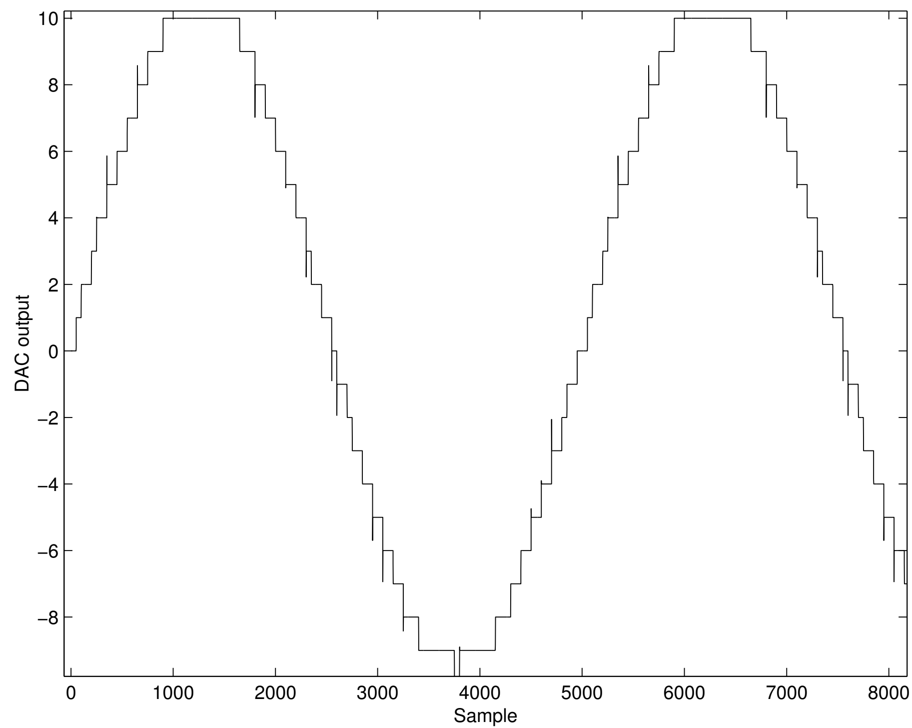


Figure 3.15: *Illustration (using a sinusoidal waveform) of the deterministic property of the modelled glitch impulse effect. The two cycles of the sinusoid were sampled at exactly the same points causing identical quantisation level transitions and identical glitches.*

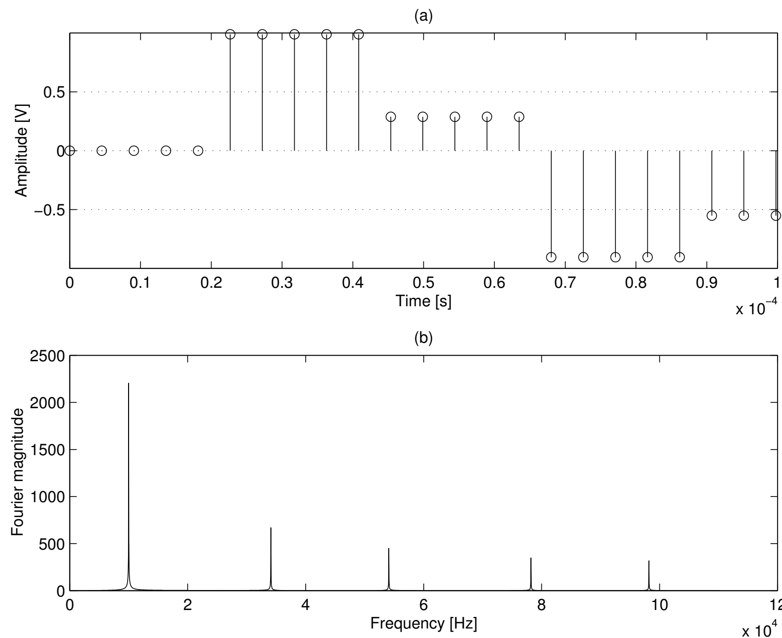


Figure 3.16: *Upsampled signal (a) with the resulting harmonics in the frequency domain (b)*

will be further illustrated. Figure 3.18 (i) shows two samples, a and b. Glitch impulse is an effect that would occur between samples and in order to represent this effect, the number of samples is increased as shown by figure 3.18 (ii) for an upsampling rate of six. The glitch model can now use the intermediate samples to represent the glitch impulse effect, as shown by figure 3.18 (iii). Note that the glitch width is represented by the number of samples, which is three in the case of figure 3.18 (iii). The glitch impulse model's hash function takes the upsampling rate into account and varies the glitch width in relation to this number. The upsampling process used here is sufficient in that just repeating the input sample by an integer amount generates the desired upsampling (zero-order-hold) harmonics.

This effect is illustrated in figure 3.18(a) which shows an example of a 100kHz waveform with a five times upsampling rate. The upsampling function generates the usual zero-order-hold harmonics (see figure 3.18(b)) that can be expected from practical DAC outputs. These harmonics are normally removed by the reconstruction filter in practical setups. It is also important to note that the upsampling rate should be high enough to include all significant frequency components (including the upsampling harmonics of interest) below half the Nyquist frequency.

In conclusion to the considerations taken for ADA modelling, it was shown that ADA will be modelled as a cascade of sub-models. These sub-models include a quantisation model, INL predistortion and a DAC glitch model. If an ideal ADA is therefore needed,

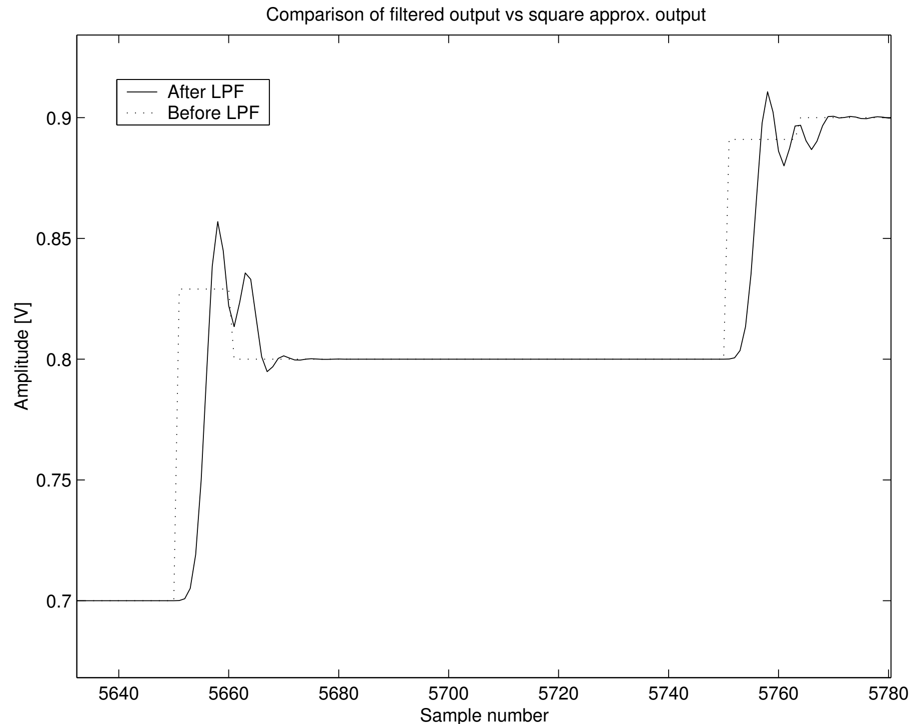


Figure 3.17: *After low pass filtering the glitch transient effects can be observed*

only the quantisation module will be used. But if, for example, a high-speed DAC with some nonlinear distortion and glitch impulse is required, then the combination of the three models can be cascaded as illustrated in figure 3.19.

3.5 Amplifiers

An ideal memoryless linear amplifier is one that adheres to the superposition theorem and can be described by the relationship

$$v_{\text{out}(t)} = K v_{i(t)} \tag{3.4}$$

where K is the voltage gain of the amplifier. This results in a straight line output-to-input characteristic curve.

Practical amplifier output values saturate at some input signal level. Fig. 3.20 (from (Hjorth & Hvittfedt 2002)) illustrates this effect, as well as other fundamental amplifier parameters which will be explained further on in the text.

The nonlinear output-to-input characteristic can be described by the Taylor series as (Couch II 1995):

$$v_{\text{out}} = K_0 + K_1 v_i + K_2 {v_i}^2 + \cdots \sum_{n=0}^{\infty} K_n {v_i}^n \tag{3.5}$$

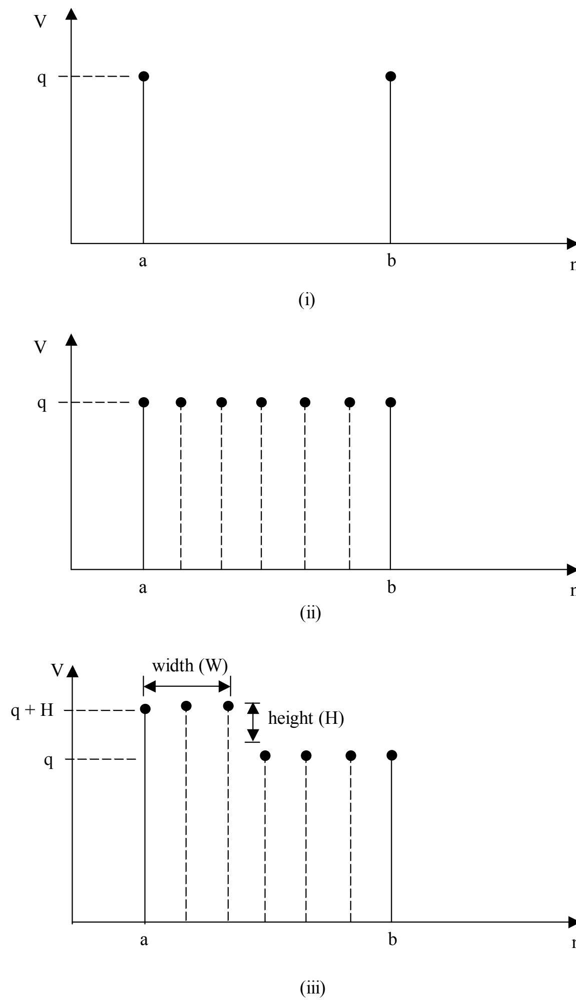


Figure 3.18: *Illustration of the requirement to increase the number of samples in order to represent glitch impulse*

where

$$K_n = \frac{1}{n!}\left(\frac{d^n v_o}{d{v_i}^n}\right)\Bigg|_{v_i=0} \tag{3.6}$$

By comparing Eq. (3.4) and Eq. (3.5) it can be seen that nonlinear effects start to show if any K_n terms are nonzero for $n > 1$. K_0 is known as the DC term and $K_1 v_i$ and $K_2 {v_i}^2$ are known as the first-order (linear) and second-order (square-law) terms respectively (Couch II 1995).

3.5.1 Saturation and clipping

Supply voltages and physical limitations set the limit on the maximum output swing of an amplifier. As stated earlier, this limit is called the saturation point of an amplifier, and when exceeded results in the waveform being clipped. This clipping in turn results

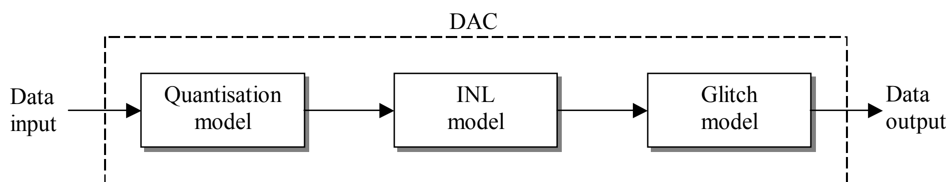


Figure 3.19: *A DAC can be implemented as the cascade of sub-model building blocks*

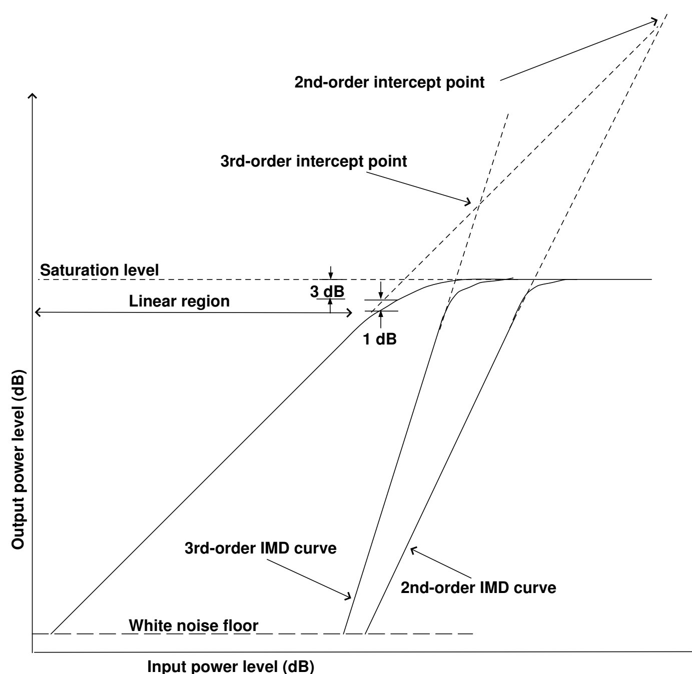


Figure 3.20: *Charateristics of an amplifier output (adapted from (Hjorth & Hvittfedt 2002))*

in harmonics that spread across the frequency spectrum due to the sharp transitions in the clipped waveform. Harmonics now appear at multiples of the fundamental frequency because the clipped waveform starts looking like a square wave. This phenomenon is illustrated in Figure 3.21. Practical amplifiers, however, don't always saturate at perfectly symmetrical positive and negative thresholds. The amplifier model should therefore include this option if it is required by the designer.

3.5.2 1-dB compression point

As saturation starts to occur in an amplifier, the gain starts to deviate from the ideal. The point at which the actual gain is 1 dB lower than the ideal, is called the 1 dB compression point (Kenington 2000). The 1 dB compression point gives an approximation of when an

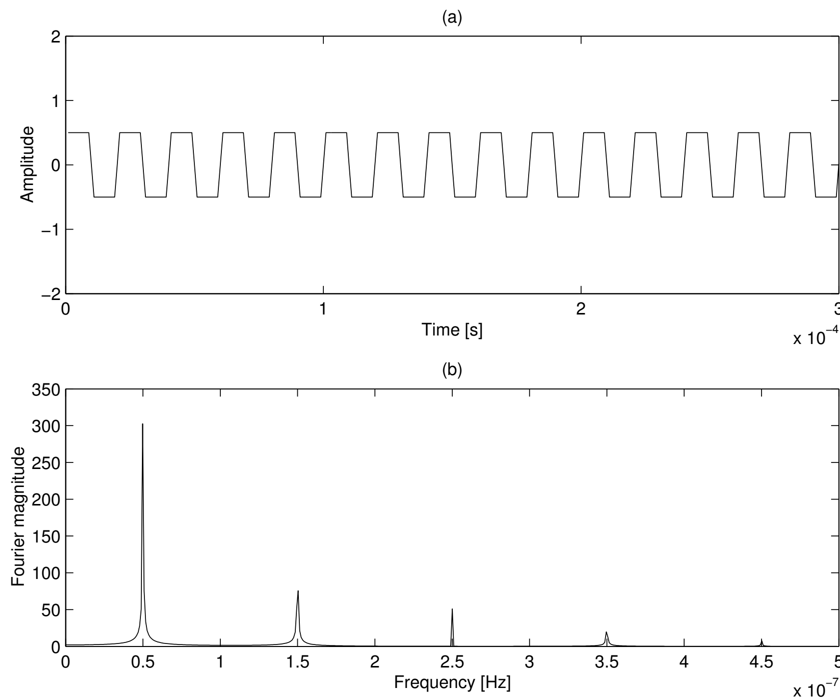


Figure 3.21: *Time (a) and frequency (b) plot for a saturated sinusoidal waveform*

amplifier's linearity has been lost, as shown in Figure 3.20. The 1 dB compression point is related to saturation in that it is approximately at 3 dB above the 1-dB compression point at which clipping of the waveform begins (Couch II 1995).

3.5.3 Harmonic distortion

By applying a single-tone test signal, $v_{i(t)} = A_0 \sin \omega_0 t$, the second-order harmonic distortion components of an amplifier can be observed. Substituting this signal into the second-order term of equation 3.5 gives:

$$K_2(A_0 \sin \omega_0 t)^2 = \frac{K_2 {A_0}^2}{2}(1 - \cos 2w_0 t) \tag{3.7}$$

From equation 3.7 it can be seen that a DC level of $\frac{K_2 {A_0}^2}{2}$ is created as well as a frequency component at double the input frequency with an amplitude of $\frac{K_2 {A_0}^2}{2}$ as illustrated in Figure 3.22 for an amplifier with a second-order nonlinearity. This leads to the general expression (Couch II 1995) for single-tone input for a n_{th} order nonlinearity (from Eq. 3.5) as

$$v_{\text{out}(t)} = v_0 + v_1 \cos(\omega_0 t) + v_2 \cos(2\omega_0 t) + v_3 \cos(3\omega_0 t) + ... \tag{3.8}$$

where v_n is the peak amplitude at the frequency $n \cdot f_{\text{in}}$. Equation 3.8 shows that harmonics occur at integer multiples of the fundamental frequency. Harmonics are always present

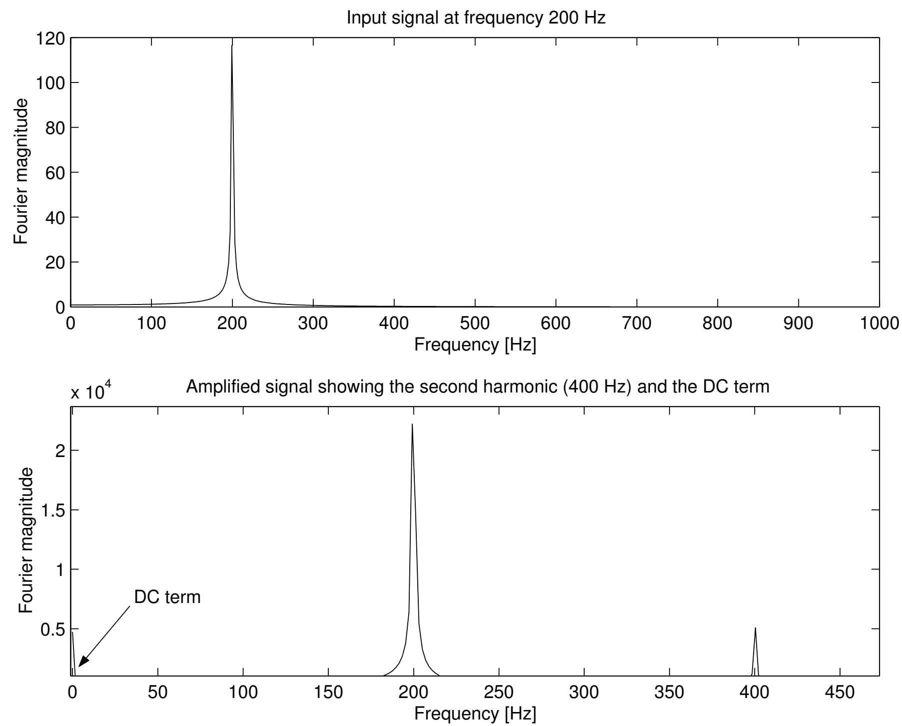


Figure 3.22: *Frequency response of an amplifier with a second-order nonlinearity to a single-tone input*

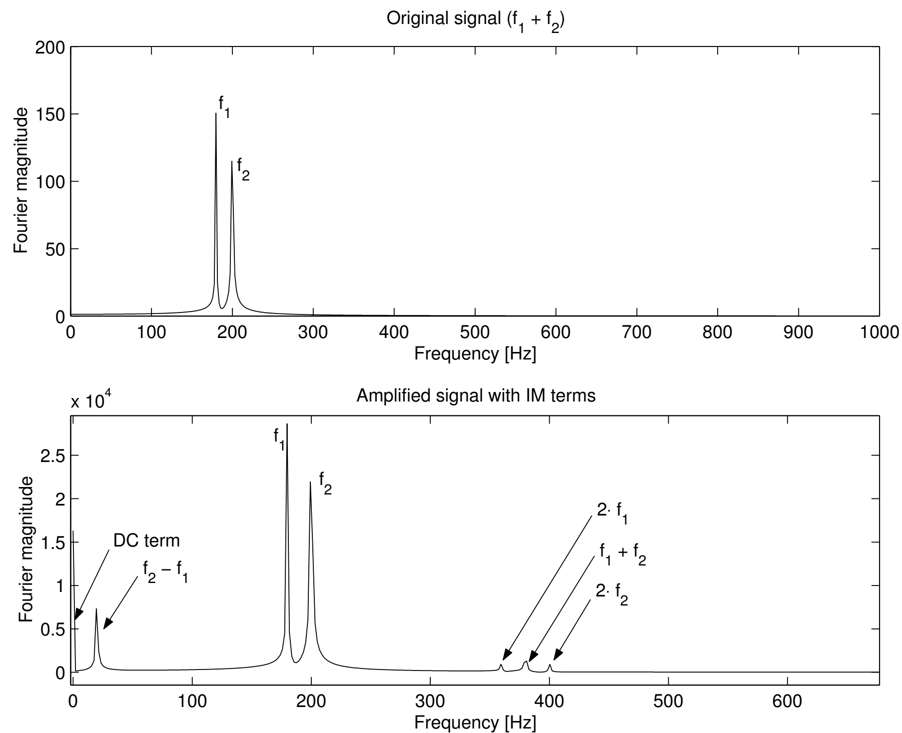


Figure 3.23: *Frequency spectrum of an input two-tone signal (a) and the resulting spectrum when passed through an amplifier with a second-order nonlinearity (b)*

in an amplifier output and are amplified just like the wanted signal. Harmonics occur as a result of amplifier nonlinearities. The intensity of the second-order one-tone harmonics depends on the level on the input signal and this measurement is specified by the *one-tone second-order intercept point*(IP_1).[2] This is illustrated in Figure 3.20 and it can be seen that the amplitude of the second-order harmonics increases as the square of the input signal level. This is as opposed to the amplitude of the fundamental signal that increase in proportion to the input signal level according to the specified gain.

When the input is a two-tone test signal at frequencies, f_1 and f_2, then the second-order nonlinearity acts like a mixer in that it not only produces even harmonics of the input signals, but also the sum $(f_1 + f_2)$ and difference $(f_1 - f_2)$ frequencies. This is illustrated in Figure 3.23 for an amplifier with a second-order nonlinearity with a two-tone test signal supplied at the frequencies $f_1 = 180$ Hz and $f_2 = 200$ Hz.

3.5.4 Intermodulation distortion

The intermodulation distortion components of an amplifier are specified by applying a two-tone input test signal. Probably the most significant of the two-tone products are the third-order components. equation 3.5 will be used to illustrate why this is so.

If the third-order term of equation 3.5 is evaluated by supplying a two-tone test signal $v_{i(t)} = A_1 \sin \omega_1 t + A_2 \sin \omega_2 t$, as the input so that

$$K_3 {v_i}^3 = K_3 (A_1 \sin \omega_1 t + A_2 \sin \omega_2 t)^3 \tag{3.9}$$

Expanding equation 3.9 will reveal not only the expected harmonic distortion terms but also cross-product terms. The most important of these cross-product terms are those with frequencies at the points $2f_2 - f_1$ and $2f_1 - f_2$. This is because these intermodulation (IM) terms could pose serious problems in a system, since the closer f_1 and f_2 are, the closer these difference IM terms will be to the desired frequencies in the passband. The spectrum for a two-tone test signal (at the frequencies $f_1 = 180$ Hz and $f_2 = 200$ Hz) passed through an amplifier with only a third-order nonlinearity is illustrated in Figure 3.24. Here the problematic in-band frequency components at $2f_1 - f_2$ and $2f_2 - f_1$ as well as the other less significant IM products can be seen.

In practical amplifier design in general, the presence of these IM products are unavoidable, and it is therefore important that the level of these third-order products be kept to a minimum. This can be done by ensuring that the input signal level stays below a specified level given by the amplifier's third-order intercept point. Another option is the use of advanced amplifier linearisation techniques (Kenington 2000).

[2]The intercept point may be quoted as either an input (for receiver front-ends) or output (for power amplifiers) and is specified as a power (dBm) (Kenington 2000)

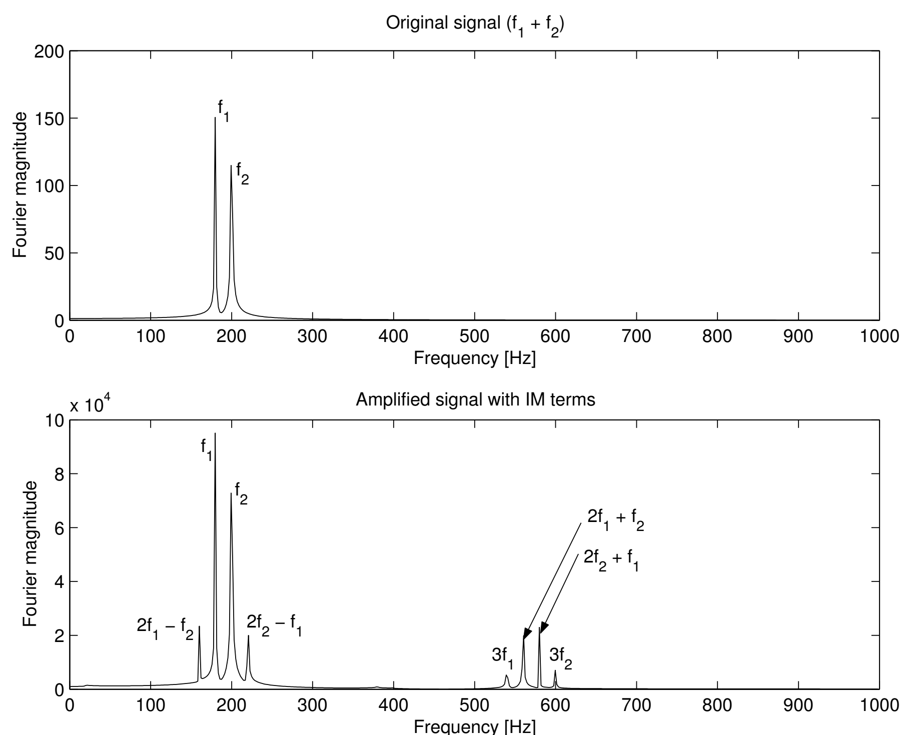


Figure 3.24: *Original signal spectrum (a) with the resulting spectrum when passed through an amplifier with a third-order nonlinearity (b)*

As with the wanted signal, the third-order products are amplified and the third-order intercept point (IP_2) specifies the level of the two-tone third-order harmonics for a given input signal level.[3] This would indicate, for example, what the maximum input should be (usually specified in dBs) to ensure that the third-order IMD products are to be kept below a certain desired level. A common measurement of IMD is performed by taking the ratio of the highest IM product to the amplitude of one of the two (equal valued) test tones (Kenington 2000).

In general, the distortion terms created for any order two-tone nonlinearity can be determined by the following relationship (Kenington 2000):

$$f_{\mathrm{IMP}} = m \cdot f_1 \pm n \cdot f_2 \tag{3.10}$$

where m, n = 1,2,3... and f_{IMP} is the frequency of the IM product. The sum of m and n is the order of distortion. If the above third-order distortion components were taken, for example, where f_1 = 180Hz and f_2 = 200Hz, then the IM frequencies can be calculated according to Table 3.1

[3]As with the one-tone second order harmonics, the intercept point is actually a fictitious point obtained by extrapolating the linear regions of the fundamental curve and, in this case, the IMD curve until the point at which they intersect.

Table 3.1: *Calculation of the third-order IM terms*

$\mathbf{m}\cdot f_1$	$\mathbf{n}\cdot f_2$	$m + n$	**Resultant frequencies**
$0\cdot f_1$	$3\cdot f_2$	3	$3f_2$
$3\cdot f_1$	$0\cdot f_2$	3	$3f_1$
$2\cdot f_1$	$1\cdot f_2$	3	$2f_1$+f_2 and $2f_1$-f_2
$1\cdot f_1$	$2\cdot f_2$	3	f_1-$2f_2$ and f_1+$2f_2$

The frequency components calculated in Table 3.1 can once again be observed in Figure 3.24

3.5.5 Amplifier modelling considerations

General considerations

In order for an amplifier model to function in the emulator, certain criteria had to be set. These include simplicity, computational efficiency and model extensibility (being able to upgrade the model without completely rebuilding it). Previous amplifier models studied had the advantage of a buffered set of data points which simplified the modelling considerably (Hjorth & Hvittfedt 2002). In the case of the emulator, data will arrive on a sample-by-sample basis, and a review of appropriate amplifier models for this purpose will be looked at next, based on the work done by Kenington (Kenington 2000).

Unless stated otherwise, the following models are all bandpass memoryless nonlinear models. This means that only the analysis of the signals in a certain band (first harmonic region) is considered and that no feedback is taken into account.

The Taylor expansion in equation 3.5 was useful for mathematical modelling, but its practical value is limited since it only shows the amplitude information of an amplifier's nonlinearity.

Both amplitude and phase nonlinearity, respectively called AM-AM and AM-PM conversion, are a more useful and important measurements of amplifier performance. Linear modulation schemes such as quadrature amplitude modulation (QAM) and quadrature phase shift keying (QPSK), in which the data is transmitted using both amplitude and phase variation in the RF signal, could be employed in a SDR configuration. These modulation schemes will require a suitable model to represent both amplitude and phase response and it is for this reason that a modification of the Taylor series in equation 3.5 is used.

This is achieved by using two Taylor series: one representing the I channel information and another series for the 90° phase-shifted Q channel information. This model is now called a Quadrature Taylor series amplifier model and is illustrated in figure 3.25

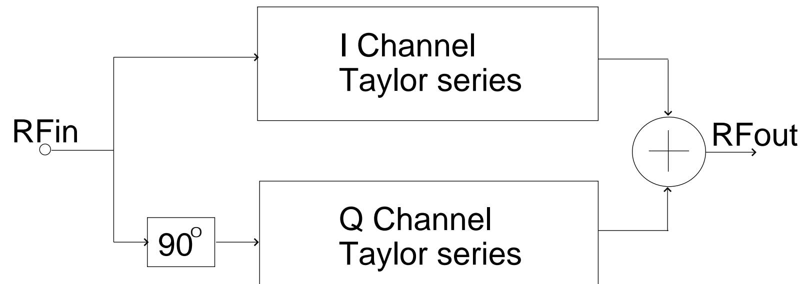


Figure 3.25: *Quadrature Taylor series amplifier model (Kenington 2000)*

(Kenington 2000).

The I and Q channel Taylor series are respectively (Kenington 2000)

$$V_I = k_I + a_I V_{in} + b_I V_{in^2} + c_I V_{in^3} + d_I V_{in^4} + \cdots \tag{3.11}$$

$$V_Q = k_Q + a_Q V_{in} + b_Q V_{in^2} + c_Q V_{in^3} + d_Q V_{in^4} + \cdots \tag{3.12}$$

where k_I and k_Q are DC offset terms. In an amplifier without phase shift error, the Q coefficients would be zero.

The feature of the quadrature Taylor series amplifier model is the simplicity and relative accuracy of the analytical functions used to model the nonlinear response of an amplifier (Kenington 2000). Another advantage of the quadrature Taylor model is that the relative level of each distortion component can be observed just by looking at the coefficients of each series. It is the order of these coefficients that determine the accuracy of the model — the more polynomial coefficients supplied, the more accurate the model. A drawback of the quadrature Taylor model is its inability to accurately represent the 1 dB compression point (Kenington 2000). This is not to much of a problem in the emulator since a separate saturation function can be inserted after the amplifier output.

In the pursuit of a suitable amplifier model, more complex models such as polar and cartesian types can be used at the expense of complexity and the requirement for accurately measured amplifier data (Kenington 2000). Like the Taylor series model, the Saleh model is an approximate form of the more complex polar and cartesian models (Kenington 2000). Although the Saleh model is based upon a single equation, this simplification results in its inability to track a highly nonlinear response (such as in a class-C amplifier response).

The choice of using the quadrature Taylor series model for amplifier modelling in the emulator was driven mostly by the relative simplicity of implementing the model, since the only parameters needed are the Taylor coefficients.

A consideration that affects all amplifiers is frequency response. This effect can be modelled by including feedback in the model. A Volterra series (Kenington 2000) is a

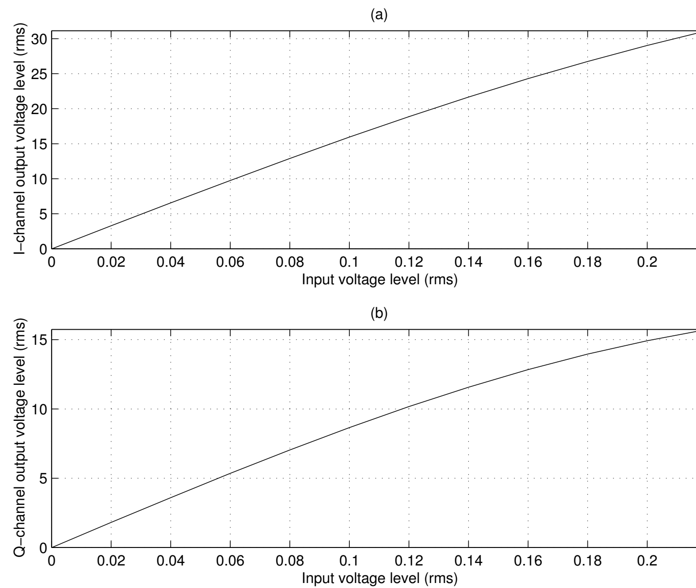


Figure 3.26: *Class-A amplifier I (a) and Q channel (b) response for a* 7th *order quadrature Taylor series amplifier*

modification of the Taylor series to include memory — this in turn also gives rise to increasingly complex equations. In order to retain a relative degree of simplicity, the bandwidth of a given amplifier can be modelled by coupling a filter to the amplifier's input.

In order to develop a working prototype model of the quadrature Taylor series amplifier, some development work was done in MATLAB. The syntax used to describe the Taylor model is written as:

```
Y = Taylor(X,I,Q)
```

The operation of the model is simply a direct interpretation of equations 3.11 and 3.12. The `I` and `Q` vectors represent the I channel and Q channel Taylor coefficients respectively. The output, `Y`, gives the amplitude and phase response of the amplifier for the input vector `X`. For the case, however, where only the amplitude response is required, the function only needs the I channel Taylor coefficients.

The amplifier model was tested using seventh-order I and Q-channel coefficients that were obtained from (Kenington 2000), which is that of a 900-MHz 40 W class A amplifier. The I and Q channel response is shown in Fig. 3.26, which clearly shows the nonlinear transfer function for both channels.

For the Fig. 3.26 plot, there was the convenience of using Taylor coefficients at hand. The ideal requirement, however, is to find a suitable relationship between the harmonic data supplied by manufactures amplifier data sheets and the corresponding Taylor coefficients. For the purpose of this thesis, this rule for converting the data sheet harmonics to

their Taylor coefficients will be illustrated for a general third-order Taylor series. Recall from equation 3.5 that a third-order Taylor series can be written as

$$v_{\text{out}} = K_0 + K_1 v_i + K_2 {v_i}^2 + K_3 {v_i}^3 \tag{3.13}$$

where v_{out} is the output signal. K_n is the Taylor coefficients and v_i is the input signal. Let the input test tone, v_i, be represented by

$$v_i = \cos(\omega_m t) \tag{3.14}$$

then substituting v_i into equation 3.13 yields

$$v_{\text{out}} = \overbrace{K_0 + \frac{K_2}{2}}^{h_0} + \overbrace{K_1 \cos(\omega_m t) + \frac{3K_3}{4} \cos(\omega_m t)}^{h_1} + \overbrace{\frac{K_2}{2} \cos(2\omega_m t)}^{h_2} + \overbrace{\frac{K_3}{4} \cos(3\omega_m t)}^{h_3} \tag{3.15}$$

From equation 3.15 it can be seen that the relating harmonics are

- $h_0 = K_0 + \frac{K_2}{2}$
- $h_1 = K_1 + \frac{3K_3}{4}$
- $h_2 = \frac{K_2}{2}$
- $h_3 = \frac{K_3}{4}$

We want to find out how the harmonics are related to the Taylor coefficients which means that the Taylor coefficients must be made the subject of the formula. Therefore, making K_n the subject of the formula as a function of the harmonics gives

- $K_0 = h_0 - h_2$
- $K_1 = h_1 - 3h_3$
- $K_2 = 2h_2$
- $K_3 = 4h_3$

In other words, K_0, the DC component of the Taylor series, is calculated as the amplitude of the harmonic h_0 (DC) minus the amplitude of the second harmonic, h_2. The example above was illustrated for only a $4th$-order harmonics series but typical Taylor amplifiers harmonics responses could contain lengths much greater than this. Higher-order Taylor series can, however, be solved in a similar way. This can be achieved by obtaining the general formula for equation 3.15 and solving the resulting simultaneous equations. An appropriate method (e.g. Cramer's method (WEISSTEIN, E.W 2003)) can then be used to solve the simultaneous equations.

Saturation

In order to implement a saturation modelling block in an amplifier, a threshold value is required. At the lower implementation level, the specified saturation value will be interpreted directly by the model. Stated otherwise, if a saturation level of 0.5 is specified (recall figure 3.21) then all values exceeding this threshold will be clipped off at the threshold value. Therefore, although saturation occurs gradually in practical amplifiers, the threshold values for the saturation model will define hard limits.

A suggestion for specifying saturation at a higher level in amplifiers is by obtaining the threshold value from the 1-dB compression point (usually found in manufacturing data sheets). The threshold value can be taken 3 dB above this point since this gives a good idea as to where saturation occurs (Hjorth & Hvittfedt 2002). This idea for converting the 1-dB compression point to a threshold value at a higher programming level, as well as the saturation flow diagram is shown in figure 3.27. The syntax used that describes the implementation of the saturation model is given as

```
y = saturate(x,k)
```

where the output `y` is the result of the input signal, `x`, saturated at the threshold limits, ±`k`. The model proposed here is for a single threshold saturation value. This assumes that the saturation occuring will always be symmetrical. In practical amplifiers, this is not always the case, as was stated earlier. Accomodating two separate saturation levels is simply a matter of a small modification to the amplifier model saturation algorithm. Therefore, if this property of non-symmetrical saturation is required, the designer can quite easily accomodate this with a slight modification of the amplifier code.

Noise

The noise contributed by an amplifier can be implemented by summing the amplifiers output to that of the AWGN noise model as explained in section 3.3 on page 18.

Amplification

The amplifier gain is specified by the coefficient of the first-order (linear) term of the Taylor series as in equation 3.5. Stated otherwise, only the second term of the I channel Taylor series would contain a gain constant and all remaining terms are zero. For an ideal amplifier, therefore, this is the only coefficient required.

3.6 Mixers

The cost of high-sampling data converters in a software radio can become quite prohibitive if the data converter were to sample directly at the RF signal. Instead, analogue frequency

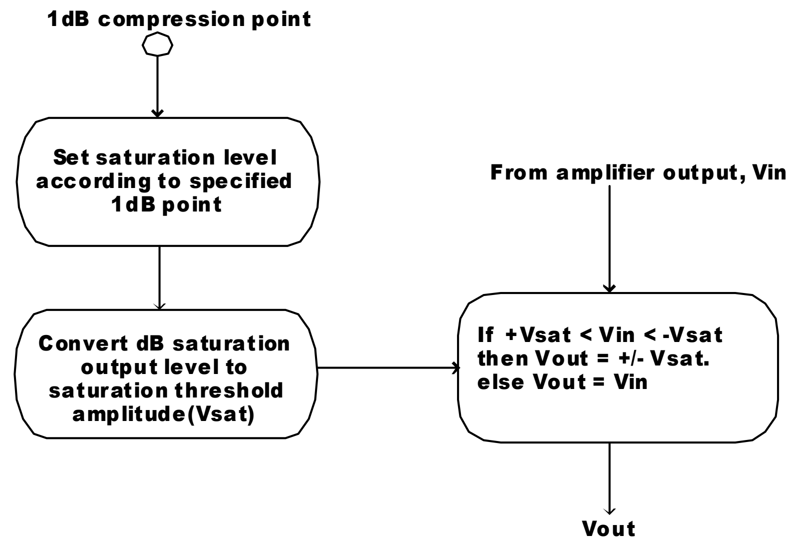


Figure 3.27: *Flowchart of saturation algorithm*

up and downconversion is still used to translate the desired signal to a frequency that is more suitable for the use of cost-effective data converters. In order to perform this frequency translation, some type of mixer is used. Mixer theory is a comprehensively covered and well-understood area documented by a number of texts (Egan 2000, Kroupa 1973, Couch II 1995, Rhode & Whitaker 2001) and it is not the purpose of this thesis to pursue this area in detail. An ideal mixer is one that performs the mathematical time-domain multiplication of two input signals (Couch II 1995). A typical downconversion mixer consists of two input ports (RF, LO) and a single output port (IF). In the ideal case when two input signals, f_1 and f_2 are mixed, only sum and difference frequencies appear on the output according to the trigonometric relationship

$$\cos(\omega_1 t)\cos(\omega_2 t) = \frac{1}{2}\left[\cos(\omega_1+\omega_2)t + \cos(\omega_1-\omega_2)t\right] \tag{3.16}$$

In the case of the downconversion mixer, subsequent filtering is performed in order to select the downconverted $(f_1 - f_2)$ IF frequency. An interesting point to note is that unlike the analogue mixing process, digital mixing is near ideal and produces only the sum and difference frequencies (Hosking 1998).

Although many alternative methods exist (Couch II 1995), the nonlinear transfer characteristic of a device (e.g. a diode) is often used in mixers to obtain the desired frequency components in equation 3.16. Practical mixer circuits also generate unwanted (spurious) frequency components (including the desired signals described by equation 3.16) called intermodulation products. There are two types of intermodulation products (Faria & Svensen 1995):

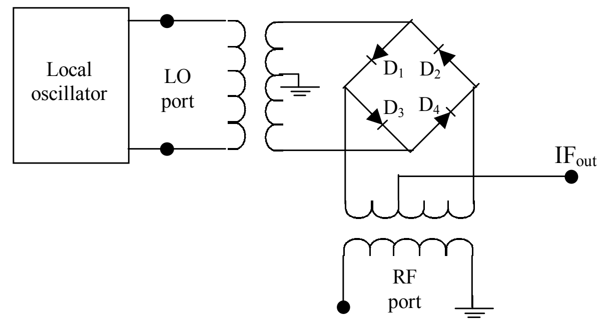


Figure 3.28: *A double balanced mixer circuit (adapted from (Hjorth & Hvittfedt 2002))*

- Single-tone: where the mixer consists of a single input signal plus the LO.
- Multi-tone: where more than one input signal to the mixer (and the LO) is used.

The single-tone IM components can be described as (Faria & Svensen 1995)

$$f_{\text{out}} = |nf_{LO} \pm mf_{RF}| \tag{3.17}$$

where m and n in Eq. (3.17) are integers $(0, 1, 2, 3 \ldots)$. The output signal for the mixer is f_{out}. The signals f_{LO} and f_{RF} are the LO and RF frequencies respectively. Of course, for upmixing, the input to a mixer can be either a RF or a IF signal, but without any loss of generality, the RF signal will be used in this analysis. The output for multi-tone IM products can be described as (Faria & Svensen 1995)

$$f_{\text{out}} = \left|(\pm n_1 f_{\text{RF}_1} \pm n_2 f_{\text{RF}_2} \pm n_3 f_{\text{RF}_3} \ldots \pm mf_{\text{LO}}\right| \tag{3.18}$$

where n_1, n_2, n_3 and m all are integer values from zero upwards.

Many types of mixers exist on the market today and can be broadly categorised as either passive or active mixers. Although active mixers have the desired property of gain, passive mixers are still very popular due to their simplicity and reliability at high and very high frequencies (Kroupa 1973). In order to gain an understanding of the kind of non-ideal effects that the practical mixing process yields, this study will fall on passive mixers and, in particular, the double-balanced diode mixer as shown in figure 3.28 (Hjorth & Hvittfedt 2002).

The double-balanced mixer is a good example of a general mixer used in practice, and is both inexpensive and offers excellent performance (Couch II 1995). For modelling considerations, both standard and quadrature modes will be investigated. After all, a quadrature mixer is essentially made up of two standard mixing stages. Only the quadrature mixer model will, however, be employed in the final test setup, but, for completeness, methods of modelling a standard mixer will be presented as well.

The input RF signal level to the double-balanced mixer is small compared to that of the input level of the LO signal. The high LO drive signal in fact turns the diodes on and off so that they act like switches. This, consequently, causes the LO signal to take the form of a square wave switching function that can be described by the Fourier expansion of a square wave (with unity amplitude) as (Miller 2001):

$$V_{\mathrm{LO}} = \frac{4}{\pi} \sum_{n=1,3,5,\cdots}^{\infty} \frac{1}{n} \sin(n\omega_{\mathrm{LO}}t) \tag{3.19}$$

The mixing processes of the double balanced mixer is called bi-phased modulation and can be mathematically represented by the product of the input RF signal, $V_{RF}\sin(\omega_{RF}t)$ and the LO signal described by equation. (3.19) to give the output mixer voltage as (Hjorth & Hvittfedt 2002)

$$V_{out} = V_{RF}\sin(\omega_{RF}t)\frac{4}{\pi} \sum_{n=1,3,5\cdots}^{\infty} \frac{1}{n} \sin(n\omega_{LO}t) \tag{3.20}$$

Equation 3.20 shows that the double-balanced mixer is designed to cancel out the even harmonics of the RF and LO frequency and, as a result, has very little spurious response (Rhode & Whitaker 2001). This is because n in equation 3.17 is now of an odd order so that $n = 1, 3, 5, \ldots$

These unwanted spurious signals must be identified and modelled correctly in order to accurately represent the response of a practical mixer for the emulator. Key parameters that classify the performance of a mixer are given next.

3.6.1 Conversion gain and loss

Conversion gain and loss are indications to which degree the frequency-shifted signal is amplified (in the case of an active device) or attenuated (for a passive device). This study will focus on passive mixers[4] and therefore conversion loss is applicable here since it is the double-balanced diode that will be looked at. Conversion loss can be defined as the input power of a mixer divided by its output power such that(Kroupa 1973):

$$L_{\mathrm{min}} = \frac{\mathrm{RF_{in}}}{\mathrm{IF_{out}}} \tag{3.21}$$

In a generalised ideal linear mixer where no IM products (i.e. for equation 3.20 n=1) and no losses occur, the theoretical IF signal level for a double-balanced diode mixer can be calculated as

$$V_{\mathrm{out}} = V_{\mathrm{RF}}\sin(\omega_{\mathrm{RF}}t)\frac{4}{\pi}\sin(\omega_{\mathrm{LO}}t) \tag{3.22}$$

[4]For a table comparing the merits of passive and active mixers, the reader is referred to p. 358 of (Rhode & Whitaker 2001).

$$= V_{\mathrm{RF}} \frac{4}{\pi} [\sin(\omega_{\mathrm{RF}} t) \sin(\omega_{\mathrm{LO}} t)] \tag{3.23}$$

$$= \frac{4V_{\mathrm{RF}}}{\pi} \cdot \frac{1}{2} \{\cos[(\omega_{\mathrm{LO}} - \omega_{\mathrm{RF}})t] - \cos[(\omega_{\mathrm{LO}} + \omega_{\mathrm{RF}})t]\} \tag{3.24}$$

The resulting IF voltage is thus

$$V_{\mathrm{IF}} = \frac{4V_{\mathrm{RF}}}{\pi} \cdot \frac{1}{2} = \frac{2V_{\mathrm{RF}}}{\pi} \tag{3.25}$$

From Eq. (3.21) the ideal conversion loss (sometimes called single sideband conversion loss and expressed in dBs) can be written as

$$L_{\mathrm{min}} = 20 \log \frac{V_{\mathrm{RF}}}{V_{\mathrm{IF}}} = 20 \log \frac{\pi}{2} = 3.92\mathrm{dB} \tag{3.26}$$

In practical mixers, however, insertion loss from various loss components (e.g. transformers, diode resistance losses) are accounted for when a given value of conversion loss is specified for non-ideal mixers (Couch II 1995).

3.6.2 Noise figure

Noise contributed by a practical mixer will always be added to the frequency-shifted signal. Ideally, the noise figure of a mixer should be equal to the result of equation 3.26 to yield of value of 3.92 dB. In practice, however, this figure of 3.92 dB normally higher which in-turn results in an increased noise figure (NF). One aspect of mixer noise, that's interesting to note, is the noise figure's dependency on the LO drive level as shown in figure 3.29 (Rhode & Whitaker 2001).

3.6.3 Conversion compression point

The 1dB compression point, P_{1dB}, for a mixer is an indication of the point at which the mixer output(IF) amplitude has fallen 1dB below the expect value. This compression occurs when the mixers amplitude response becomes nonlinear when a certain input level is exceeded. For a double-balanced mixer, the P_{1dB} point is typically 3dB below the LO power (Rhode & Whitaker 2001).

3.6.4 Harmonic intermodulation products

The spurious products harmonically related to the input signals f_{RF} and f_{LO} are called the harmonic intermodulation products (HIP) (Rhode & Whitaker 2001). IM distortion is the most severe problem associated with mixers, since they can interfere with the desired signal if they lie in the IF passband.

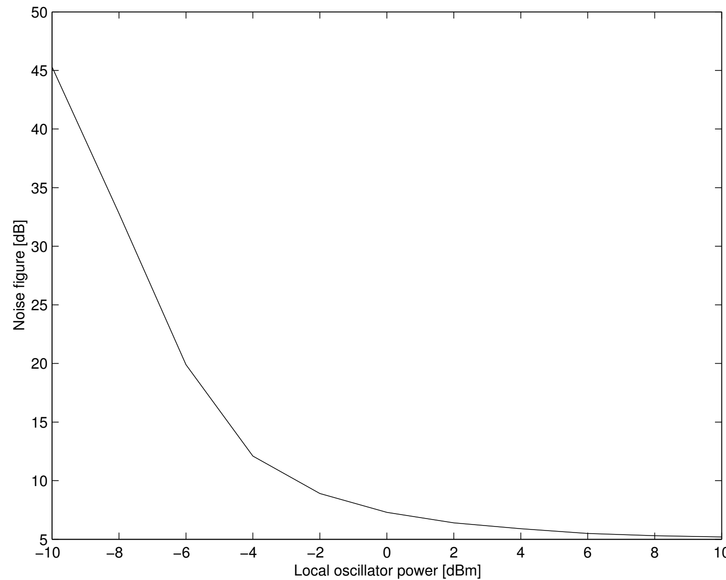


Figure 3.29: *Noise Figure versus LO power for a generic diode double-balanced mixer (adapted from (Rhode & Whitaker 2001))*

Table 3.2: *Typical performance figures for a passive mixer (f = 900MHz)(Rhode & Whitaker 2001)*

Parameter	Typical value
Conversion gain	−10 dB
IP_3 (input)	+15 dBm
LO power	+10 dBm
P_{1dB}(input)	+3 dBm

The IM distortion products appear due to the way mixer input signals interact with each other, and is defined according to equation 3.17. The associated IM distortion levels are specified with an (input or output) n_{th} order intercept point. The third and the fifth order intercept points (i.e. IP_3 and IP_5 respectively) are commonly used to characterise these non-ideal effects (Rhode & Whitaker 2001).

Typical values for a passive mixer that take into account conversion gain, IP_3, LO power and P_{1dB} are shown in table 3.2 (Rhode & Whitaker 2001).

3.6.5 Interport isolation

The interport isolation (also known as port-to-port isolation), R_{ip}, is a measure of the amount of leakage between any two given ports of a mixer. The strongest signal in a mixer output is normally the LO signal — this can be attributed to the high LO drive

level as well as the poor R_{ip} between the LO and IF ports (Hjorth & Hvittfedt 2002).

3.6.6 Quadrature mixers

Mixers are conventionally used as single devices and are often cascaded to form n mixing stages if needed. In some applications there is a requirement to translate two frequencies by the same frequency shift, simultaneously. A device used to achieve this is called a quadrature mixer.

A quadrature mixer uses two mixing stages to frequency translate an I (in-phase) and Q (quadrature-phase) signal to the desired output frequency. A functional block diagram of a quadrature mixer (within the dashed enclosure) is shown in figure 3.30. Figure 3.30 also shows a DAC and a LPF on the input of each channel to the quadrature mixer. This would be a typical setup for when the quadrature mixer is employed to up-translate the input signals to a desired frequency, specified by the LO frequency. The I and Q signals, in this case, are typically generated from a digital system. The DACs are used to bring the digital signal over to the analogue domain, after which, the LPFs are used to filter out the DACs zero-order-hold harmonics. It will be shown later that these two components, the DAC and LPF, play are major role in ensuring the spectral purity of the frequency-translated signal.

The I and Q baseband input signals are ideally of equal amplitude and the Q input channel ideally lags the I input channel by precisely 90°. The quadrature mixer then generates the product of I(t) and Q(t) with $W_{\mathrm{I}(t)}$ and $W_{\mathrm{Q}(t)}$ respectively. Typically, $W_{\mathrm{I}(t)}$ and $W_{\mathrm{Q}(t)}$ are sinusoidal waveforms in which $W_{\mathrm{Q}(t)}$ has an ideal phase lag of 90° with respect to $W_{\mathrm{I}(t)}$. Subsequent to the mixing stages, the baseband signals are translated to the carrier frequency. The last stage subtracts the Q channel output from the I channel output to produce, $y(t)$, the desired RF output.

Quadrature mixers can be used for alias-free frequency translation — that is, in an ideal quadrature mixer, only the single sideband of the translated single is produced. A typical application of quadrature mixing is in complex-valued modulation and demodulation schemes. For a software radio that employs quadrature mixing, it is important to study the effects that this type of mixer will have on the signal.

If the input is taken as a cosine waveform on the I channel input as $\cos\omega_s t$ and with an ideal 90° phase lag and the input on the Q channel is a sinusoidal waveform, $\sin\omega_s t$, then the following equation (interpreted directly from figure 3.30) confirms the upmixing process in figure 3.30 (van Rooyen 2000):

$$\cos\omega_c t \cdot \cos\omega_s t - \sin\omega_c t \cdot \sin\omega_s t = \cos(\omega_c + \omega_s)t \tag{3.27}$$

Equation (3.27) represents the ideal output of a quadrature mixer, which in this case is an upmixed component that lies at the sum of the carrier frequency and the baseband

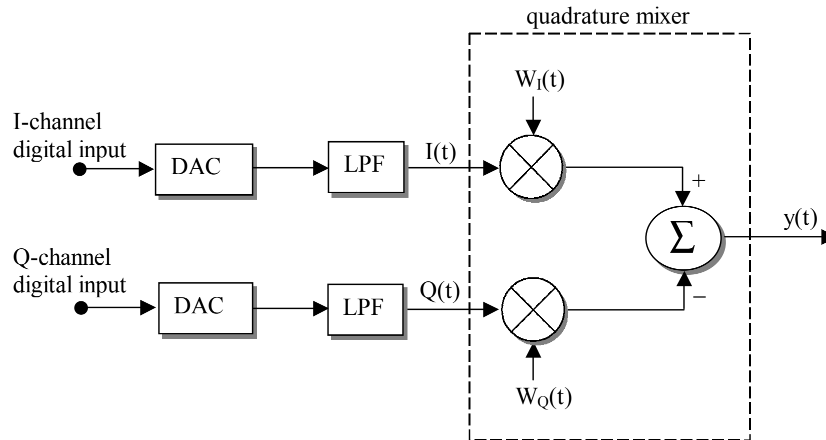


Figure 3.30: *Illustration of a quadrature mixer block diagram, showing the positions of the DAC and LPF for each channel*

signal frequency. It will be shown later that this setup (that of generating only a single frequency using a cosine and sinusoidal input) will be used in chapter 5 as a benchmark signal to compare outputs of a real system and an emulated system.

Of course, in a real system, a quadrature mixer is a practical device that inherently exhibits non-idealities of its own. In a typical quadrature mixer, the following non-ideal effects result in distortion of the output signal (van Rooyen 2000):

- I and Q signal amplitude mismatch
- DC offset on I and Q inputs
- Local oscillator leakthrough
- I and Q phase error

These effects can be compensated for in practice (van Rooyen 2000), but in order to obtain a practical model that generates these non-ideal effects, these quadrature inaccuracies must be taken into account. For this thesis, a quadrature mixer will be modelled that will represent these non-ideal effects. We will take a brief look at what causes these inaccuracies as well as the result of these quadrature imperfections in the frequency domain.[5] The basic quadrature impairment overview presented here will be adequate to use in order to obtain a good interpretation of the outcomes in the hardware test setup in chapter 5.

[5]For an in-depth analysis on the effects of quadrature mixing inaccuracies (as well as digital compensation techniques) the reader is referred to Van Rooyen (2000) (van Rooyen 2000).

Amplitude Imbalance

The incoming I and Q signals to a quadrature mixer should ideally be perfectly matched in amplitude — this is not always the case and in a practical system, some amplitude deviation effects could be present. The cause of amplitude imbalance for I and Q signals can normally be attributed to mismatched DACs, poorly matched LPF amplitude responses in the passband or, in certain cases, imperfect quadrature mixing stages where the amplitude of $W_I(t)$ and $W_Q(t)$ are not equally matched (van Rooyen 2000). The I and Q amplitude deviations can be represented as

$$I(t) = (A + \beta)\cos\phi(t) \tag{3.28}$$

$$Q(t) = A\sin\phi(t) \tag{3.29}$$

respectively and regardless of where the amplitude indifferences occur within the system, the RF output can always be written as (van Rooyen 2000):

$$y(t) = (A + \beta)\cos\phi(t) \cdot \cos\omega_c t - A\sin\phi(t) \cdot \sin\omega_c t \tag{3.30}$$

The actual error signal can be observed when eq. (3.30) is re-written as

$$y(t) = \overbrace{A\cos\phi(t) \cdot \cos\omega_c t - A\sin\phi(t) \cdot \sin\omega_c t}^{\text{wanted RF upmixed signal}} + \overbrace{\beta\cos\phi(t) \cdot \cos\omega_c t}^{\text{error signal}} \tag{3.31}$$

Expanding the error signal yields

$$\frac{\beta}{2}\cos[\omega_c t - \phi(t)] + \frac{\beta}{2}\cos[\omega_c t + \phi(t)] \tag{3.32}$$

which shows two components exist at the frequencies $\pm\omega_c$ with amplitudes of $\frac{\beta}{2}$ in the time domain. In the frequency domain, it can be shown that amplitude deviations therefore cause a spurious frequency to occur at $\omega_c - \dot{\phi}(t)$ and at $\omega_c + \dot{\phi}(t)$ with an amplitude of $\frac{\beta}{4}$ (van Rooyen 2000). This error signal is symmetrical about the carrier frequency and the component at $\omega_c + \dot{\phi}(t)$ is added (either constructively or destructively depending on the sign of β) to the desired component. The spurious free dynamic range, the ratio of the desired signal to that of the largest spurious component, can be calculated using (van Rooyen 2000)

$$\text{SFDR}_\beta = 20\log\left(\frac{2A + \beta}{|\beta|}\right) \text{ dB} \tag{3.33}$$

DC offset and LO leakthrough

For an ideal I and Q signal, no DC offset exists, but because real world problems can come into play, an I and Q channel DC offset could be present. This DC offset could be the consequence of DAC non-idealities (van Rooyen 2000). In the practical mixing stages of a quadrature mixer, it is quite possible that LO leakthrough can occur through imperfections in this mixing stage. Finite interport isolation could also contribute to LO leakthrough. The mathematical illustration and consequence of DC offset can be illustrated as was done for amplitude deviations. However, in order to prevent side-tracking into too much mathematical derivation theory, the result of DC offset is ultimately what is required. An important point to note is that DC offset and LO leakthrough produce the same result (van Rooyen 2000) — a spurious signal of constant amplitude added at the carrier frequency. If the amount of DC offset is represented by ρ and the amplitude of the signal is represented by A, then the SFDR can be calculated as (van Rooyen 2000)

$$\text{SFDR}_\rho = 20\log\left(\frac{A}{\rho}\right) \tag{3.34}$$

The magnitude of the DC artifact can be shown to be $\frac{1}{2}\rho$ (van Rooyen 2000).

Phase angle error

Quadrature signals are expected to have a perfect 90° ($\frac{\pi}{2}$) phase shift between each other. The same goes for the I and Q carrier frequencies. Phase error can, however, occur in practical systems. One of the possible sources of such phase error could be unmatched phase responses in the passband of the LPFs used after the DACs (van Rooyen 2000). An interesting finding is that the spectral result of the output (with amplitude A) with phase angle error (α radians) is similar to the error produced by amplitude deviations — the only difference being the phase of error signal (van Rooyen 2000). A phase angle error results in a spurious component symmetrical around the carrier frequency with magnitude $\frac{A\cdot\alpha}{4}$ for which the SFDR can be calculated as (van Rooyen 2000):

$$\text{SFDR}_\alpha = 20\log\left(\frac{\sqrt{\alpha^2+4}}{|\alpha|}\right) \tag{3.35}$$

The individual quadrature impairments have been briefly touched on and the resulting consequences in the frequency domain were highlighted. In a practical system, the combined effects of these quadrature inaccuracies can occur. This is an important point since quadrature impairments, individually, are relatively easy to calculate. When the quadrature impairments are combined, however, it becomes difficult to mathematically predict their effects on the modulated/demodulated signals. In this regard, the emulation of these effects is a valuable tool for analysis. Therefore, chapter 5 will demonstrate these

effects by analysing the impairments individually in the test system, in order to gain an understanding of the error contributed by each impairment.

Modelling consideration will be given next for both standard and quadrature mixers. Considering that the theory on standard mixers was presented first, the order will begin with standard mixer modelling considerations, followed by quadrature mixer modelling.

3.6.7 Mixer modelling considerations

General considerations

The mixer theory presented in section 3.6 showed that practical mixers generate a host of distortions. These unwanted effects include conversion gain errors, the addition of noise as well the generation of spurious IM frequencies. To obtain the spectral components of an ideal mixer requires just the multiplication of the RF and LO signal. This would be a straightforward implementation. What is needed, however, is a method for predicting the IM terms and its suppression and spectral position with respect to the given input signal frequency components (RF and LO) and their respective power levels. The remaining inaccuracies such as noise will be introduced by using the AWGN model illustrated in section 3.3.1. Conversion gain errors can be represented simply by using an ideal amplifier module with a specified attenuation factor. The generation of the mixer IM terms, however, calls for some further investigation. It must be stated upfront that a standard mixer will not be employed in the final emulated test system. A method for obtaining the necessary mixer modelling data and translating it to something useful to model a standard mixer is, however, proposed and is presented next.

Essentially, what is required in order to achieve this is a way of predicating the mixer IM components described by equation 3.17. Studies have shown that a practical and effective way of representing these IM products in a mixer model is by utilising what is known as an intermodulation table (IMT) (Faria & Svensen 1995). The IM products for mixer modelling considerations will only represent that of single-tone IM products (equation 3.17) since multi-tone intermodulation is beyond the scope of the thesis. IMTs represent the mixer IM output components. Every mixer has its own unique IMT.

A segment of an IMT is shown in figure 3.31 for illustration. The IMT segment contains some of the more predominant IM entries from the complete IMT. The remainder of the entries are relatively insignificant. The IM values in the table were generated with a signal level of −10 dBm and a LO level of 7 dBm (Faria & Svensen 1995). IMTs can be obtained from manufacturing product charts, (Hjorth & Hvittfedt 2002) but for best results it is recommended that the IMT files be constructed from mixer output spectral measurements (Faria & Svensen 1995).

The IMT in figure 3.31 represents the mixer IM product levels based on given local

n · LO →

m · SIGNAL ↓	0	1	2	3	4
0	99	39	42	46	58
1	25	0	39	11	50
2	68	67	76	67	80
3	63	58	65	60	65
4	96	80	96	80	95

Signal level = -10 dBm
LO level = 7 dBm

Table entries specified in dB below the desired signal

Figure 3.31: *Illustration of a typical intermodulation table (values adapted from (Faria & Svensen 1995))*

oscillator and input signal frequencies and powers. The resultant output signals are then a direct mapping of each input signal with respect to each LO signal. The IMT can therefore be interpreted as follows (Faria & Svensen 1995):

- Each entry is the amplitude relative to the desired frequency translated mixer output. Amplitude levels are specified in dBc which is relative to the mixer output power at the fundamental frequency.
- The harmonic number of the LO signal is represented by the horizontal row (n) of integers.
- The harmonic number of the input signal is represented by the vertical column (m) of integers.
- The item at position $m = n = 1$, which in this case is zero, corresponds to the fundamental signal, which could be either the sum or difference frequency.

For example, if the entry at column $n = 2$ and row $m = 3$, is taken (with an input signal of -10 dBm and a LO signal at 7 dBm), the following interpretations can be made. Investigation of figure 3.31 shows that there will be (from equation 3.17) an IM product at the frequency point $2 \cdot f_{\mathrm{LO}} \pm 3 \cdot f_{\mathrm{RF}}$ with a power level of 67 dB *below* the sum or difference signal frequency. In other words, this IM component is suppressed by 67 dB with respect to the fundamental frequency.

It is important to note that the IMT file only applies for a specific power level for both the LO signal and the input signal. Interpolation can, however, be performed to determine the new IMT elements for different input power signal levels (Faria & Svensen 1995).

What is required now is a way to translate this information from the IMT into something useful that can be modelled to represent the mixer IM components. An interesting

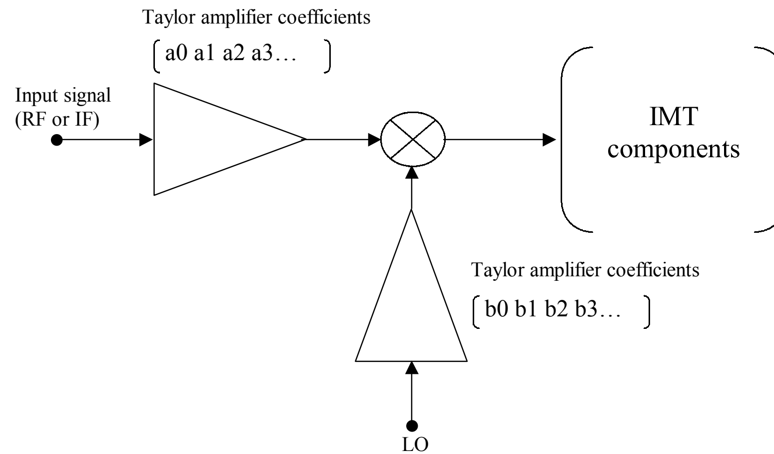


Figure 3.32: *A mixer described by a perfectly singular IMT matrix can be modelled as only the ideal product of two Taylor amplifiers*

observation is made when the entries in figure 3.31 are converted from dBs to absolute power levels. Close inspection of the IMT in this form reveals that the rows tend to be multiples of each other. The same trend can be observed for the columns of the IMT. This indicates that there is large degree of linear dependency between the rows and columns of the IMT - stated otherwise, the IMT matrix is close to singular.

For the IMT matrix in figure 3.31 to be completely singular, the entire range of rows and columns must have an exact linear relationship. An IMT matrix that is completely singular can always be written as the product of two vectors. For the purpose of mixer modelling, this concept makes it possible to model a practical mixer as the product of two non-ideal (Taylor) amplifiers as illustrated in figure 3.32. Of course, the series of harmonics described by the two arrays need to be converted to equivalent Taylor series coefficients for this type of mixer model to be realised.

In practical cases, however, IMT matrices are rarely perfectly singular and a technique is needed to address this type of condition. A linear algebraic method known as singular value decomposition (SVD) can be used to best approximate a non-singular matrix into the product of two vectors. SVD separates a $N \times N$ matrix into N vector products so that the sum of these vector products result in exactly the original product. If the matrix is close to singular then some of these vector pairs will dominate and the rest of the vector pairs will only make subtle differences to the original matrix. In the case where the matrix is extremely close to being singular, only the product of the first vector pair could be required to approximate the original matrix.

SVD divides a matrix, B, with N rows and M columns so that

$$B_{N \cdot M} = U_{N \cdot N} \cdot S_{N \cdot M} \cdot V^T_{M \cdot M} \tag{3.36}$$

where S (with the same dimensions as B) is a matrix that contains eigenvalues along the diagonal (arranged in descending order) and zero elsewhere. A completely singular matrix has just a single eigenvalue (at position $S_{(1,1)}$ in the matrix) with the rest of the entries equal to zero. For matrices that are non-singular, the magnitude of the eigenvalues in the diagonal gives an indication of how significant the corresponding vector pair is. If the first eigenvalue is an order larger in magnitude than the rest in the diagonal, then the matrix can be approximated well by multiplying the first column of U by the dominant eigenvalue, $S_{(1,1)}$. The result of this product must be further multiplied by the transpose of the first column of V in order to yield the approximated original matrix. If the first eigenvalue is not dominant, however, then sums of the vector pairs should be taken in order to better approximate the original matrix.

In order to further interpret this information for mixer modelling, the values in the vectors of the U and V matrices of the decomposed IMT can be seen as the output harmonics of the amplifier which still needs to be converted to equivalent Taylor series coefficients. Note that it is the convolution of these amplifier harmonics that produces the intermodulation components. The S matrix eigenvalues gives an indication of the signification of the columns of U and V. For the case where the first eigenvalue is orders of magnitude larger than the remaining eigenvalues, the mixer model can be approximated using two vector pairs converted to equivalent Taylor series coefficients (recall figure 3.32). The accuracy of the IMT mixer model can be improved by linking remaining vector pairs as needed as illustrated in figure 3.33. The addition of further vectors pairs improves the accuracy at the expense of computational speed.

The actual process of converting the harmonics to equivalent Taylor series coefficients requires finding the relationship between the vector harmonics of the decomposed IMT and its corresponding Taylor coefficients. Recall that this process was illustrated for a general third-order Taylor series (which corresponds to a four element IMT vector) in section 3.13 on amplifier modelling. The example illustrated only a $4th$-order harmonic series, but typical IMT vector pairs could contain lengths much greater than this (typically 15 harmonics per series (Faria & Svensen 1995)).

The obtained Taylor coefficients can now be directly substituted into the Taylor amplifier series was shown in figure 3.32. Depending on the respective weighting of the eigenvalues, additional vector pairs should be added if an increase in accuracy is required using the technique described by figure 3.33.

In order to verify the operation of the mixer model using an IMT described by the product of Taylor amplifiers, an experiment was performed. The input signal to the mixer model used was a single-tone signal at 100 Hz with a power level (specified by the IMT) of -10 dBm (Faria & Svensen 1995). The LO signal was set at 500 Hz, also with power specified by the IMT as 7 dBm. A sampling frequency of 3kHz was used. For

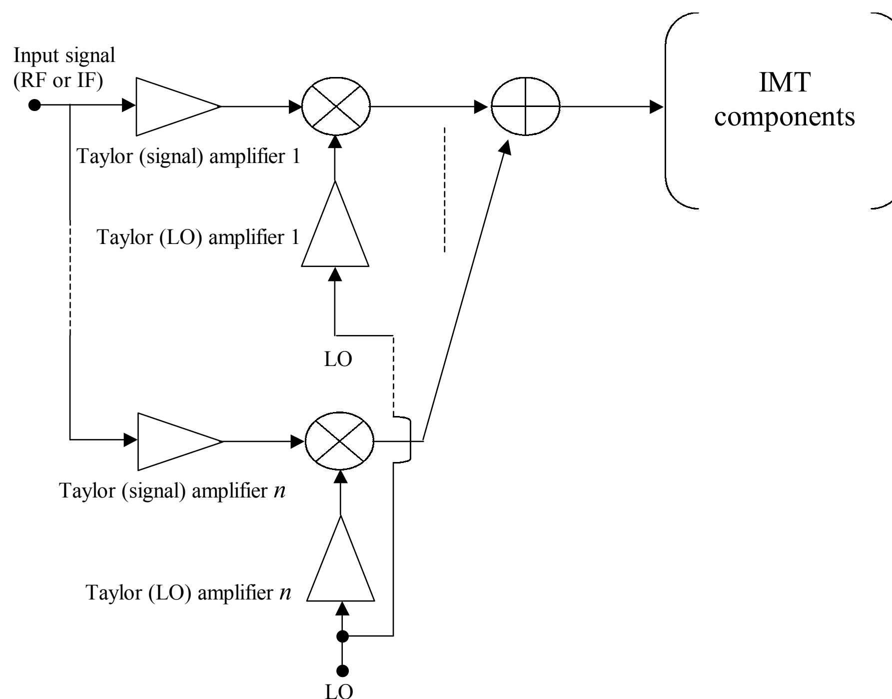


Figure 3.33: *Cascaded mixer model to improve the accuracy of IM component representation*

the mixer model, however, only a section of the IMT from figure 3.31 was taken and the IMT reduced to a four by four matrix as shown in figure 3.35. This was done in order to simplify the operation of calculating the relationships between higher-order harmonics and their equivalent Taylor coefficients. This simplification, however, does not interfere with final mixer model and only means that fewer IM products are represented. Recall from figure 3.33 that the mixer output level precision could be increased (in the case where the IMT matrix is not singular) by summing more vector pairs modelled as the product of Taylor amplifiers. At this point it is important to highlight and distinguish the two types of approximations presented for the mixer modelling considerations:

1. Firstly, the number of IM mixer harmonics generated by the mixer model is proportional to the size of the IMT used. Therefore, for a reduced IMT there will be less Taylor terms and therefore fewer IM products will be generated.

2. The output level resolution of the IM harmonics is directly dependent on the number cascaded Taylor amplifier pairs used (provided that the IMT matrix is non-singular).

The spectral output for the mixer model (described by the IMT in figure 3.35) approximated as the product of a single vector pair (recall from figure 3.32) is shown in figure 3.34.

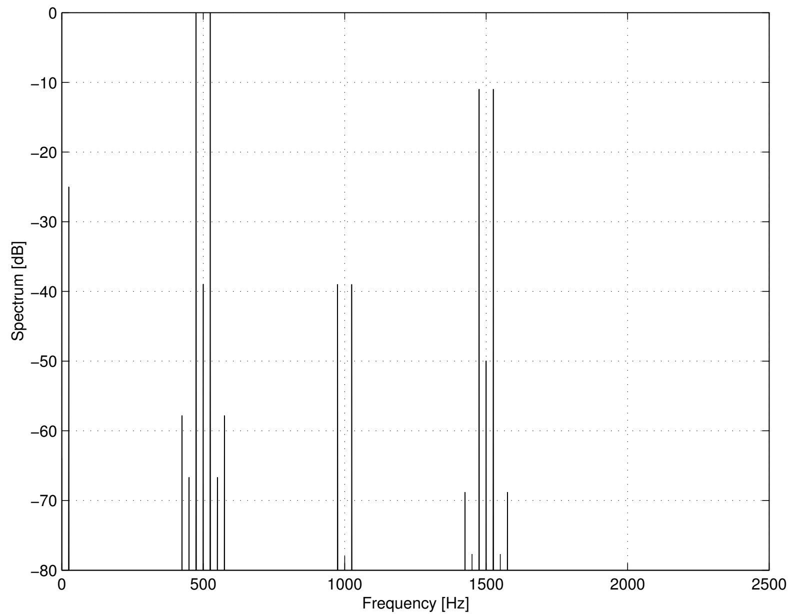


Figure 3.34: *Output of the mixer model using a single Taylor amplifier pair*

From the IMT in figure 3.35 we should expect the fundamental signal to be the largest component in the frequency plot, which is precisely what we observe. Taking a random entry, the IMT in figure 3.35 also indicates that a component 39 dB below the fundamental signal at a frequency of 500 Hz $(1 \cdot F_{LO})$ should be present. This is clearly the case looking at figure 3.34 and was measured at exactly 38.9 dB below the fundamental frequency. This high degree of accuracy can be attributed to the dominance of the first eigenvalue in the S matrix of the decomposed IMT.

Therefore, upon adding a second Taylor pair, there should not be a vast improvement in the spectral output as shown by figure 3.36. The 500 Hz component is now suppressed at exactly 39 dB below the fundamental signal, as described by the IMT. Once again, this slight improvement in value can be attributed to the approximation of using two Taylor amplifier pairs. For all practical reasons, such a slight improvement does not readily justify the use of an additional Taylor amplifier series pair and in this case, was not really required in order to model the IM product levels of the IMT.

Although a method for modelling a single mixer has been proposed, it is the quadrature mixing process that will be used in the sample system to be verified with hardware (chapter 5). For this reason, a basic quadrature mixer model has been developed that takes in an I and Q signal and causes some of the non-ideal quadrature mixing effects to be generated.

m · SIGNAL \ n · LO	0	1	2	3
0	99	39	42	46
1	25	0	39	11
2	68	67	76	67
3	63	58	65	60

Signal level = -10 dBm
LO level = 7 dBm

Table entries specified in dB below the desired signal

Figure 3.35: *The IMT used to verify the operation of the mixer model*

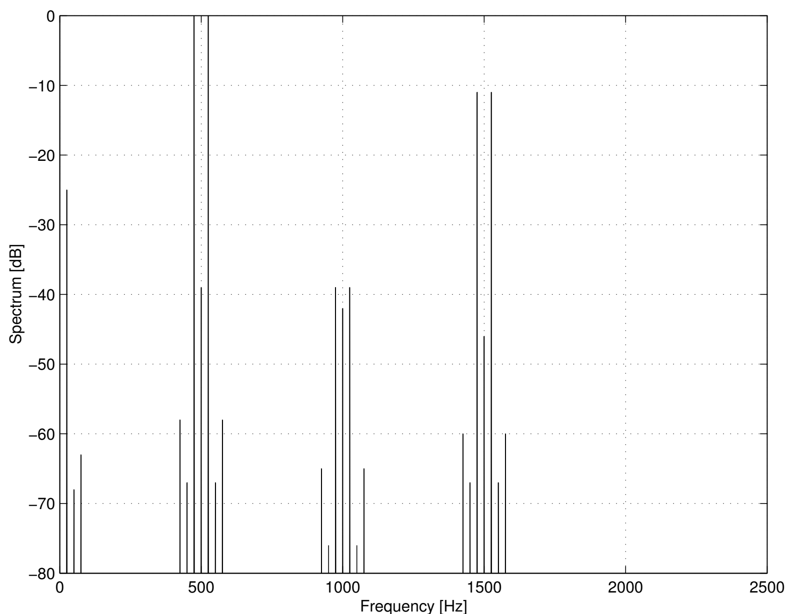


Figure 3.36: *Spectral output of the mixer model using the sum of two Taylor amplifier pairs*

3.6.8 Quadrature mixer modelling considerations

General considerations

The algorithm used to model the behavior of the quadrature mixer is fairly straightforward. The model is essentially a direct interpretation of figure 3.30. The core operation of the quadrature mixer prototype developed in MATLAB can be shown in a singe line of code as (van Rooyen 2000):

```
RF = I.*cos(2*pi*Fc*t) - Q.*sin(2*pi*Fc*t) + ...
     LTAmp.*cos(2*pi*Fc*t + LTPhase*pi/180);
```

where `I` and `Q` are the in-phase and quadrature input signals respectively. `Fc` represents the carrier frequency and was set to 300 kHz for illustrative purposes. A sampling period of 1 μS was used throughout the system. The third expression describes the quadrature error signal in the form of LO leakthrough. The LO leakthrough amplitude is specified as a percentage where, for example, a `LTAmp` of 0.01 translates to 1% carrier leakthrough. The LO phase error is specified in degrees (and converted internally to radians). Amplitude deviations are specified directly by adjusting the input signal level on one of the channels.

Quadrature mixer model validation

The quadrature mixer model was tested using a single-tone input with a frequency of 10 kHz. This meant a cosine waveform and sinusoidal waveform, each with a frequency of 10 kHz, was used on the I and Q channels respectively. A SFDR simulation algorithm was used in order to determine the SFDR of the various spectral plots (van Rooyen 2000).

For reference purposes, the output for an ideal quadrature mixer (no quadrature impairments) with a LO frequency of 300 kHz is shown in figure 3.37(b) with the spectrum of the 10 kHz input shown in figure 3.37(a). The overall gain for the quadrature mixer model was set at unity.

Carrier leakthrough and DC offset: It was stated in section 3.6.6 that carrier leakthrough and DC offset both cause a carrier frequency component to appear at the LO frequency. Figure 3.38 shows the output spectrum with a 10% DC offset added to the I channel of the ideal quadrature mixer model. From equation 3.34, the expected spurious component should lie at 20 dB below the carrier frequency — this was exactly the result measured by the simulation algorithm (van Rooyen 2000) .

Amplitude imbalance: In order to simulate the effects of amplitude deviation, a 10% amplitude difference was imparted on the I input signal (with respect to the Q input signal). The theoretical studies of amplitude deviation in section 3.6.6 indicated that

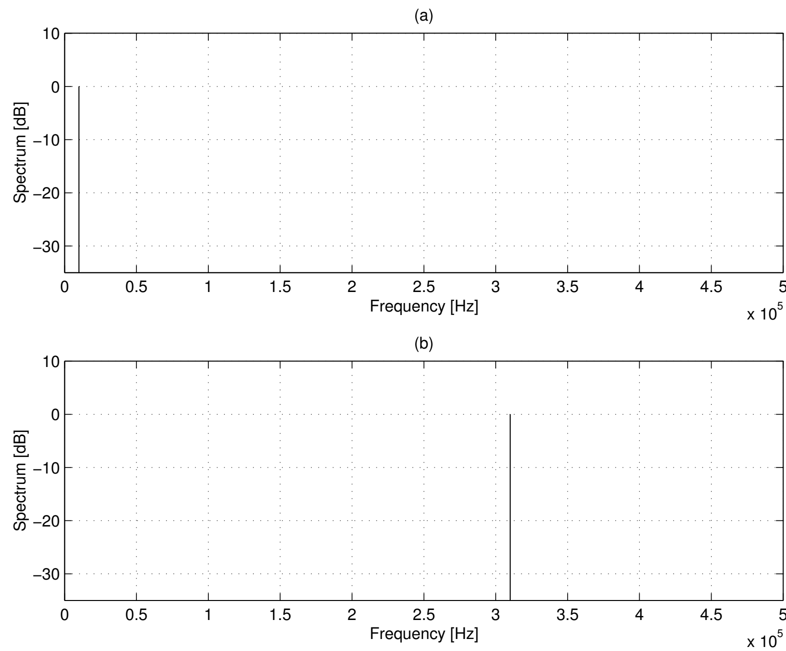


Figure 3.37: *Ideal quadrature mixer output for a 10 kHz input signal (a) upmixed with a carrier frequency of 300 kHz (b)*

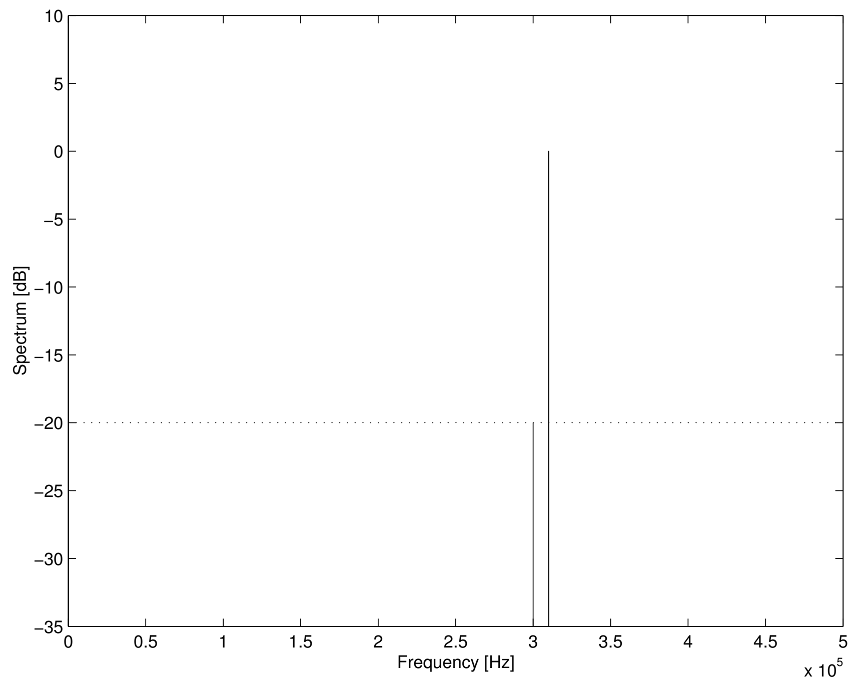


Figure 3.38: *Spurious component 20 dB below the desired signal as a result of 10% DC offset*

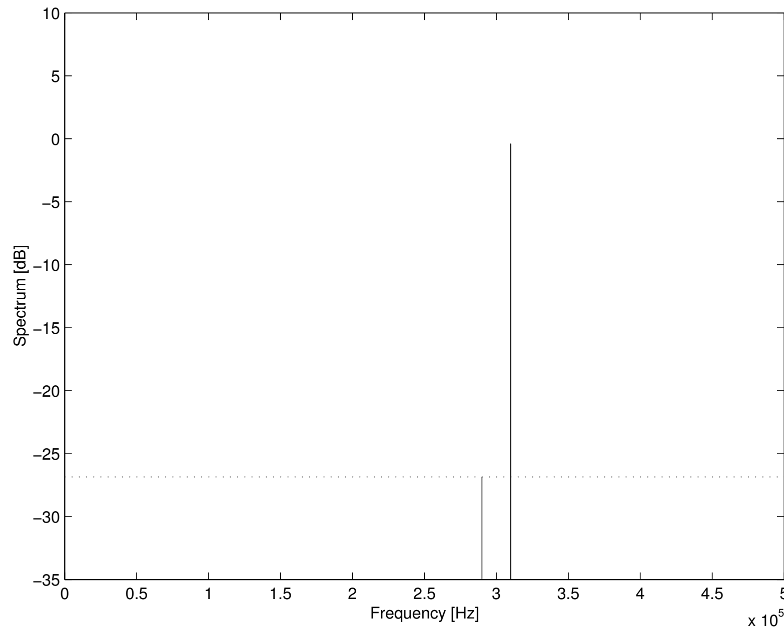


Figure 3.39: *Spectral output showing the spurious signal at 26.8 dB from the desired signal as a result of a 10% amplitude imbalance*

we should observe a spurious component at ±10 kHz around the carrier frequency, as shown in figure 3.39. Of course, the positive 10 kHz is added to the desired signal at 310 kHz. For a 10% amplitude deviation, the predicted SFDR from equation 3.33 yields a result of 27 dB, which is precisely the level measured by the simulation algorithm (van Rooyen 2000).

Phase angle error: Section 3.6.6 stated that the expected spectral response from a phase angle error is similar to that of amplitude deviation — a spurious component symmetrical around the LO frequency with a different phase in the error signal. Figure 3.40 illustrates the spectral output for a 3° phase error added to the Q channel input signal. The level measured by the SFDR simulation algorithm was 31.6 dB, which was precisely the result predicted by equation 3.35.

The effects exhibited by the various quadrature impairments were illustrated in order to get a better understanding of outputs that can be expected in the hardware emulation test setup in Chapter 5. In order to convert the quadrature mixer prototype model in something compatible in C++, certain modification were taken. The details of these modifications will be given in chapter 4.

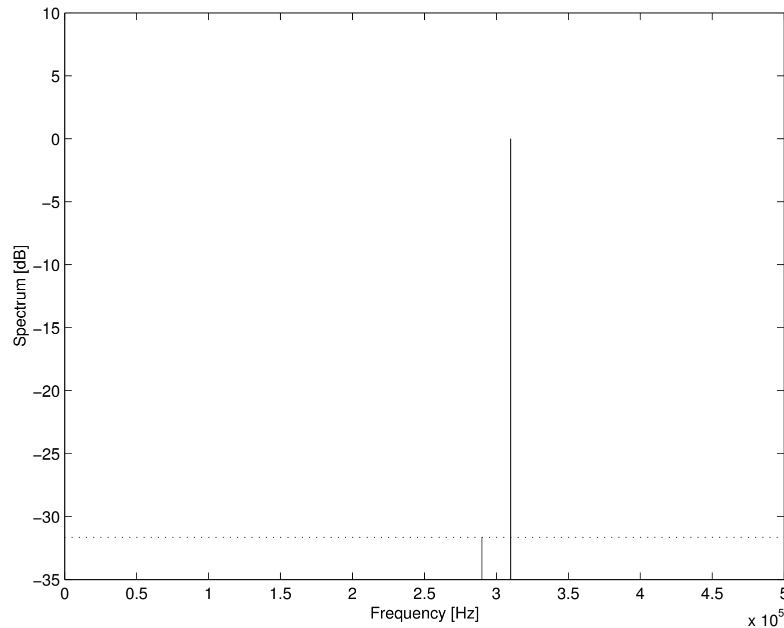


Figure 3.40: *Output of the quadrature model showing the effects of* 3° *phase error on the Q channel*

3.7 Filters

3.7.1 Filter classification

A electronic filter can be seen as a device that modifies the frequency spectrum of a signal (Chen 1995). There are various ways of classifying a filter. Frequency selectivity is one of the more commonly used classification methods where the class of filter (e.g. lowpass, highpass, bandpass, bandstop) describes the frequency range that is manipulated. A further way of classifying these filters is by their frequency selectivity characteristics with respect to the filters' amplitude and phase responses (Rhode & Whitaker 2001). Some of the more common filter responses are Butterworth, Chebyshev and elliptic responses.

Filters can also be classed as either analogue or digital. The RF front-end of SDR such as those illustrated in figure 2.3 of chapter 2 employ mainly analogue filters and this must should be taken into account when modelling these type of filters. Such analogue filters can be characterised by its linear transfer function, $H(s)$, which can generally be expressed as (Thede 1996):

$$H(s) = \frac{K\left(s^m + a_{m-1}s^{m-1} + a_{m-2}s^{m-2} + \cdots + a_1 s + a_0\right)}{\left(s^n + b_{n-1}s^{n-1} + b_{n-2}s^{n-2} + \cdots + b_1 s + b_0\right)} \tag{3.37}$$

$H(s)$ is the transfer function used to describe a filter in the frequency domain. A filter can also be described and analysed in the time domain by its impulse response, h(t). The

Laplace transform of $h(t)$ can be used to determine the system transfer function, $H(s)$. For the remainder of the thesis, filters will be analysed using their frequency domain description for both sampled and non-sampled systems.

Eq. (3.37) shows that the general s-domain transfer function of a filter can be described by the ratio of two polynomials. The order of the numerator and denominator is specified by m and n respectively. K is the overall gain constant. The rate at which the filter roll-off occurs is determined by the order, specified by the value of n.

If the polynomials of equation 3.37 are factored into first-order factors, then the poles (denominator) and zeros (numerator) of the transfer function can be obtained. These poles and zeros completely describe the characteristics of a filter and also give important information on the stability of the filter (Rhode & Whitaker 2001). The polynomials of equation 3.37 can further be factored into different orders and the importance of this will be shown later when the considerations for filter modelling are taken into account.

The complete coverage of filter theory has a rich history, and its complete study is beyond the scope of this thesis.

3.7.2 Filter modelling considerations

Previous sections on modelling components such as amplifiers and mixers showed that there were a number of component non-idealities to consider. Filters, however, behave pretty much according to the theoretical predications. Of course, when real components are used such as those in the RF front-end, one would expect slight discrepancies in the calculated response.[6] One of the reasons is due to the addition of thermal noise by passive components such as resistors — thermal noise will be accounted for with the AWGN model described in section 3.3.1 of chapter 3.

Analogue filters are defined in the s-domain. Equation 3.37 was used to describe a s-domain transfer function for an analogue filter — such a filter could occur in the RF front-end of a SDR. The emulation of the RF front-end components will, however, be done on a digital platform therefore requiring a digital filter implementation.

Although filter systems can directly be described and designed in the digital domain, what is basically required is a digital equivalent of the s-domain representation (Kuc 1988). The z-transform makes this possible. The z-transform is a mathematical tool used to describe discrete-time signals and systems such as digital filters (Kuc 1988). The s and the

[6]Sensitivity analysis (Thede 1996) can be performed in order to determine how various component tolerance levels affect the overall performance of a system. For a filter, these components could include resistors and capacitors that, due to age and temperature drifts, result in a slight change of value. Sensitivity analysis, however, does not fit the scope of this thesis and the reader is referred to the work of Thede (Thede 1996) for further reading in this regard.

z plane do not correspond directly with each other, and required techniques are available to enable this transformation process. One way of mapping from the s-domain to the z-domain is by using a technique known as the bilinear transform (Rhode & Whitaker 2001). The transformation used by the bilinear transform is given as (Rhode & Whitaker 2001):

$$s = \frac{2(1 - z^{-1})}{T(1 + z^{-1})} \tag{3.38}$$

where T is the sampling period of the digital equivalent. Therefore, any analogue filter can be implemented by its z-domain equivalent. The primary concern of filter modelling for the hardware emulator is to develop a way to implement digital filters. The next additional step would be to implement a technique to convert s-domain transfer functions to the z-domain (for example, by using the bilinear transform). This second step is not critical for the operation of the emulator, but it would be a useful upgrade and therefor does not necessarily form part of this thesis. For the purpose of converting from the s to the z-domain, however, MATLAB's built-in bilinear transformation function, `bilinear`, could meanwhile be employed.

One way of describing a digital filter is by using its difference equation which in general is written as (Thede 1996)

$$y(n) = \sum_{k=0}^{M} a_k \cdot x(n-k) - \sum_{k=1}^{N} b_k \cdot y(n-k) \tag{3.39}$$

where the coefficients a_k and b_k completely describe the system. The transfer function, $H(z)$, of a digital filter can be obtained by using the z-transform to transform the difference equation 3.39, which yields the form (Thede 1996)

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} a_k \cdot z^{-k}}{1 - \sum_{k=1}^{N} b_k \cdot z^{-k}} \tag{3.40}$$

where a_k and b_k are the system function coefficients. The z^{-k} components of equation 3.40 represent delays of k sample periods in the time domain and the importance of these delays in digital filter implementation will be described in the following section.

Digital filter implementation

Digital filters may be implemented in a number of ways obtained directly from either their difference equation or z-transform. One such implementation is called a direct form I structure and is shown in figure 3.41 (Rhode & Whitaker 2001). Figure 3.41 also illustrates the problem with the direct form I structure in that it is not very memory efficient, since it requires two sets of delays — one for the input and one for the output sequence. This problem is magnified when large filter orders are required.

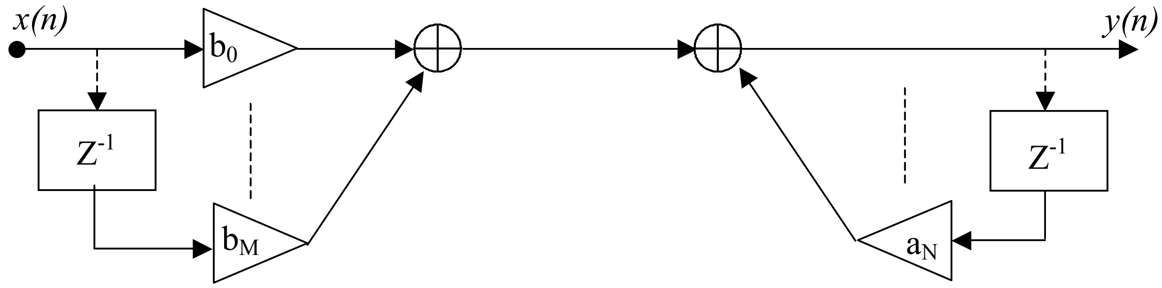


Figure 3.41: *Direct form I filter implementation*

A modified structure that uses a minimal number of delays, called the direct form II structure (Kuc 1988), will be used to implement digital filters in the hardware emulator. A modification of equation 3.40 can be shown to yield the following (Thede 1996)

$$Y(z) = W(z) \cdot \sum_{k=0}^{M} a_k \cdot z^{-k} \tag{3.41}$$

where

$$W(z) = X(z) + W(z) \cdot \sum_{k=1}^{N} b_k \cdot z^{-k} \tag{3.42}$$

In the time domain, equation 3.7.2 and equation 3.42 can be written as (Thede 1996)

$$y(n) = \sum_{k=0}^{M} a_k \cdot w(n-k) \tag{3.43}$$

and

$$w(n) = x(n) + \sum_{k=1}^{M} b_k \cdot w(n-k) \tag{3.44}$$

From equation 3.43 and equation 3.44 we can see that only a single delay element array, w(n), is required. One of the shortcomings of implementing such a design directly is that a new structure is required for different filter orders (Kuc 1988). One way of getting around this problem is by cascading second-order (quadratic) sections, achieved by factorising the transfer function, $H(z)$, into second-order polynomials represented as (Kuc 1988)

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(1 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2})}{(1 - b_1 \cdot z^{-1} + b_2 \cdot z^{-2})} \tag{3.45}$$

An example of such a single quadratic structure is illustrated in figure 3.42 (Thede 1996). The form of the coefficients that will be used in the filter model is represented as illustrated in figure 3.42. The a_0 coefficient, the transfer function gain, will always be factored out and simply be seen as a gain constant that will get multiplied by the input signal sample.

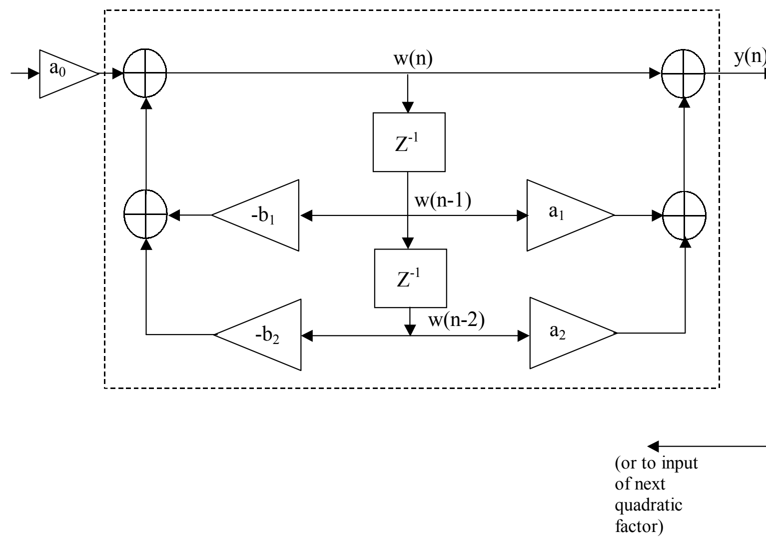


Figure 3.42: *Example of a second-order polynomial structure*

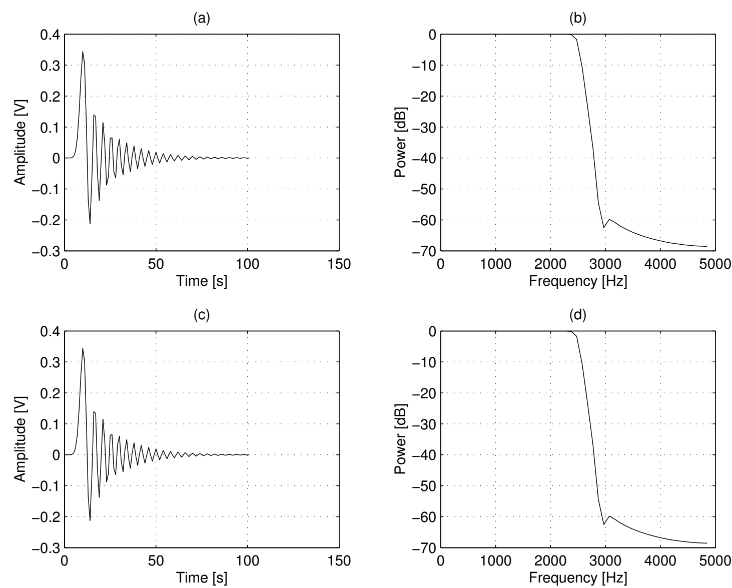


Figure 3.43: *Impulse response of the quadratic filter prototype in the time (a) and frequency (b) domain compared to the time (c) and frequency (d) impulse response of MATLAB's filter function*

The b coefficients in the system will be the negative of their value in the transfer function because of the manner in which the Direct Transpose II form is derived.

A function written in MATLAB, `poly2quads.m`, is used to convert the MATLAB generated filter coefficients (given as a complete n^{th} order polynomial) to second-order polynomials recognised by the quadratic filter model prototype. The syntax that describes the function is

```
[num2,den2,gain] = poly2quads(B,A);
```

The variables `B` and `A` represent the numerator and denominator input coefficient arrays respectively. The `poly2quads.m` function extracts the transfer function `gain` which is required by the emulator filter model. The MATLAB m-file used to create the filter prototype model has the following syntax:

```
y = quadratic_filter(a,b,g,x)
```

where `a` and `b` are the numerator and denominator quadratic coefficient groups respectively, returned by the `poly2quads` function. The variable `g` represents the transfer function gain coefficient and `x` is the input sample sequence.

Considering that the operation of `quadratic_filter` should, principally, work the same as MATLAB's built-in `filter` function, the built-in function was used as a benchmark to assess the performance of the `quadratic_filter` function. This is because they both perform the task of filtering according the specified input filter transfer function. In order to verify the operation of the `quadratic_filter`, the `poly2quads.m` function was used to group the following coefficients into second-order polynomials — the coefficients obtained from an order 20 butterworth LPF with a cutoff frequency of 2.5 kHz, designed using MATLAB's `butter` function. An unit impulse was supplied as an input to the filter models, in order to compare the impulse response of the quadratic filter against MATLAB's `filter` function. As can be expected, the impulse response and frequency response of both filter outputs should be identical, as shown in figure 3.43.

In summary, it was shown that filters in the emulator will be modelled using second-order polynomial structures. This technique was shown to be more memory efficient and the idea of using second order building blocks means that a quadrature filter implementation is highly flexible — a much desired property of the emulated components.

Recall from chapter 2 that the components identified in typical software-defined radios included the modelling of antennas and communications channels. Antenna (and channel) modelling is a vast subject on its own, and goes far beyond the scope of this thesis. Antennas can be characterised by a linear transfer function, and in its simplest form, can be viewed as a type of filter for which the ideal antenna would be an all-pass filter (Witkowsky & Van Rooyen 2002).

Chapter 4

Architecture design and model implementation

4.1 General design considerations

Chapter 3 was used to identify and expand on some of the theoretical background for the various models identified for SDR hardware front-ends. MATLAB models were used to test and verify the algorithms used to describe the components non-ideal behavior. The objective of chapter 4 is to focus on how the models are best placed together as software objects in a suitable architecture. In particular, chapter 4 discusses the design considerations surrounding the development of the emulator architecture using an object-orientated approach in C++. Additionally, the considerations taken for porting the core MATLAB model algorithms over to C++ is an aspect which is described in detail in chapter 4.

Although the modelling of components is important in the emulator system, the manner in which the models integrate collectively in a supporting framework is equally, if not more, vital. For the emulator, the architecture should be a supporting framework that allows the evolution of the system for improvements and also enables a high-degree of interchangeability for the modelled components. A well thought-out architectural design at the higher-level should foresee, and make provision for, as much expected future change as possible in order to prevent having to completely redesign the architecture. Also, conventional systems make use of physical interfaces to couple the various components in a real system. In contrast, the goal of the emulator architecture is to develop an efficient software linking method that enables the connection and optimal communication between the various components in the emulator.

Furthermore, in order to facilitate future development for the emulator, the architecture must be designed in such a way as to render the code easily understandable.

4.2 Architecture objectives

Selic (Selic et al. 1998) defines the properties of a good architecture as one that supports the evolution of a system and goes on to say that this is achieved by carefully examining the system requirements. With this in mind, the emulator architectural framework can be summarised with the aim for the following requirements:

- A high degree of interchangeability amongst components
- A high-level architectural design that is not concerned with too much lower-level model operational details.
- A design that easily facilitates the inclusion of new component models
- A design that allows the collection of processed data in order to enable effective post-processing analysis
- The emulated system should be designed to facilitate the emulators integration with a SDR software architecture
- The design should make use of a type of system topology that is easy to use, understand and modify

4.3 Architecture design

Keeping in mind the requirements listed above, the following sections aim to outline the approach to addressing the system architecture objectives. It is important to note that the emulator architecture design would ultimately be identical to that of a normal SDR architecture — it is only the functionality, which in the case of the emulator is a hardware specific design, that is different.

4.3.1 High-level design considerations

In order to meet the primary requirement that the components be highly interchangeable, a high level of abstraction is needed. In an object-oriented context, this high-level design requirement means that the majority of the functionality should be defined by the abstract base class (also known as the parent class). In essence, an abstract base class is required that defines both a general "component" as well as the way it interfaces with other "components".

This objective is aided by the fact that the SDR hardware components, at a higher level, all operate on the same basic principle. This principle is that, in general, all the

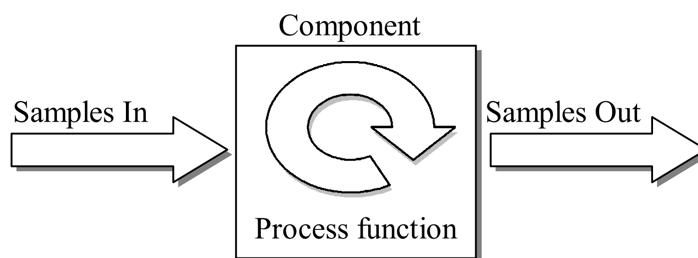


Figure 4.1: *Illustration of the commonality between SDR hardware components*

components receive, process, and output a stream of samples as illustrated in figure 4.1, regardless of the number of input and output ports.

In a certain sense, each component can be thought of as a "converter" of samples — in that the components all convert the samples in some unique way. The only difference is then the type of processing that gets executed in individual components. The exact process function of each component is defined by the type of component being modelled.

This means that the abstract base class (now called converter) need not know the exact details about the components individual processing function. The converter base class just understands that each component can receive and output a stream of samples. When the process function for each component (e.g. amplifier) is called, the individual component knows exactly how to operate on the incoming samples. The processing details, therefore, of the individual components operations need never be understood at a higher level. This high level of abstraction is what enables the ease of interchangeability for the various modelled components in the emulator. This is because, the emulator at a higher-level is indifferent to which component is being used. The emulator, therefore, does not cater for specific components. This simple but powerful shallow hierarchy design also allows the inclusion of improved model designs with relative ease.

Figure 4.2 illustrates this design and shows the declaration of the abstract converter base class. The generic functionality resides within the converter base class and all descendant classes. This concept is based on the "strategy" design pattern, in which a derived class implements the functional details of a generic base class within a shallow hierarchy (Gamma et al. 1995).

The function details shown in figure 4.2, called methods, of the generic converter base class that can be performed by derived classes (such as DAC and amplifier models) will now be explained.

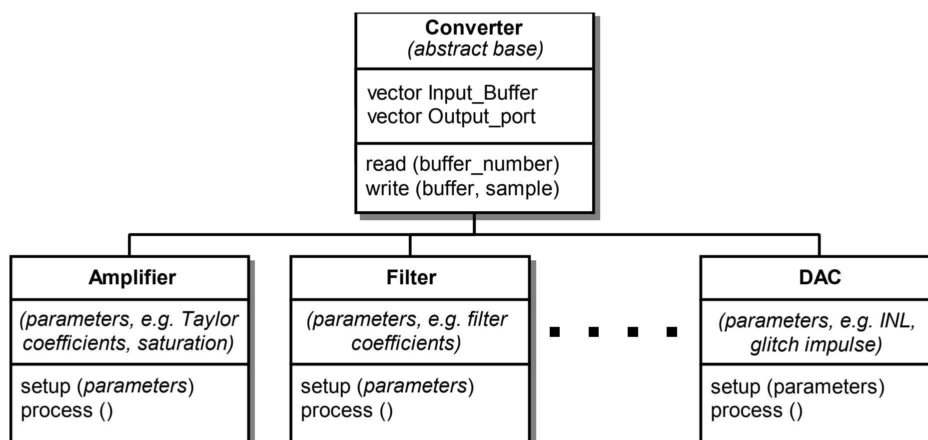


Figure 4.2: *Illustration of the strategy-based shallow hierarchy design employed for the emulator architecture*

Converter methods

One of the first elements highlighted was the commonality of receiving, processing and outputting samples. The functions used to achieve this in the architecture are illustrated as follows:

- A "read" method that acquires samples from the component's input buffer
- A virtual "process" function. A virtual function just ensures that the descendant class specifies what process gets performed
- A "write" method to dispatch samples to a input buffer of a component

These three methods form the nucleus of the emulator architecture. An important function shown in figure 4.2 is the setup method. The setup method is used by individual components to initialise the input parameters (e.g. Taylor coefficients for an amplifier) to a specified state. The components default values (specified in an objects constructor) are used in the case where the setup function is not called.

The buffering system in the emulator is implemented using a container class called a vector (Eckel 2000). A vector is a C++ class template, which implies that it can be used hold different types — in the case of the emulator, the vector will contain data samples specified by floating point values. A major advantage of vectors is that they automatically expand to hold incoming data - unlike arrays that require the size of the data array to be specified upfront.

It is also important to realise that modelled components can contain more than one input and output channel, such as a mixer which has two input (RF and LO) and a single output (IF) channel. Accommodating multiple input ports is simply matter of simulating

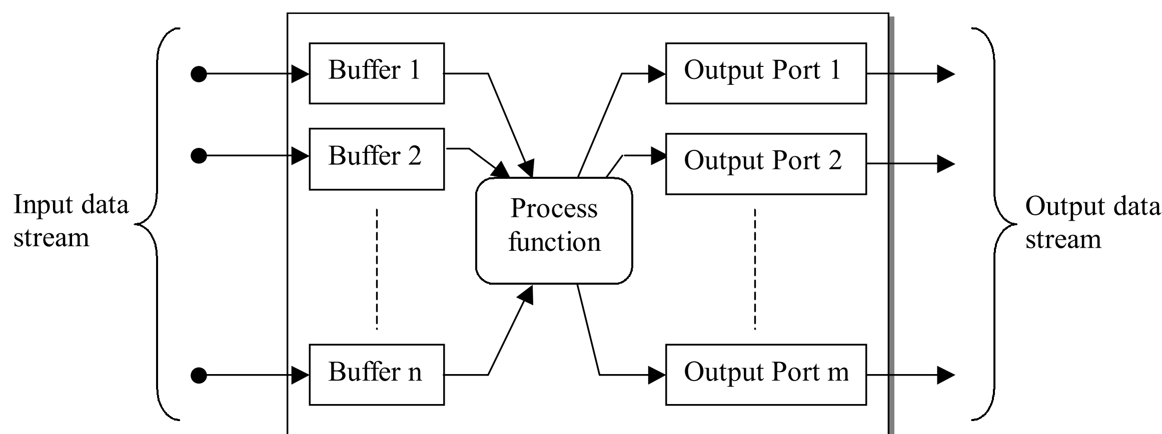


Figure 4.3: *A generic component illustrating the input and output interfaces*

a two-dimensional array - this is done by creating a vector of vectors. The different input vector buffers can be indexed just as one would index a multidimensional array, which makes working with vectors in this regard relatively simple. The advantage of using multidimensional vectors (as apposed to arrays) is that vectors automatically take care of allocating and deallocating memory. Vectors, therefore, take care of any possible memory leaks that might accidentally occur.

The output ports of a component is specified by a vector of structures, with one structure representing one output port. The output ports do not actually buffer the data but instead directs the data samples to the next input buffer of the desired destination component. The details of the output ports structure content will be explained later but for now figure 4.3 gives an overall idea of the basic layout of a modelled component's input and output facilities.

The remainder of the abstract base class functions are developed in support of the three primary converter methods. In summary, these functions just check to ensure that the input buffer is ready to process samples. There are some other converter methods that exist, but there importance will be shown later in the application design. The manner in which the samples are transported between various components depends on how well, topologically, the architecture is designed and the various consideration in this regard is illustrated next.

4.3.2 System topology

The manner in which the different modelled components transfer data samples amongst each other depends strongly on the type of topology used. Data in the emulator is streamed on a sample-by-sample basis (as apposed to a complete vector of samples) and a suitable topology must be selected in order to support this. A control-based topology

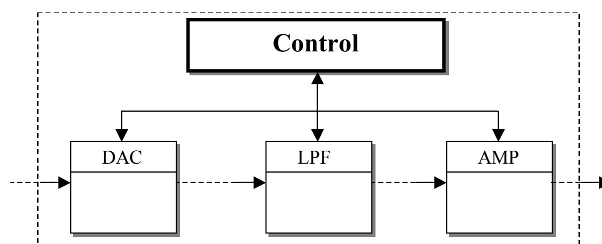


Figure 4.4: *Illustration of the emulator's control-based interface which is used to monitor the status of the buffers*

is employed in the emulator so that the emulated components are contained within a scheduling object as illustrated in figure 4.4. Here each component has its own input buffer and it is the responsibility of the control object to monitor the buffer's status. In an emulated system, the control object polls the various components in a certain order and the components in turn respond by indicating the status of their buffers.

Alternatively, it would have been possible to design the architecture on a peer-to-peer based topology. With such a topology, components are stringed together in a domino-like effect so that one object just points to the next. There is no need for input buffers since groups of samples are immediately processed and subsequently passed on to the following component. The obvious advantage of this type of topology is its simplicity since no buffers, and therefore no buffer control and monitoring, is required. The drawback, however, is strong pipeline flow of samples. Processing samples one batch at a time can result in irregularities in the process flow, especially in time-critical applications. Additionally, complex architectural configurations that include feedback networks, can be very computationally taxing on a peer-to-peer topology if large numbers of samples are processed.

Currently, the emulator is designed only for a single execution run, as opposed to being run continuously. With this in mind, when the control object determines that the samples in the system have all been processed, the control sends a message to indicate to the application program that the emulation is complete. The higher-level application can then decide what action to take once this signal is received.

Linking

The basic model interfaces have been discussed. These include the input buffers and output ports. The input buffers were examined earlier and were shown to use vector series to enable multiple inputs. The same principle is applied to the output ports, except that structures are used instead of buffers. The information in the structure tells the component the exact destination details in terms of where it is directing the outgoing

samples. The information specified in the output port structure consists of:

- A destination component
- The corresponding input buffer number of the destination component

These are the only two pieces of information required to effectively link components together. With the output structure designed in this way, each component knows exactly where the processed samples are being directed — this information is especially useful for application design, as will be shown later in section 4.6. The operational principle of the output port is fairly simple — once a data sample is processed using the output port information, it immediately gets placed in the input buffer of the destination component. Using only an input buffering system, as apposed to an additional output buffer, means greater overall design simplicity.

In order for the output port structure to direct samples to a destination input buffer means it must direct the samples to a new area of memory. In a programming context, pointer variables are normally used to hold the address of the specified area of memory been targeted (i.e. where the buffer resides). Conventional (static) pointers are fairly simple to implement.

The drawback of using static pointers, in the context of the emulator architecture, is that when the object being pointed to is destroyed, the pointer will be pointing to nothing (an unknown area of memory), and this can result in segmentation faults. It is therefore necessary to explicitly delete pointers when objects (i.e. modelled components) are removed.

An elegant solution to this problem is to use a modified version of a pointer called a reference counted pointer[1] (RCPtr) (du Preez 2001). A RCPtr is a C++ class which is similar in use to a static pointer, but is functionally much safer. It is important to realise that a RCPtr is, therefore, an object and not a static pointer. An example of an output port structure utilising a RCPtr to point to the next component, is shown in figure 4.5.

A RCPtr basically works by counting the number of pointer references to an object. The RCPtr has a built-in safety net, in that when the RCPtr detects that all pointer references to an object no longer exist, it destroys the object that was being pointed to. In the context of the emulator, this is useful because if components are removed in an application, there is no chance of accidentally writing to an unknown space in memory since the component will only be removed from memory once all references to the object no longer exist. This is because, for example, if we have the following setup: A DAC, filter and amplifier model all linked in series. Remember that, at a higher-level, the object processing samples does not need to know anything about the linked object except that

[1]The RCPtr is part of the patrecII library, used by the DSP group of the University of Stellenbosch.

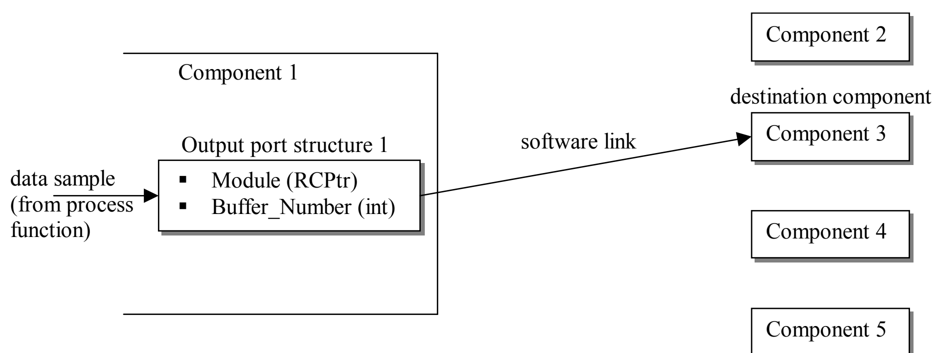


Figure 4.5: *The RCPtr in the output port structure is used to hold information about destination of the linked component*

it's a descendant of the converter base class. Suddenly the user decides to remove the amplifier in the application program. What the RCPtr does now is recognise that there is still something pointing to the amplifier (the filter) and does not therefore actually destroy the amplifier object in memory. In other words, the filter is still pointing to something, which prevents the program from accidentally writing to unknown areas of memory when the program is executed, which in turn can cause the program to crash. It only destroys the object once all the references to the object (the amplifier in this case) no longer exist.

In summary, some of the advantages of RCPtr's over standard pointers are (du Preez 2001):

- No memory leaks, since memory being pointed to does not need to be explicitly deleted. The RCPtr class automatically applies a type of garbage collection.
- Newly created RCPtr are automatically initialised to point to nothing.

RCPtr's are extensively used throughout the emulator's design in order to facilitate the linking of components. A link method was developed for the converter base class that just allows the destination component and destination input buffer number to be specified, and is given by the following syntax:

```
int link(int output_port_number, RCPtr<Converter> destination_module
 ,int destination_portnumber);
```

The `link` method therefore requires the following parameters in order to successfully set up a link with a specified component:

- The output port number of the source component.
- The RCPtr to the destination component.

- The input buffer number of the destination component.

Recall that components can have multiple inputs and outputs, and it is for this reason that the exact input (buffer) and output port number must be explicitly specified for a specific destination component.

4.4 Sample rate considerations

In order to avoid aliasing in sampled signal system, Nyquist criterion states that the sampling frequency must at least be twice the frequency of the highest highest sampled frequency component (Couch II 1995). Nyquist criterion is therefore an important consideration to bear in mind in order to accurately represent the signals in the sampled data domain. It is also important to note that signal bandwidths can vary between different components in SDR front-ends. therefore, in order to observe momentary (analogue) effects such as glitch impulse, a much higher sampling rate is required at the output of a DAC as apposed to its input. This is because glitch impulse could last for only fraction of the sampling period (Witkowsky & Van Rooyen 2002) on the output of a DAC and its important to be able to represent this in the emulator.

It should be assumed in general that the emulated component's input and output sampling rate can be different. The control-based topology used in the emulator allows samples to be transferred as needed which means that a change in components sampling rate can be accommodated for. The idea of upsampling a signal was touched on in section 3.4.4 of chapter 3. A function that works on the same principle to increase the sampling rate is implemented for the C++ models which has the following piece of syntax for the simple setup function:

```
int setup(int new_ratio)
```

The function basically repeats the incoming sample by the amount specified by `new_ratio`. Models such as the quadrature mixer, make use of a upsampling process in order to accurately generate the output signal, as will be illustrated in the following section.

4.5 Model transfer considerations

Recall that MATLAB was used to develop and verify the algorithms designed to model the hardware front-end components. In order to use the outcomes of the MATLAB developed prototypes, certain modifications to the models were made. These changes included modifications to allow the modelling algorithms to effectively operate in C++ since it is not possible to directly port the MATLAB code into C++. This section will give a description of the modification that needed to be performed.

One of the biggest considerations that needed to be taken into account was the manner in which samples are processed. The emulator's models should process the samples one at a time but the MATLAB prototypes simply processed a complete array of buffered samples. Programming loops were used in order for the C++ models to process samples one at a time.

In the emulated model's design in general, vectors were used in C++ instead of arrays because of the high degree of flexibility associated with vector. Something that was also very important to note was that array indexing in MATLAB starts at one, whereas C++ indexing begins at zero. Another consideration that needed to be taken into account was to find equivalent C++ functions for the MATLAB built-in function employed by some of the prototype models. The specific changes required for each model in order to successfully migrate the MATLAB algorithms into C++ will be given for the remainder of this section.

4.5.1 Data converter models

Quantisation model

The quantisation model was fairly simple to port over to C++ since the model effectively is described by a single line of code in C++ as:

```
double y = sign(x)*(ceil(fabs(x)*(q-1)/(2.0) - 1.0 + EPS)
           + 0.5) / (q-1) * 2.0;
```

The output, `y`, is specified as a double floating point precision variable. The `sign` function was the only function developed using a separate utility library. The `sign` function is principally similar to MATLAB's equivalent function in that they both just indicate the sign of the number via there return value. The remainder of the functions are all part of the standard C++ math library.

INL model

In order to implement the curve fitting algorithm, such as the cubic spline function used in the MATLAB prototype, an equivalent public-domain C version of the cubic spline routine, obtained from the Numerical Recipes website (Press et al. 1992), was used. The function will be used to generate the desired INL curve, such as a S-shaped or bow-shaped curve. The cubic spline function is made up of two functions, `spline` and `splint`. The `spline` function has the following syntax:

```
spline(xx, yy, n, 1.0, 1.0, yy2);
```

The spline function, which can be seen as an initialisation function, is used only once to process an entire tabulated function for the markers specified in the arrays `xx` and `yy`. The two entries of 1.0 each specify the derivatives (a straight line) at the very first and last point for the given curve — the `spline` function requires that the gradient of the end-points of the curve be explicitly given. Given this end-point information, the `spline` function is able to calculate the derivative of the remaining points as the gradient between the surrounding points. Following the calling of the `spline` function, the `splint` function is then used to obtain interpolated values for the function specified by the curve markers. Stated otherwise, a given input value will produced a distorted output value based on the shape of the INL curve specified by the markers in the `xx` and `yy` input arrays.

Glitch model

The only substantial change made between the MATLAB glitch prototype and the C++ model was the implementation of a different hashing function. Recall that the deterministic hashing function was used to generate a glitch height and width value (in the range of 0 to 1) based on three input parameters: the current quantisation level; the last quantisation level; and a seed value. The drawback of the simplistic hashing function used in the MATLAB prototype was that it produced unwanted patterns in the output transitions. Stated otherwise, the values generated by the hashing function did not seem random enough. For the C++ implementation, a more advanced hashing algorithm is used called MD4. The MD4 function is traditionally used as a security algorithm and traditionally finds its application as an encryption tool.[2] The MD4 library was used (with slight modifications in order to make the overall code size smaller) to implement the function in the DAC model. The original MD4 function works by taking a string as an input, and produces a 128-bit message digest. Because of its powerful algorithm, the MD4 function makes this process appear to be completely random, but the same input string always produces the same 128-bit digest — this principle is important since it will be used to adapt the MD4 into a suitable hashing function for the glitch model as follows: The last, current and seed values are placed in an input data array after which, the MD4 algorithm is called. The MD4 function then returns a 128-bit result (message digest) which is then masked to take the first byte as the height and the second byte is taken as the width value. A subsequent programming routine just conditions the output of the height and width values (using the modulus operator, '%') to fall in their respective ranges — the height value ranges from ± 0.5 and the width value ranges from 0 up to 1, as illustrated by the following piece of actual syntax used:

[2] Although the MD5 (MD4's upgrade), a more complex and secure algorithm, is available, its application as a hashing function would be superfluous for use in the DAC model.

```
*height = (float)(digest[0] % 100)/100 - 0.5;
*width = (float)(digest[1] % 100)/100;
```

Both the height and the width values are specified as pointers, and both outputs are type-casted to ensure floating point output values. The final hashing function, as used by the DAC glitch model, has the following syntax:

```
void makehash(unsigned int last, unsigned int current,
 unsigned int seed, float *height, float *width)
```

As explained earlier, the `makehash` function uses a last, current and seed value to generate output values for the height and width pointers. The height and width values are then used directly to specify the shape of the glitch.

4.5.2 Amplifier model

No major changes were required in order to accommodate the MATLAB saturation algorithm into C++, since its principle operation is based on simple techniques — using programming loops. The implementation of the Taylor amplifier's core processing algorithm also requires no special functions, and therefore the operations available in the standard C++ math library could be used. However, the Hilbert transform function used in the MATLAB models is slightly more challenging to implement in C++. The final emulated test system used in chapter 5 does not require the phase distortion information of an amplifier. The implementation of a C++ Hilbert transform, therefore was deemed superfluous for the purpose of this thesis and could be implemented as further development work. For now, the MATLAB prototype is sufficient to illustrate the operation of the Hilbert transform a quadrature Taylor amplifier model.

4.5.3 Mixer model

Although no actuall C++ mixer model was developed, the mixer modelling consideration in chapter 3 showed that mixers can simply modelled as the product of Taylor amplifiers — this makes the implementation of mixers a fairly straightforward operation, since existing (amplifier) models can be used as the primary building blocks. The mixer model should also be designed to accommodate various orders of Taylor amplifier pairs. For the moment, all the preconditioning work for the IMT decomposition is done in MATLAB. The resulting Taylor coefficients can be used in a C++ mixer model.

4.5.4 Quadrature mixer model

Recall that quadrature mixers use two equal carrier signals (a sinusoidal and cosine waveform) in order to perform complex frequency translation. In order to develop a quadrature

mixer model for the emulator, it is therefore important to understand the concept of generating sinusoidal signals in the discrete-time domain.

In the sampled data domain, the concept of an absolute frequency does not directly apply, such as those used to classify frequency in an analogue signal (e.g. 10 kHz). A discrete-time sinusoidal may be expressed as (Proakis & Manolakis 1996):

$$x(n) = A \cdot \cos(2\pi f n + \theta), \quad -\inf < n < \inf \tag{4.1}$$

where:

- n = integer variable
- A = signal amplitude
- f = frequency (now in cycles per second)
- θ = phase (in radians)

The expression $2\pi f$ can alternatively be written as ω, which is expressed in radians per sample. f, is expressed as a relative frequency and can lie in the range (Proakis & Manolakis 1996):

$$|f| \leq \frac{1}{2} \text{ or } |\omega| \leq \pi \tag{4.2}$$

The fundamental period, N, is $\frac{1}{f}$. Frequencies in this range ($|f| \leq \frac{1}{2}$ or $|\omega| \leq \pi$) are identical to those frequencies obtained with $|f| > \frac{1}{2}$ or $|\omega| > \pi$ and are regarded as alias frequencies (Proakis & Manolakis 1996). therefore, frequencies in the range specified by equation 4.2 are all unique, and the highest rate of oscillation would occur at $\omega = \pi$ (or $\omega = -\pi$) or $f = \frac{1}{2}$ (or $f = -\frac{1}{2}$). Figure 4.6 shows the results of four plots for different relative frequencies for the signal in equation 4.1. Note that an increase in the relative frequency results in a decrease in the period, and that no oscillation (DC) occurs when $\omega = 0$ or $f = 0$. In practice, quadrature mixers take a single input LO signal and performs a $\pm 45°$ phase shift to produces the sinusoidal and cosine carrier waveforms. The LO input of the model, however, indicates the input floating point value that specifies the relative frequency of the carrier waveform, as will be shown later. The actual quadrature mixer model is implemented with a separate sinusoidal and cosine signal that uses the principle of phase accumulation to produce the periodic carrier waveforms as shown by the C++ code syntax:

```
LOPhase = mod(LOPhase + 2*PI*f, 2*PI);
```

so that the I and Q channel carrier waveforms are generated with the syntax

```
I_carrier = cos(LOPhase);
Q_carrier = sin(LOPhase);
```

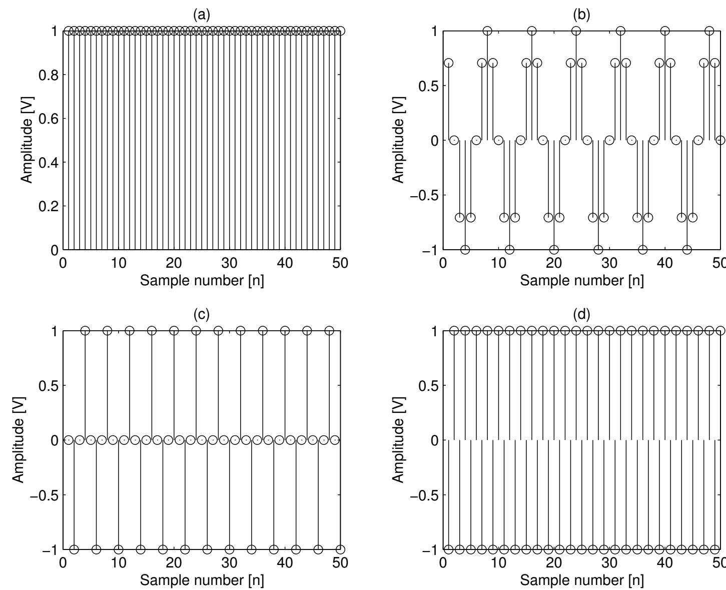


Figure 4.6: *Output for selected discrete-time frequency values: (a) $f = 0$; (b) $f = \frac{1}{8}$; (c) $f = \frac{1}{4}$; (d) $f = \frac{1}{2}$*

The initial value of `LOPhase` is zero, and is then increased by the amount of `2*pi*f` for every further increment. The variable `f` determines the carrier frequency of the discrete-time output signal. The `mod` (modulus) function ensures that the `LOPhase` value is wrapped around 2π intervals (i.e. a periodic waveform).

Recall in section 4.4 it was mentioned that the input and output sampling rates of components need not be the same. This is especially true in the quadrature mixer model since the frequency translated signal can be much higher (in the case of up-mixing) than the input signal. Therefore, in order to accurately represent this up-mixed signal, the output sampling rate must be at least twice that of the highest output frequency — in accordance with the Nyquist criteria. In order to adhere to the Nyquist criteria, the output sample rate should be adjustable in quadrature mixer model.

This implementation of oversampling can simply by achieved by repeating each input signal sample (I and Q) by an equal integer amount, N. The upsampling rate, N, essentially specifies the number of output samples generated for each input sample.

By the nature of the upsampling process using the method of sample repetition, the usual sample-and-hold harmonics are generated. This is an undesired effect for quadrature mixer sample-rate-conversion requirements. Therefore, a LPF is required in order to remove the sample-and-hold harmonics[3].

---

[3]A more advanced method of upsampling, such as interpolative upsampling, should be employed in order to address to problem of the zero-order-hold harmonics — for a text containing in-depth treatment

Considering that the upsample rate can change, means that the cut-off frequency of the LPF should change correspondingly. For a C++ implementation, this means having to recalculate the filter coefficients for different upsampling rates. Recall, however, that the task of calculating the filter beforehand coefficients in the emulator was performed using MATLAB. The task calculating high-order filter coefficients in C++ for the filter model is beyond the scope of this thesis. This is because the filter model also requires that the calculated filter transfer function to be grouped into second order polynomials — a task which could become complex in C++. therefore, for the purpose of this thesis, all upsampling and filtering requirements will be performed prior to the quadrature mixer model.

The relative frequency, f_{c}, for the quadrature mixer model is calculated relative to the output sampling rate as the input LO rate divided by the output sampling rate. I.e.

$$f_{\mathrm{c}} = \frac{\text{LO frequency}}{\text{Output sampling frequency}} \tag{4.3}$$

The result of equation 4.3 must fall in the range $0 \leq f_c \leq 0.5$ in order to generate a distinct output carrier waveform, where a f_c of 0.5 translates to a maximum carrier frequency that is half the output sampling rate. The best way to illustrate the setup of the quadrature mixer model is by example — the input conditions (i.e. input sampling rate) should be known beforehand. If, for example, the following setup exists:

- A 5 kHz single-tone input signal (i.e. a 5 kHz cosine and sinusoidal signal on I and Q channels respectively)
- An input sampling rate of 5 MHz.
- A desired LO frequency of 1 MHz

These are the only items needed to calculate the variable f_{c}. In order to accurately represent the frequency-translated signal of 1.005 MHz, the input sampling rate of 5 MHz will be sufficient. The LO frequency variable, f_{c} is calculated according to equation 4.3 and yields a value of $f_{\mathrm{c}} = 0.2$ for the relative frequency value.

4.5.5 Filter model

The core filter algorithm used in the MATLAB filter models can be transfered directly to the C++ models — only the indexing was changed to start from zero, and not one. It would be ideal to have a C++ wrapper that takes in the analogue filter's specifications and automatically generate the filter coefficients for the model, but currently all the

of sampling rate conversion and the various implementation techniques, the reader is referred to the textbook by Proakis (Proakis & Manolakis 1996).

pre-processing work gets performed in MATLAB. The conditioned output, which is the numerator and denominator coefficients and well as the transfer function gain gets used as input parameters to the C++ filter model.

4.6 A front-end application design

One rewarding aspect of a well-designed architectural system (in terms of its resources) is that it should easily facilitate the development of a front-end application. The extent to which the emulator's architectural framework supports the development of a simple front-end application program will be tested in the section. The main objective for the development of the front-end application program is to show how easily components can, from a user-level, be interchanged to form various hardware front-end configurations — this was one of the primary objectives set out in chapter 1. The operations, and essentially the user menu, therefore include the following options:

1. Add or remove a component
2. List current component selection
3. Link components
4. List component links
5. Run the emulation

The remainder of this section aims to show how these options were implemented in the C++ front-end application. The design and ideas presented are for a basic front-end system, but the intention is to illustrate how, given the design of the architecture, easy it is to develop a basic front-end application for the architectural framework.

4.6.1 Add and remove components

Components in the emulator are all declared as RCPtrs. This is done in order to enable the linking of components in the system, as will shown further in section 4.6.3. All components (e.g. amplifiers, filters, mixers) are contained within a single vector — this makes adding, removing and indexing components very straightforward. Therefore, all the powerful additional features associated with the vector class can also be used to work with the series of components in the vector. The vector methods "push_back" and "erase" are used to add and remove components in the system respectively. These operation are simply done by indexing to the correct component in the vector (as is done with array indexing).

An important aspect of erasing a component in the application, is that if it was linked to other components, the links will be broken too. Section 4.6.3 on linking will illustrate how this problem is elegantly dealt with.

4.6.2 List components

Considering that the user should be able to add and remove components at any time, the option to list the selected components is required. Some additional thought was required to implement this option. This is because the main application just has a vector of components to work with — this vector does not directly convey information about the exact type of object (e.g. an amplifier, filter, mixer). The main application only knows that each object in the vector is a decent of the converter class. Some type of identification 'tag' is therefore required in order to indicate the type of component in the vector. The approach taken for the emulator was to include an additional method in the converter class, given by the following syntax:

```
virtual const char* isA() const = 0;
```

The method works simply by returning the name (actually a pointer to a string of characters) of the descendant class (e.g. filter) to the main application. The `virtual` keyword means that the implementation of the method will be defined by the descendant classes and the syntax will always look exactly like this. The `virtual` keyword also implies that when the main application calls a converter's `isA()` method, the descendants `isA()` method is the one that actually gets executed. The first `const` keyword is used to ensure that when the `isA()` method is called, the contents (the character string) pointed to by the return value, cannot be changed by the function calling the `isA()` method. The `isA()` method now ensures that it is possible to identify the components in the component vector simply by indexing the correct component and calling its respective `isA()` method. The `isA()` method also now means that an enumerated list of components can be developed for the user in order to distinguish items in the case where more than one of the same type of component is used.

An additional factor that must be taken into account is that components can have multiple inputs and outputs, such as a quadrature mixer. In this case, it will be useful to have a method that counts the number of inputs and outputs of a component. To implement this option is a straight forward operation since all that is needed is the `size` vector method — this illustrates once again the benefits of implementing components within a vector.

If the user request to view all the selected components, both the components, and the numbers of input and output ports are now indicated.

4.6.3 Link components

The concepts related to how links for the components will be formed in the architecture (as well as the method) were illustrated in section 4.3.2. The main point highlighted in section 4.3.2 was that the models use an output port structure that holds information about the destination component and the corresponding input buffer number it is being linked to. A well-designed class should, therefore, provide member function by which the application program can adjust an object's properties (e.g. setting up links). The class should also provide a way for the application to query the current object properties — section 4.6.4 will highlight why this is important. With this in mind, there are basically two approaches to handling the linking components:

1. A top-down control approach where the application is responsible for keeping track of the component links in the system. Here the application also takes control of collecting output samples and sending them to there correct destinations.

2. The components, themselves, know where they are linked to, and the application need only request that the object makes or breaks its links. The approach means that samples are sent directly from one object to another object in the system.

The two points listed above warrant some further explanation. The drawback of the first point is that a large amount of functional responsibility is placed in the application program — this is not good for efficiency purposes. Therefore, for reasons of efficiency, option (2) is the approach used in the emulator since samples are now directly passed on between components in the system. The linking information (which component is link where) is contained within the objects themselves and the application need not keep track at all about the linkages between components.

The method given in section 4.3.2 is the only one used to efficiently link the various components within the emulated system. Once the components in the system have been been loaded, the user needs to supply four parameters:

1. the source component and its output port number.

2. the destination component and its input buffer number

The application is designed so that information about the exact input buffer or output port number is only required if more than one of either exists for a particular component.

Because of the simple manner in which components can be linked, it was possible to implement an option to automatically link components — here components with a single input and output are connected serially according to the way they are placed in the vector of components.

Lastly, the point made in section 4.6.1 about dealing with broken links will be addressed. Links are found both to and from components and as soon as a component is removed, the link from the source component still exists. The safety net built into a RCPtr ensures that, in memory, the object still exists but to the user it is removed. This is because if the emulation is run with this broken link, no segmentation faults (or even worse complete system crashes) can occur since the source object still writes to a destination object in memory. The object in memory will be destroyed only once the link to it is changed. However, this method of passing samples to an object that is no longer part of the emulated system is very memory-inefficient, because nothing gets done with the written samples. The samples continue to build up without being processed until the application runs out of memory. A simply and elegant approach used by the emulator is that when broken links are found, they (the RCPtrs) are just assigned to point to nothing (NULL). Recall, however, that a RCPtr is actually an object and not a pointer which makes it impossible to directly assign it to NULL. However, a new RCPtr is always initialised to NULL by its defaults constructor. therefore, when broken links are found, the system simply points (links) the components to a new RCPtr (i.e. NULL). Keeping in mind that a RCPtr is a object and not a pointer, a RCPtr method `objPtr` can be used to verify whether a RCPtr points to NULL.

4.6.4 List component links

It is important to provide the user with information on how the various components are linked in an emulated configuration, in order to keep track of the system's components interconnections. Therefore, some supporting methods were developed in the converter base class in order to facilitate this requirement to query the links in the system. The syntax of the primary method used to achieve this is given as:

```
int query_link(int output_port_number, RCPtr<Converter>* ...
 destination_module, int* destination_portnumber);
```

The `query_link` method works by taking the output port number of a given (source) component, and the method returns pointers to the linked components name and input buffer number. The application program has a function called `ListLinks` that, using the `query_link` method, obtains information about linked components. The `ListLinks` function works simply by going through the vector of components and uses the `query_link` method to retrieve the details of the component's links. The `isA()` method, mentioned earlier, is used in the `ListLinks` function to obtain the names of the components.

In order to display something useful to the user about linked components, the application checks the corresponding input and output ports. For example, if an amplifier and a filter are linked in a system, the message to the user will be displayed as:

```
Amplifier(1) Output Port[1] LINKED TO Filter(2) Input Port[1]
```

This information indicates to the user that the first component's output port on the list (amplifier) is linked to the input port of the next component on the list (filter). Notice that the indexing, as it appears to the user, starts from one — the actual indexing in the system starts at zero but the former is more user-friendly.

4.6.5 Run emulation

One would expect the implementation of a "run" function to be a fairly complicated piece of code. The advantage, however, of having well-designed descendant class member functions means that minimal complexity is required on the part of the application in order operate the process function for each component. It is, in fact, so simple, that all the application is required to do is call the component's `process()` function and the correct operations will happen for a component to process samples — this is exactly how the run function is implemented in the application. The following piece of syntax taken from the `run` function illustrates the simple way in which each component's process function is called:

```
do {
theFlag = 0;
for (int x = 0; x < Components.size(); x++) {
if(Components[x]->process()==1) {
theFlag = 1;
}
}

 } while(theFlag != 0);
```

The `run` function goes through the objects within the vector of components and calls the `process()` function for each component. The integer variable, `theFlag`, is used to verify the status of the samples in the system. Once the condition that no samples are being processed by any component is met, the flag status will be left at zero, and the program exits the `run` loop.

Recall that a control-based topology was selected for the emulator architecture for reasons of efficiency, since processing a complete batch of samples at a time could lead to processing bottlenecks in the system. The control-based topology, therefore, gives each component in the system a chance to process one sample at a time. The application does this by querying the objects input buffer to see whether there are samples available

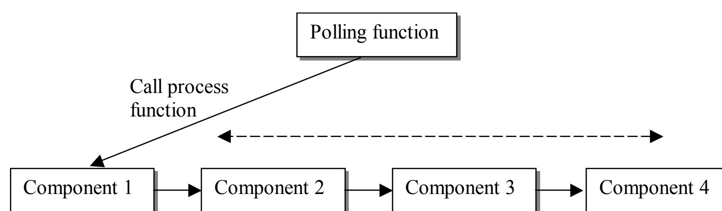


Figure 4.7: *Illustration of the polling of components in the emulator. The process function for each component is called if samples are available in its input buffer.*

for processing. If no samples exist in the input buffer, then the application moves on to the next component in the series. Recall that the number of samples in the input buffers of components can be completely different, since some components make use of upsampling which increases the number of output samples. The application essentially works by polling each component in the vector of components as illustrated by figure 4.7. Therefore, regardless of the order in which the components are linked in the system, the polling system always allows each component to process a single sample at a time. This process repeats itself along the list of components until all the components have no remaining samples in there respective input buffers — this is an indication that the emulation is complete. Once the emulation is complete, an additional method is used to re-initialise the input buffers of the components. This method is not completely necassary but is required in order to reset the buffer of the source component — for the emulator, this is a data source file and reseting it's buffer ensures that multiple emulation runs can be performed. The following section outlines the purposes of the source, as well as other supplementary, supporting modules.

4.7 Generating and monitoring samples

In order for an emulation to render itself useful, pertinent data about the performance of the emulated system must be made obtainable in order to have a complete understanding of the modelled systems performance. Fortunately, this is made very easy in a digital system since samples at key points can simply be written to a file for analysis purposes. This is exactly the approach taken in the hardware front-end emulator — a monitor module is used to probe key points in the emulated system. therefore, if six monitor probes are used, there will be six data files with information about the point in the system being probed. Essentially, a monitor module is just another component in the system that can be placed between components. The supplementary functions in the emulator can be listed as:

- Source component

- Monitor component
- Sink component

The monitor component was explained earlier but the remaining items deserve further explanation. Currently, the considerations needed for interfacing the hardware emulator to the software radio architecture have not been taken yet. therefore, in order to facilitate the operation of the hardware emulator as a stand-alone application, some form of source and destination component is needed to initiate the delivery and collection of samples respectively. Of course, future software developments would include writing the output samples of the emulator to the input of the software architecture of the SDR. As stated earlier, the source component is just a file that holds source data information — this might, for example, be samples of an audio waveform. Once samples in the system (from the source component) have been completely processed by all the components, the sink module is used as a means of collection. Here the completely processed samples are written to a destination file — in this way the sink module is similar to the monitor module except that the sink module does not have an output port to write samples to another component.

Once the samples are available in a file, a simple MATLAB post-processing function is used in order to evaluate the monitored waveforms in the time and frequency domain. The advantage of using MATLAB as a post-processing tool is that determining parameters such as the SFDR of signals is made fairly simple. MATLAB also has facilities to develop a graphical user interface (GUI), and this approach will certainly be considered for future versions of the post-processing software in order to render it more user-friendly. For now, however, a simple MATLAB test-based menu is used in order to analyse the output of the emulator's waveforms. The application, written in support of the emulator, gives the user the option to either write data to a file, or to read data from a file. These are the precise functions that will be used to analyse the data obtained from a sample emulated system, presented in the next chapter.

Chapter 5

Emulator evaluation — a sample system

In order to determine the accuracy of the hardware emulator, a comparative analysis of an emulated system will be performed against an equivalent physical hardware setup. One of the design goals is to select a hardware setup that inherently generates some of the non-idealities which the emulator models were designed to produce. This chapter describes the hardware and emulated setup in detail. The chapter then goes on to analyse the results generated from each setup and finally concludes by providing a comparative analysis for the results of the two configurations.

It is for this reason that the following setup was selected, and a description of it will be given next.

5.1 Hardware system setup

5.1.1 System overview

The choice of hardware setup was based on an existing SDR quadrature modulator, and therefore no actual physical construction was required. The hardware setup chosen also reflects a typical setup that could be employed by other parts of the SDR project. The hardware configuration is illustrated in figure 5.1.

The choice of setup was driven by a need to use components that would show some strong non-ideal effects. Therefore, the soundcard of a PC was used in order to represent the DACs and LPFs in the system, with the left and right channel outputs used for the I and Q signals respectively. The sound card employs a 16-bit DAC, but in order to emphasise quantisation noise, the data values were pre-quantised to 8 bits prior to being passed to the DACs, using the following syntax in MATLAB for both channels:

```
I = round([i(1:1e5)]'*128)/128;
```

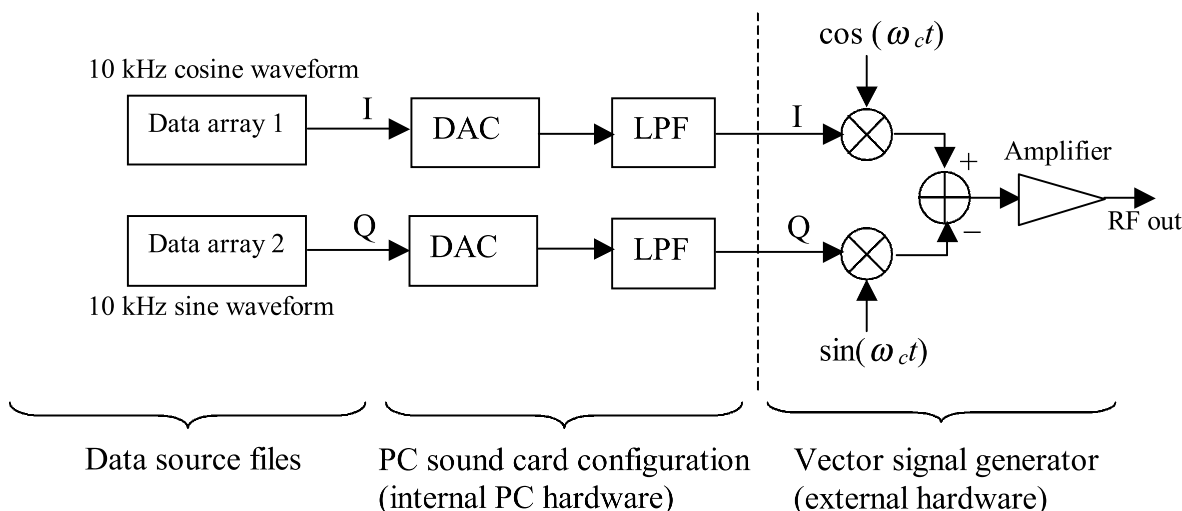


Figure 5.1: *Block diagram of the hardware system setup*

```
Q = round([q(1:1e5)]'*128)/128;
```

The code above indicates that the vector lengths for the I and Q inputs signals were altered — this was done in order to find an optimal output time window in order to observe and capture the results on the spectrum analyser.

The exact gain response of the LPF was not known, but later in the chapter it will be shown how this was measured. The input source file data values for the I and Q channels are synthesised cosine and sine wave samples respectively — 10 kHz waveforms sampled at a rate of 44.1 kHz (as is supported by the soundcard). The source files were generated in MATLAB and MATLAB's built-in `wavplay` function was used to write these samples directly to the soundcard output. The sine and cosine components are used to generate an up-mixed frequency using the quadrature mixer. Recall that equation 3.27 in chapter 3 was shown to mathematically illustrate this process.

The hardware used to perform the quadrature up-mixing was a Rhode and Schwarz (model SMIQ 04B) vector signal generator which had the capabilities to not only perform quadrature mixing, but also to deliberately add quadrature impairments (ROH 2003). A carrier frequency of 300 kHz was selected for the quadrature mixer, since this was found to be a relatively quiet part of the spectrum in terms of any noise sources present. All spectral measurements were made with the Hewlit Packard HP8562A spectrum analyser. In summary, the hardware was setup as follows:

- A 10 kHz synthesised cosine and sine source files with a sampling frequency of 44.1kHz
- The soundcard's on-board DACs (effectively 8 bits)

- The soundcard's on-board reconstruction LPF filters — the filter specifications will be measured in section 5.1.2
- A quadrature mixer with center frequency at 300 kHz and gain set at 0.5 dB

5.1.2 Measurements

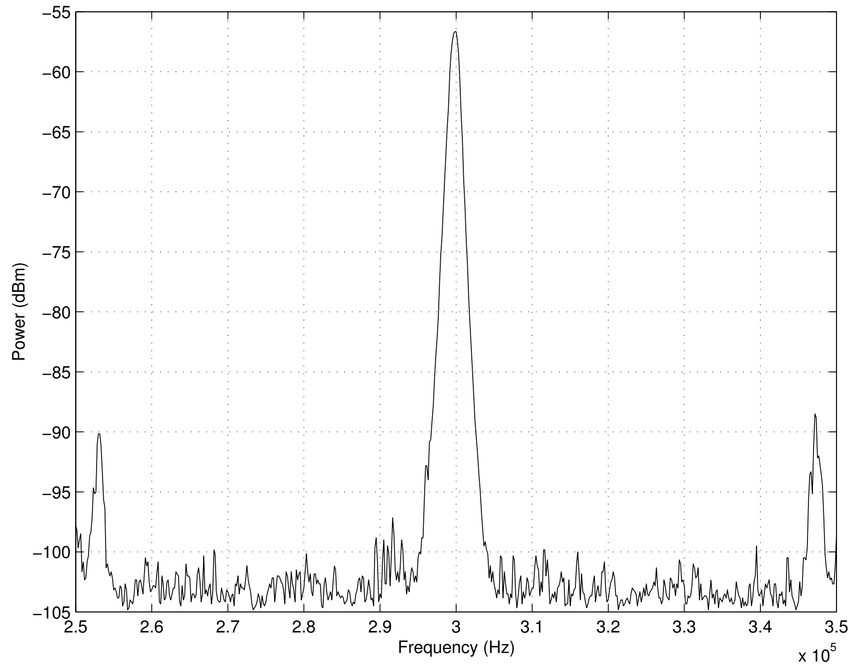


Figure 5.2: *Quadrature-mixed soundcard output with no input data, clearly indicating a carrier leakthrough component at −56.7 dBm generated n the soundcard*

Initial conditions

Figure 5.2 shows the spectral output of the system with no input data. Ideally, there should be no spectral components displayed at all. The soundcard is, however, a non-ideal device and its imperfections can clearly be seen in the form of a strong spur at the carrier frequency, measured at a level of −56.7 dBm. There is a high probability that this carrier leakthrough component is caused by a slight DC offset on the soundcard output. This can be expected, since a soundcard is normally used to drive speakers, and a slight DC offset usually goes unnoticed. It will be shown later, when the response of the low-pass filter is determined, that the soundcard does not allow a DC signal to be generated, but it does have a slight (unchangeable) DC offset on its output.

The two peaks at either end of the spectrum was found to be interference caused by surrounding components such as monitors and did not cause any significant interference

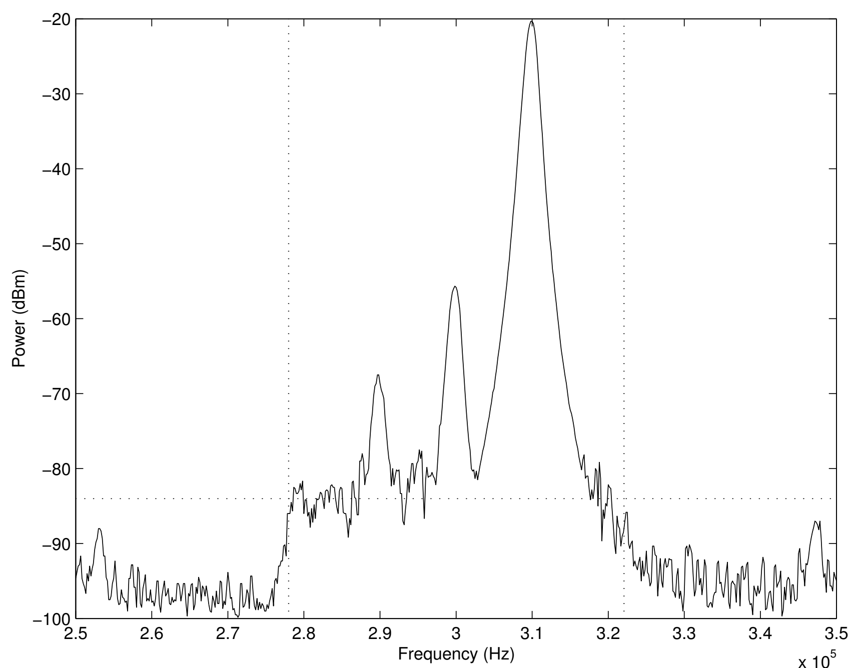


Figure 5.3: *Quadrature-mixed soundcard output with I and Q data inputs applied*

with the measurements.

Addition of data inputs

Figure 5.3 shows the spectral output with the I and Q data inputs applied. Recall that the source data was restricted to 8 bits — this is indicated by the increase in the quantisation noise floor between 280khz and 320kHz. This observation confirms the 44.1 kHz sampling bandwidth used on the input, as indicated in the window between the vertical markers in figure 5.3. The quantisation noise floor level in the demarcated window in figure 5.3 lies at approximately −84 dBm. The upmixed component is positioned at 310 kHz (with a level of −20.33 dBm) as expected, and carrier leakthrough from the DC offset produced by the soundcard still remains.

There is, however, a new spurious component at 290 kHz with a power of −67.5 dBm (i.e. 47.5 dB below desired signal). There is a possibility that this spurious frequency is generated due to phase differences (i.e. not a perfect 90°) between the I and Q channels, but this is unlikely. This is because the signals were accurately generated using MATLAB — analysis later also reveals that a more probable cause of the component at 290 kHz could be the result of amplitude imbalance between the left and right channel outputs of the soundcard. An effect of this nature is typically attributed to unmatched DACs, and poorly matched amplitude responses of the soundcard's lowpass filters (van Rooyen 2000). Once again, this is understandable since in normal computer audio applications, this slight

amplitude discrepancy is not very significant.

Determination of soundcard spurious levels

It is possible to compensate for this imbalance by altering the amplitude level of one of the channels — this also enables us to determine approximately what percentage of amplitude imbalance exists between the two channels, by trial and error techniques. Therefore, the Q channel was arbitrarily selected for adjustment, and trial and error showed that at a certain amplitude the spurious frequency (caused by amplitude imbalance) at 290 kHz was at its lowest. This amplitude compensation clearly shows a decrease in the spurious sideband component as illustrated in figure 5.4 — this minimum was found with the I channel level set to 1 and the Q channel level set to 1.007 which concludes that there is an approximate 0.7% amplitude imbalance between the two channels.

This result is, however, just an approximation. A more accurate result can be obtained using the equation 3.33 in chapter 3 to determine the SFDR for amplitude imbalance. Using equation 3.33 gives an amplitude deviation of 0.0085 for a SFDR of 47.5 dB for an input signal with unity amplitude. This result of 0.0085 (or 0.85%) amplitude deviation is important since it must be taken into account when the development of the emulated system takes place in order to correctly represent the imperfections of the signals from the soundcard prior to the input of the quadrature mixer. The output of the desired component was measured at -20.33 dBm, which will result in a lower amplitude deviation error level. This effect is taken into account and its exact setup is further explained in the setup of the emulated system.

Using figure 5.3, we are now also able to measure the difference between the desired quadrature mixed signal and the carrier leakthrough component. By observation, this was found to be 35.3 dB. Converting 35.3 dB to an absolute level means that the upmixed desired component at 310 kHz is 3388 times greater than the carrier leakthrough component. Therefor, with respect to the desired signal, the carrier leakthrough component is relatively small.

Calculation of soundcard LPF frequency response

In order to obtain the frequency response of the soundcard's LPF, MATLAB's built-in `randn` function was used to produce some white gaussian noise as inputs on the I and Q channel to the quadrature mixer. The syntax used to generate the white gaussian noise directly on the output of the soundcard is as follows:

```
wavplay(randn(1e6,2),fs)
```

Recall that `fs`, the sampling frequency, was set at 44.1 kHz. The resulting plot is shown in figure 5.5 and clearly shows the characteristic LPF response subsequent to frequency

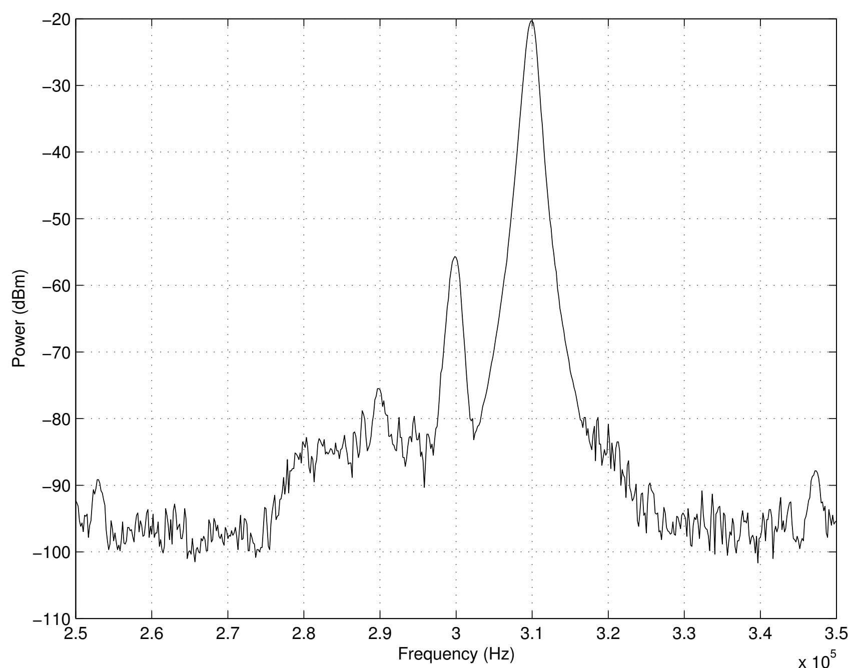


Figure 5.4: *Amplitude compensated output showing a decrease in the adjacent sideband component*

translation. The cut-off frequency was measured to be at approximately 22.05 kHz (which is half of 44.1 kHz, the sampling frequency) since the power difference at this point was approximately 3 dB. The notch seen at 300 kHz is as a result of the soundcard's built-in filter response — judging by the output of figure 5.5, it can be seen the soundcard's filter does not allow very low frequencies to pass (including DC). This means that the filter actually has a bandpass response with a very low first cut-off frequency. Therefor, after frequency translation, the notch appears at the carrier frequency (300 kHz). Because the soundcard is not DC coupled, carrier leakthrough compensation (by using of a DC offset) is not possible. This technique could have be used to determine, by trial and error, the level of DC offset required for the emulator. However, this parameter can easily, and very accurately, be calculated using equation 3.34 in chapter 3. The exact level of DC offset will be calculated in the section concerning the emulator's setup.

Addition of quadrature impairments

Now that the characteristics of the soundcard have been determined, the next step is to deliberately add some quadrature impairments. This facility is conveniently provided by the vector signal generator and allows the user to add a percentage of the desired quadrature impairment. The quadrature impairments will first be added independently for examination purposes, and then the total effects will be combined.

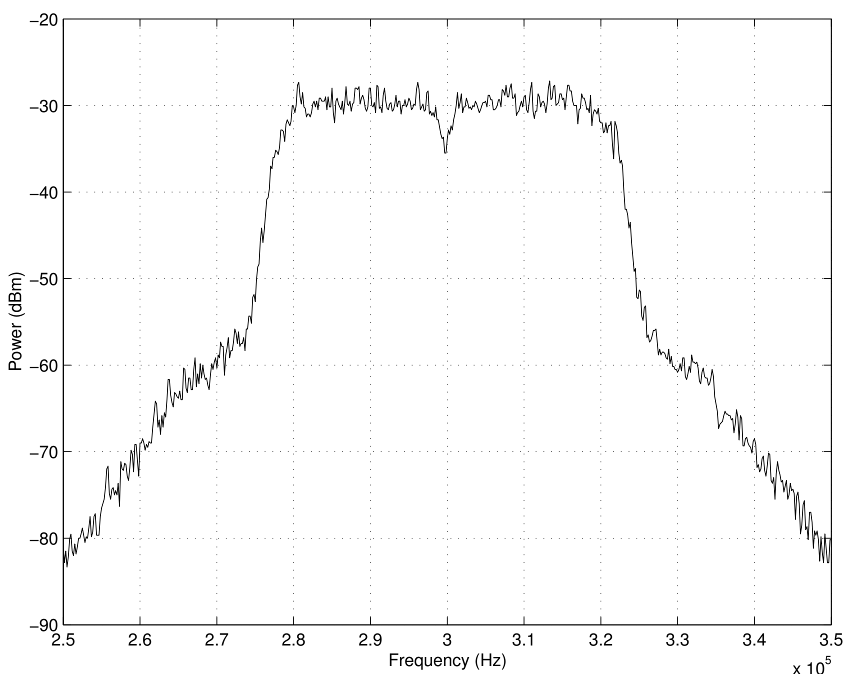


Figure 5.5: *Determination of soundcard's filter transfer function*

The first non-ideal quadrature mixing effect, carrier leakthrough, is investigated by adding a 1% carrier leakthrough component. The resulting effect can be observed in figure 5.6 which shows a definite increase in the carrier leakthrough component, which now has a level of −44.3 dBm. Depending on the phase of the carrier leakthrough component caused from the soundcard, the overall combination of both leakthrough effects (the soundcard and the quadrature mixer) can either be constructive or destructive. Refering once again to figure 5.2, the measured difference between the sound cards and quadrature mixers carrier leakthrough components is 12.83 dB. The difference value of 12.83 dB is important and it will be shown later in section 5.2.1 how it is used to implement the correct amount carrier leakthrough in the quadrature mixer model.

Next, the leakthrough impairment is again set to zero and a 1% amplitude imbalance is introduced using the vector analyser. We would expect the adjacent sideband (at 290 kHz) level to increase and this is now evident in figure 5.7. Recall that there was already a 0.7% amplitude imbalance introduced by the soundcard and the 1% of amplitude imbalance introduced from the quadrature mixer is now being added to this to yield a total output level of −63 dBm.

The last impairment added is phase imbalance. Recall from chapter 3 that a phase imbalance leads to the same effect as amplitude imbalance — a sideband spur at 290 kHz. With all the other quadrature impairments set to zero again, the phase imbalance was arbitrarily selected as 3°. Looking at figure 5.8, a difference of 32 dB is now evident

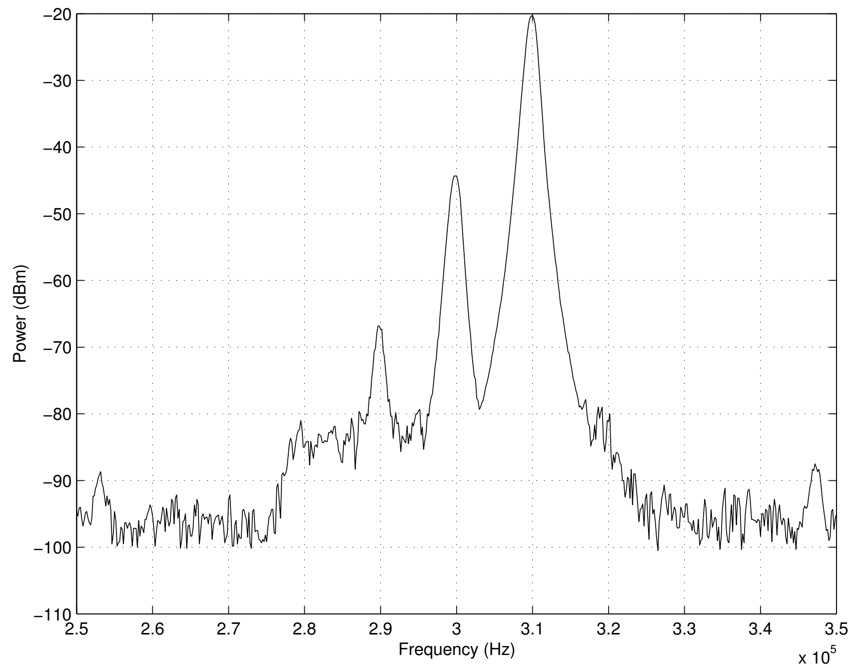


Figure 5.6: *Output with 1% carrier leakthrough added*

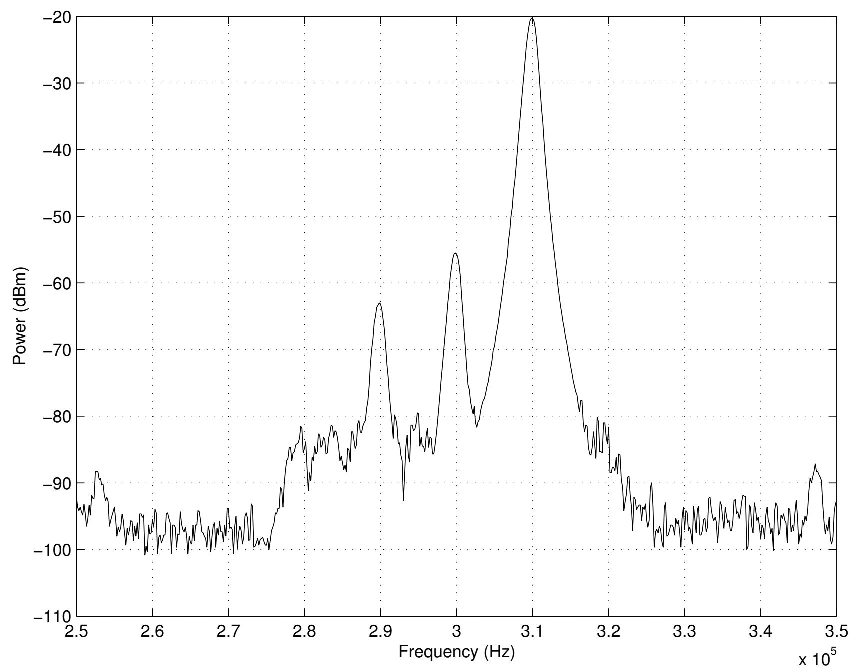


Figure 5.7: *Output with 1% amplitude imbalance*

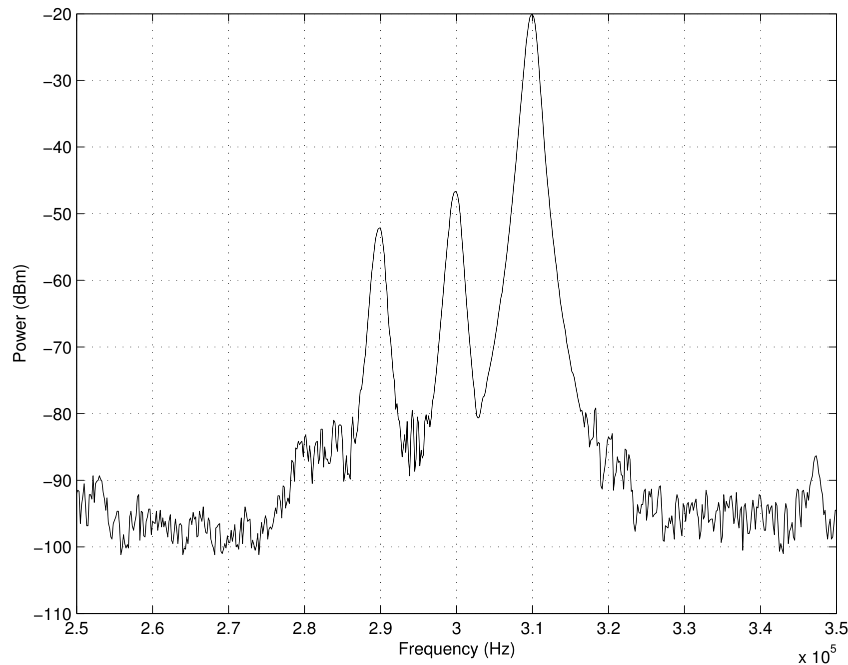


Figure 5.8: *Output with 3° of phase imbalance added*

between the desired component and the spurious component at 290 kHz due to the 3° phase imbalance — this is exactly the level predicated in equation 3.35 in chapter 3 by making the SFDR the subject of the formula for phase imbalance.

Careful inspection of figure 5.8, however, reveals that the carrier component has increased as well. Theoretically this is not expected. It is speculated that this change in carrier leakthrough power could be as a result of the internal operations within SMIQ signal generator. Therefore, the SMIQ signal generator could be attempting to maintain a certain total signal power, irrespective of the distortion applied to the signal. In this case, the signal generator was set up for a signal with an output level of -20 dBm and then set up with some phase imbalance. The signal generator is therefore still going to try to control the output power to lie at exactly -20 dBm by internally adjusting both channels in order to create a modulated signal with a power of -20 dBm.

Stated otherwise, the attempt to maintain a constant output power could be part of the gain control circuitry of the SMIQ signal generator. It is therefore assumed that the constant-power theory (van Rooyen 2003) applies to the output of the SMIQ signal generator.

Figure 5.9 just shows the combination of all three quadrature impairments added to the signal. The levels observed for the hardware setup's measurements will be examined in the following section, when the outcomes of the emulated system are studied.

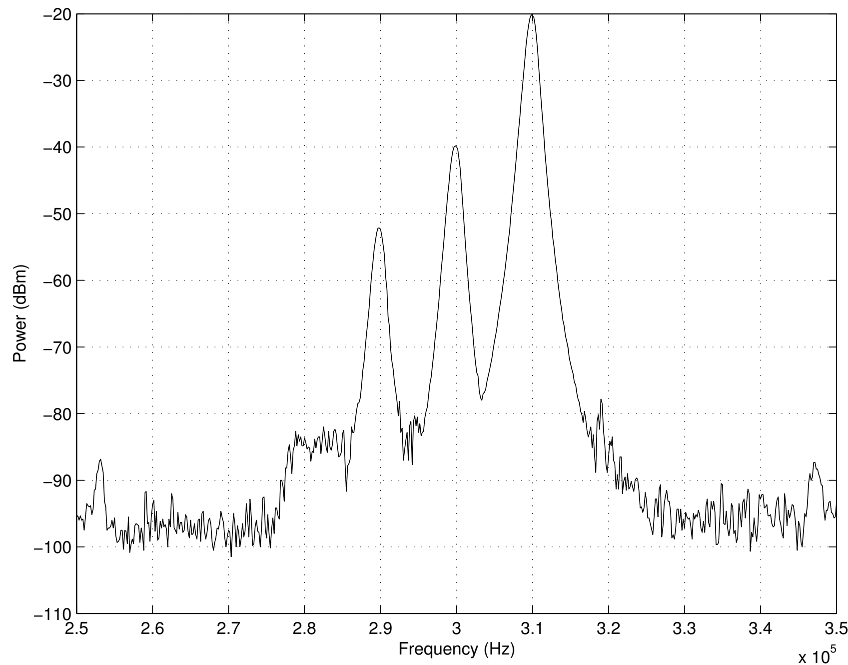


Figure 5.9: *Combination of the three quadrature mixer impairments*

5.2 Emulated system setup

5.2.1 System overview

The primary aim of chapter 5 is to evaluate the accuracy of the emulator relative to the physical hardware setup. In order to do this correctly, the same test setup conditions, that were measured in the physical setup, need to be applied to the emulated setup. Recall that the idea behind emulating a system is not to accurately represent the system in terms of its precise architectural makeup. Instead, appropriate models should be used in order to mimic the non-ideal behaviour of the components in the system so that, externally, the behaviour of the emulated system should correspond with (or at least come close to) an equivalent physical setup. The complete setup of the emulated system was done using the front-end application as described in section 4.6 of chapter 4. An image of the opening screen shot is shown in figure 5.10.

Emulator input parameter calculations

At this stage it might be important to state that the emulator's use depends highly on the designers requirements as well as the type of the system required for emulation. Some of the different types of scenarios might include:

1. The case where a complete and thorough understanding of the hardware system is

```
-- SDR Hardware Emulator FrontEnd Application Ver.1.0 --
1. Add a component
2. Remove a component
3. View list of selected components
4. Link components
5. Quick Link
6. View component links
7. Run!
0. Exit

Please select a menu option:
```

Figure 5.10: *A screen shot of the user front-end application showing the various options.*

known. This will obviously lead to the most accurate results in the emulated setup.

2. Only outputs at certain points of the hardware system are known. This will allow the designer to work backwards from the measurements in order to setup up the emulator in the best possible way.

3. Where no knowledge of the system is known and its development in the emulator is designed on paper. This kind of scenario would occur quite often, since it is the purpose of the emulator to allow the designer to experiment with different design ideas prior to hardware implementation.

For the hardware setup in this chapter however, option 2 is applicable. This is because no direct knowledge of the system prior to the output of the soundcard is known. Stated otherwise, it was made very difficult to determine the outputs of each component in the soundcard due to restricted accessibility in order to measure the intermediate points on the soundcard. In order to best address this limitation, the spectral output of the soundcard was used to determine the soundcard's performance. Recall that figure 5.3 was used to illustrate the output of the soundcard (subsequent to being passed through the quadrature mixer). With the input signal applied, the spurious component and carrier leakthrough in figure 5.3 indicated that some gain and DC offset error was present on the output of the soundcard respectively. It was stated earlier that the exact source of these imperfections were not known but most like cause was due to mismatched components — particularly the DACs. The quadrature mixer used in the vector analyser is an extremely high-precision device and its contribution to any quadrature errors being added (with no quadrature impairments added by the user) was considered negligible.

In order to adjust the output amplitude from the soundcard, the Windows operating system's volume control was used. From figure 5.3, the output power of the source signal was measured on the spectrum analyser at -20.33 dBm. Taking the 0.5 dB gain on the vector analyser (for the quadrature mixer output) into account, this translates to a signal level of 4.1 mV being generated on the output of the soundcard according to the following calculations: The -20.33 dBm less the 0.5 dB gain of the vector analyser yields an output of -20.83 dBm. Converting the power level, P_s, to a voltage level, V_i with the equation $V_i = \sqrt{2 \cdot P_s}$ gives a result of 0.0041 or 4.1 mV.

The amplitude of the cosine (I) and sinusoidal (Q) source files used in the emulator should be set at this level, Vi, so that the input to the quadrature mixer sees an input level of 4.1 mV. The setup parameters for the input waveforms should remain the same for the emulated setup and in summary are given as:

- Input frequency for cosine and sinusoidal waveforms = 10 kHz
- Input sampling rate = 44.1 kHz
- Input amplitude = 0.0041 V

Sample number considerations

The exact number of samples used for the generation of the source waveforms need not be exactly equal to the hardware setup's configuration, but it must at least be large enough in order to yield accurate Fast Fourier Transform (FFT) results. This is because the number of points and sampling rate ultimately determine the resolution and frequency range of the spectral plot on x-axis. Therefore, the number of samples chosen in the emulator can be taken in such a way as to maximize the frequency resolution. However, in practical measurements such as those performed in the hardware setup, it is not always possible to achieve this. Also, with the emulation, the sampling rate can change at any time. Therefore, the number of samples was arbitrarily chosen as 3000. In the section to follow, featuring the results of the emulation, the fundamentals of FFT-based analysis are briefly discussed in order to gain a better understanding of FFT-based measurements.

Representation of soundcard inaccuracies

In order to represent the inaccuracies (gain and DC offset) of the soundcard, there are essentially two ways of performing this in the emulator:

1. Include these inaccuracies in the (I and Q) source files themselves.
2. Develop a separate model that adds gain and DC offset to the signal at any point in the system.

Although both options will ultimately produce the same results, employing option 1 would not be an very accurate model of the emulated system. As stated earlier, certain DACs can cause some gain and offset errors to arise due imperfect components that make up the device. Therefor, in all likelihood, it is the output of the DACs at which the gain and DC offset errors of the soundcard occur. Therefore, it would be more prudent to insert a gain and DC offset model directly after the DAC model. Option 2 is therefore the best solution and is simply implemented as the algorithm:

$$y = m \cdot x + c \tag{5.1}$$

where the output, y, is simply a function of the values m and c, the gain and DC offset error respectively. In order to calculate the exact level of gain and DC offset, we need to refer to figure 5.3 again. Using the SFDR equations of the section 3.6.6 in chapter 3, we can make the error the subject of the formula and use the SFDR in order to determine value of this error. Therefore, in order to calculate the amount of DC offset added by the sound, figure 5.3 displays a carrier leakthrough component that is 35.3 dB below the desired signal. Recalling equation 3.34 of chapter 3 and making ρ the subject of the formula puts the equation into the following form:

$$\rho = \frac{A}{10^{\left(\frac{\mathrm{SFDR}_\rho}{20}\right)}} \tag{5.2}$$

With the amplitude level, A, set as 0.0041 (4.1 mV) and SFDR_ρ set to 35.3 dB gives a DC offset error level of 7.04×10^{-5}. The same process can be used for the calculation of the gain error level from equation 3.33 of chapter 3 to yield the following form:

$$\beta = \frac{2 \cdot A}{10^{\left(\frac{\mathrm{SFDR}_\beta}{20}\right)} - 1} \tag{5.3}$$

Again, with the amplitude level A set to 0.0041, and (from figure 5.3) SFDR_β is set to 47.5 dB yields gain error of 3.47×10^{-5}. It is not important on which channel the impairments are added, since the same result is obtained either way. Therefore, only a single gain and DC offset model can be placed on the output of either channel of the selected DAC. Another important aspect of figure 5.3 to take into account is the level of the noise floor at approximately −100 dBm. It will be shown later in the chapter how the level of AWGN will be added in order to introduce the correct amount of noise to the emulated system.

Acquisition of emulated waveforms

Figure 5.11 is provided in order to visualise the emulated system setup. So far, only the synthesised input waveforms setup have been illustrated. The connections in figure

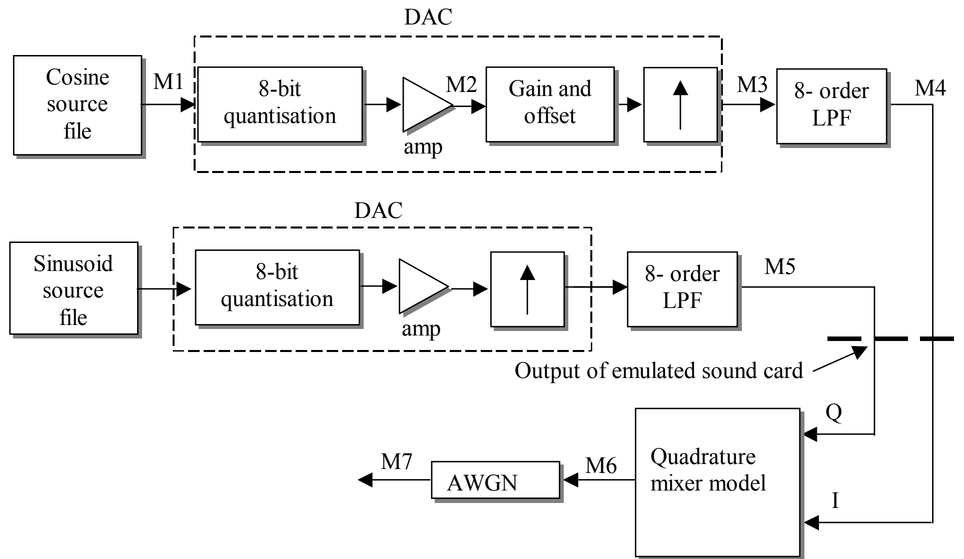


Figure 5.11: *Illustration of the emulated setup indicating the position of the various monitor files*

5.11 marked with a 'M' indicate points at which monitor files where placed within the emulator. The monitor files are used capture the output at desired points in order to perform post-processing such as data visualisation. Therefore, the monitor at point M1 is used to acquire to output of the synthesised I-channel waveform. It might be thought of as an unnecessary to place a monitor file at this point considering that the input file itself can simply be loaded direct into MATLAB. The reason, however, for the inclusion of this monitor file is to ensure that the synthesised waveform generated in MALTAB and the captured waveform in the monitor file correspond — this is done to ensure that no discrepancies lie in the acquisition of the data through the monitor files.

DAC model initialisation

Moving sequentially from the source components, the next component is the DAC — one for each channel. The DAC, in the emulated system, is made up of a cascade of three models:

1. A quantisation model
2. An ideal amplifier (gain module)
3. A gain and DC offset model (for the I-channel DAC only)
4. An upsample model

In the emulated setup, no INL model is included. This is because no measurable INL was observed. However, the advantage of emulation is that we do not need to represent the DAC architecture exactly, but only its behaviour. In order to achieve this, the gain and DC offset model will be used to represent the effects as a result of the possible INL errors. Recall that the value of these parameters were calculated earlier as:

- DC offset — 7.04×10^{-5}
- Gain error — 3.47×10^{-5}.

For the hardware setup, software techniques were used in order to effectively enable 8-bit quantisation. The emulator's quantisation model, with the input parameters set at 8-bits, was used to achieve this. The quantisation model is a normalised function which means that the input signal must span ±1 in order for the function to work correctly. An ideal amplifier is used subsequent to the quantisation model in order to attenuate the quantised signal to the desired level of 0.0041. This setup is done for both the I and Q channel. A monitor file, M2, is placed on the output of the amplified quantised signal on the I channel to probe this output point.

A gain and DC offset model is placed directly on the output of the I channels amplifier. This was done in order to include the non-ideal effects of the soundcard as was observed in the hardware setup's measurements. The syntax used to implement the gain and DC offset error was given as

$$y = m \cdot x + c \tag{5.4}$$

where c would be set at a value of 7.04×10^{-5}. The gain error value requires pre-calculation before the parameter is entered as a suitable gain factor in the amplifier model. The value for the gain variable, m, is calculated as

$$m = \frac{A_v + G_e}{A_v} \tag{5.5}$$

where A_v is the amount of attenuation required, in this case 0.0041 and G_e is the amount of amplitude deviation error, which in this case is 3.47×10^{-5}. This gives a value of 1.0085 for m, which will be used an the gain input parameter on the gain and DC offset model.

Upsample model initialisation

The upsample model is used to generate the usual sample-and-hold harmonics that are characteristic of all DACs. The upsample model increases the effective sampling rate and therefore makes the upsampling requirements of the quadrature mixer (to be explained later) slightly easier. In fact, the upsample model in the DAC will be used in order to completely avoid upsampling in the quadrature mixer. The main advantage of this

approach is that no filtering of the upsample harmonics are required within the quadrature mixer. A quadrature mixer would also require the two LPFs to have an adjustable cut-off frequency that is depended on the upsample rate. Developing a LPF for this purpose, with a order suitable enough to effectively remove the upsampling sample-and-hold harmonics, can get very complex and is beyond the scope of this thesis. If it is within the interest of the designer to pursue the area of suitable methods for sample rate conversion within the quadrature mixer, the reader is referred an in-depth treatment of the topic in Chapter 10 of Proakis (Proakis & Manolakis 1996).

Generally all practical DACs generate some form of sample-and-hold harmonics due to the internal operations of the DACs. The DAC used in the soundcard is no exception, and in order to properly represent the system in the emulator, this effect must be included. The upsample model will therefore be used to introduce this effect. The input parameter to the upsample model is the upsampling rate. Because the lowpass filter following the DAC is used to suppress these sample-and-hold harmonics, it is made impractical to measure the order of the sample-and-hold harmonic generated by the soundcard's DAC. The order of the sample-and-hold harmonics is, however, directly proportional to the upsampling rate. Because the upsample model will be used to relieve the quadrature mixer of any upsampling duties, as mention in the previous paragraph, the upsample rate will be set to 30. This is because the input sampling rate of 44.1 kHz, upsampled by a rate of 30, gives a resulted sampling frequency of 1.323 MHz — this will ensure that the quadrature-mixed signal with a carrier frequency of 300 kHz can effectively be represented without aliasing. The monitor file, M3, will be used to observe the output after the upsampling process.

Note that no glitch model is included in the emulation since it was not possible to measure the amount of glitch directly at the output of the DACs. Attempting to measure the level of glitch impulse on the output of the soundcard resulted only in the desired glitch transients being filtered by the soundcard's lowpass filters.

LPF model initialisation

The LPF, as mentioned earlier, would be used to remove the sample-and-hold harmonics. From figure 5.5 it could be observed that the LPF in the soundcard has a cutoff frequency of 22.05 kHz. Careful observation of figure 5.5 actually revealed that the filter has a bandpass response. However, the system will not be operating with low range frequencies and it is therefore not critical to model this as a BPF. In order to determine the order of the filter model in the emulator, some trial and error work was needed. Using MATLAB as a benchmark, it was determined that a 8th-order LPF filter best approximated the measured response for the sound cards LPF, using the same input. Therefore, 8-order butterworth IIR filter coefficients with a cutoff frequency designed at 22.05 kHz will be used in the emulator's filter model. The monitor file, M4, will be used to acquire the

output at the LPF. From figure 5.11 we see that the monitor file at M5 is also placed at the output of the Q channels LPF. This is done in order to verify that the results (less the effects of the gain and DC offset on the I-channel) correspond with each other.

Quadrature mixer model initialisation

The next component in the system is the quadrature mixer model. Recall that the I and Q signals at the input of the quadrature mixer will have a sampling rate of 1.323 MHz. This ensures that the quadrature mixer does not perform any upsampling. The frequency parameter for the quadrature mixer was calculated by dividing the desired carrier frequency of 300 kHz by the output sampling frequency of 1.323 MHz. This results in the quadrature mixer's frequency parameter value of 0.227 which will be used directly in the model. A gain of 0.5 dB was set in the vector analyser and this translates to a gain level of 1.059 for the amplifier in the quadrature mixer.

In order to add the correct amount of WGN to the quadrature-mixed RF signal, the value of the noise power must be calculated beforehand as follows:

$$N_P = 10^{\frac{F}{10}} \tag{5.6}$$

where N_P is the noise power parameter used in the AWGN model. The variable, F, is the average height (in dBm) of the noise floor. From figure 5.3, the noise floor level can be observed to lie at approximately -96 dBm. Making D = -96 dBm gives a noise power of, N_P, of 2.51×10^{-10}.

5.2.2 Emulated system outcomes

FFT overview

Before the measurements of the emulator are presented, some background understanding is required on the way the Fast Fourier Transform (FFT) is used to calculate the spectral plots. The Fourier transform is a mathematical tool used for establishing the frequency content of a signal (Rhode & Whitaker 2001). The discrete Fourier transform (and its computationally faster variant, the FFT) is used to calculate the Fourier transform of sampled signals. The Fourier transform operates by assuming that the time-domain signal being operated on is periodic and repeats itself indefinitely (Rhode & Whitaker 2001). Most real world signals fall short of these properties and at best, the Fourier transform provides an approximation of the frequency content of the signal. Therefore, in practical work, a convenient time window is usually chosen and the periodicity and time windowing assumptions of the Fourier transform are normally not met.

A phenomena known as spectral leakage occurs as a results of taking the FFT of an signal that does not meet the assumptions of the Fourier transform. In the time domain,

this can be thought of as taking a time window of a signal which has a non-integer number of cycles. Repeating this time window indefinitely and connecting the end of each window to the beginning of the next leads to discontinuities. These discontinuities in-turn represent spurious energy in the frequency domain which causes some of the fundamental signals energy to leak over into other frequencies, resulting in a lowered main peak and more energy in adjacent frequencies.

Windowing

The situation can be improved somewhat by taking more samples, but that does not necessarily address the problem since the discontinuities can actually worsen. This is because increasing the number of samples could cause the discontinuity error between to time frame windows to become greater. A technique used to remove (or at least reduce) these discontinuities, known as windowing (Rhode & Whitaker 2001), can be applied — but with some trade offs. Incidentally, windowing occurs even when no window is used. This is because, by the very nature of taking a snapshot of the input signal in time, the signal is multiplied with a rectangular-shaped window of uniform height – this is know the Rectangular or Uniform window. Other types of windows include the Hanning, Hamming, Blackman-Harris, Blackman and Flat top window to name but a few (Cerna & Harvey 2000). Windows are characterised in the frequency domain by various parameters, with each window having its own properties. With windows in general, the trade-off is generally less spectral leakage at the cost of spectral resolution.

Different windows are used for different applications and selecting an appropriate window is primarily based on the signal's frequency content (Cerna & Harvey 2000). Table 5.1 summaries some of the suggested window choices based on the spectral content of the signal (Cerna & Harvey 2000). The signal content for the emulator consists generally of a combination of sine waves and therefore, according to table 5.1, the Hanning window would be most appropriate. Selecting an alternative window, however, can be done fairly easily in order to find the best window for the application.

Table 5.1: *Suggested window choices based on signal frequency content (adapted from (Cerna & Harvey 2000))*

Signal content	**Window**
Sine wave or combination of sine waves	Hanning
Sine wave (with preserved amplitude accuracy)	Flat top
Closely spaced sine waves	Uniform, Hamming
Signal content unknown	Hann

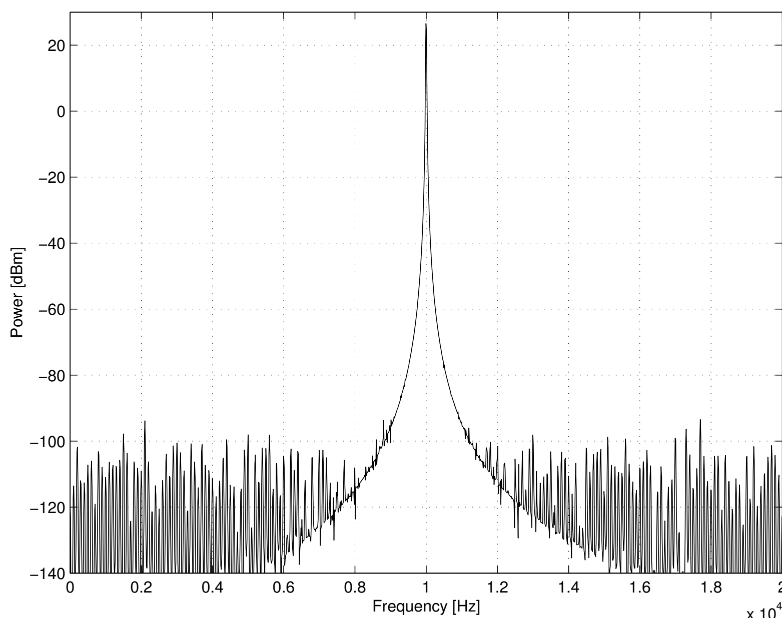


Figure 5.12: *Spectrum of the 10 kHz input cosine waveform with a power of 27 dBm.*

One of the effects of windowing is that the overall amplitude accuracy of the signal is altered by the type of window used. In order to compensate for this error, the windowed signal can be re-scaled by dividing the window array by the coherent gain for that particular window. For a uniform window, the coherent gain would naturally be unity, but for a Hanning window, the coherent gain is 0.5 (Cerna & Harvey 2000). Unless mentioned otherwise, all spectral plots for the output of the emulator were operated on using a Hanning window as well as scaling it with its coherent gain factor.

Referring again to the the emulation setup in Figure 5.11, the outputs at various points will now be examined. We would expect to see similar results to those measured in the hardware setup, since the input parameters to the models were setup based on the outputs of the hardware measurements. It must be said again, however, that the hardware system measurements obtained are, at best, approximations of the system setup and this point must be kept in mind when analysing the results of the emulation.

Source file data measurements

Referring to figure 5.11, the data acquired at the various monitor points from the emulation will now be examined, beginning at the source file, M1, and ending at the monitor point M7. Figure 5.12 shows the spectrum of the 10 kHz cosine waveform at the monitor point M1. The reason a monitor file was placed at this point was to ensure that no errors were introduced from the acquisition of data from the monitor file. The power level of the fundamental signal is measured at 27 dBm, which corresponds to the level of unity

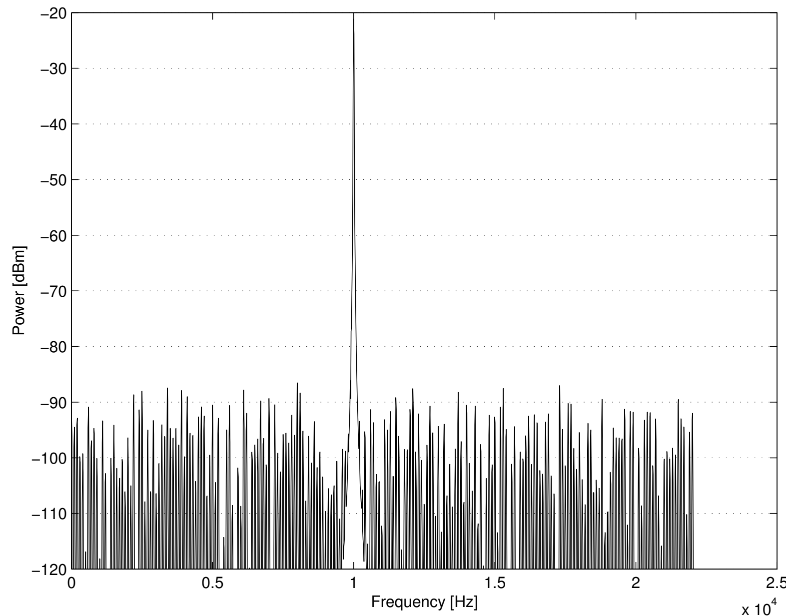


Figure 5.13: *Scaled output after quantisation*

for the synthesised input waveform files.

The quantisation model that follows requires that an input signal level of unity amplitude and therefore the power of the component at 10 kHz lies at an expected level of 26.57 dBm. This level will be scaled to the correct level after quantisation using an ideal amplifier. The spectral components at approximately -100 dBm are there as a result of the windowing process.

DAC model measurements

The DAC that follows the first monitor file is composed of the following components:

- A 8-bit quantisation model
- An ideal amplifier
- A gain and offset error model
- An up-sample model

The output of the monitor file point, M2, is shown in figure 5.13. The output of the quantisation model was scaled to the correct level (using an ideal amplifier model), which now lies at exactly -21.17 dBm. The quantisation noise floor can be seen to lie at approximately 86 dB below the fundamental signal. This observation correlates well with the output of the hardware setups level of approximately -84 dB. The time domain signal was operated on in order to calculate the SQNR of the signal in figure 5.13. Therefore, the

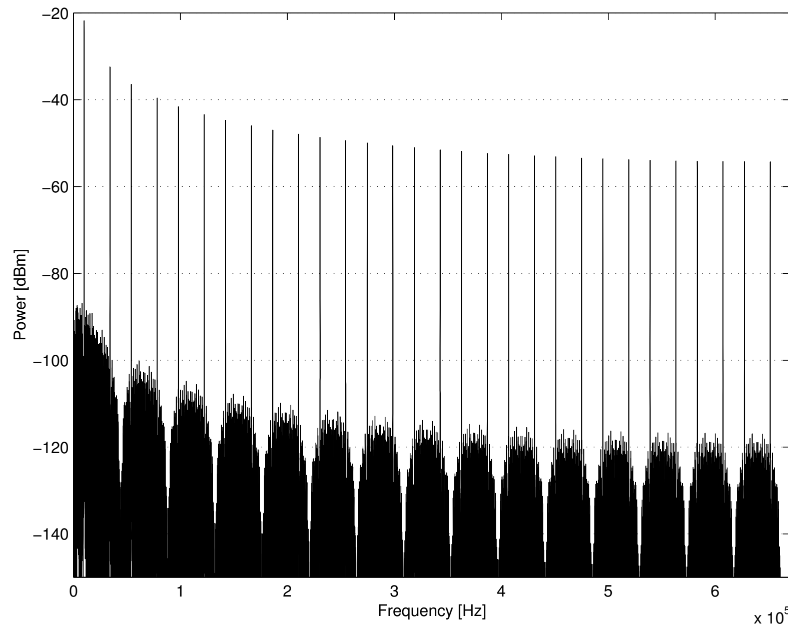


Figure 5.14: *Frequency plot for the monitor file, M3, showing the effects of upsampling by a rate of 30.*

quantised signal used was the unity amplitude signal obtained directly after quantisation, prior to amplification. From first principles, the SQNR was calculated using the equation:

$$\mathrm{SQNR} = 10 \cdot \log \frac{P_s}{P_e} \tag{5.7}$$

where P_s and P_r is the power in the original and (quantised) error signal respectively. This yields a result of 50.05 dB. This value is extremely close to the expected SQNR value of 49.92 dB, which was calculated from the formula $\mathrm{SQNR} = 6.02N + 1.76$.

Upsample model measurements

The output plot for the monitor file, M3, shows that the signal was up-sampled by a rate of 30 — this is evident by the observation of the 30 sample-and-hold harmonics. In addition to this, the DC offset and gain errors were also added to the upsampled signal. The peak of the desired component was measured at −21.84 dBm which, strangely enough, indicates that the signal was slightly attenuated. In order to investigate what the cause of this attenuation was, a monitor file was later placed directly after the gain and offset model. This was done to ensure that the correct amount of gain was added. Looking at the results of this monitor file in the frequency domain revealed an spectral peak at a level of exactly -21.84 dBm which corresponds to a gain of 1.0085 — exactly the level set by the gain and offset model. This result indicates that the error in signal level obtained at the output of the I channels upsample model is a result of the upsample model itself.

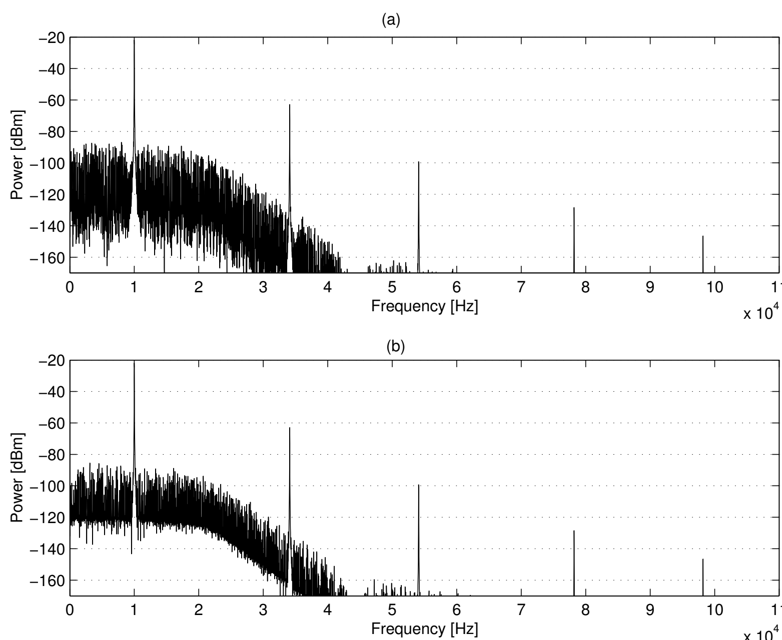


Figure 5.15: *Output showing the filtered signal on the (a) I channel and (b) Q channel output. The cut-off frequency can be observed at 22.05 kHz for both signals.*

This minor discrepancies between results could be attributed to the spreading of energies in FFT in the upsampled signal. In addition to this, there is the possibility that the windowed FFT might have resulted in a slight loss of amplitude information.

It should be noted, however, that this slight discrepancy in the signal levels should not be a major cause for concern. This is because the models used in the I and Q channel, up until the input to the quadrature mixer, are subject to exactly the same signal processing (less the effects of the gain and offset model, of which no signal processing errors where found) and therefore the changes in amplitudes should be equal for both channels. Stated otherwise, we could expect the absolute value of the fundamental signal to be slightly attenuated after quadrature mixing, but the relative levels of the signals should remain the same.

LPF model measurements

The data at the monitor points M4 and M5 should be exactly the same except for the addition of the gain and DC offset errors on the I channel. Figure 5.15 (a) and (b) shows the outputs of the I and Q channel respectively after being passed through there respective 8th order low pass filters. From figure 5.15, observing the second harmonic at approximately 40 dB from the fundamental signal, it is clear that some of the lower order sample and hold harmonics could be present in the quadrature mixed signal. The fundamental signal of the I channel and Q channel output in figure 5.15 was measured

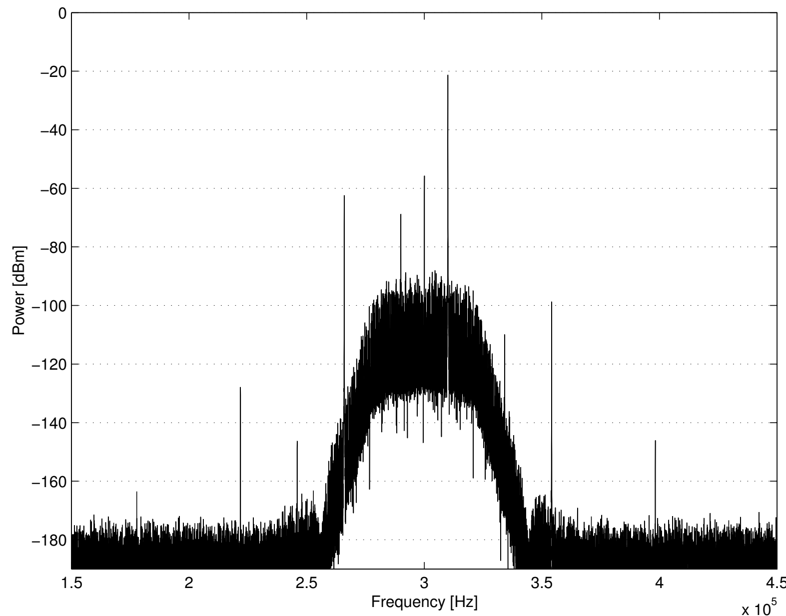


Figure 5.16: *Output of the quadrature mixer model, clearly showing the effects of the low order LPF, resulting in frequency translation of the lower order sample-and-hold harmonics*

at a level of -21.81 dBm and -21.88 dBm respectively. This indicates the expected gain difference of exactly 0.07 dB between the two signals — the error set by the gain and offset model.

Quadrature mixer model measurements

The output of the quadrature mixer model is captured in the monitor file, M6, and the respective frequency plot is shown in figure 5.16. As expected, the effects of upsampling have shown through — the second order sample-and-hold harmonic at 270 kHz still lies at a level of −62.43 dBm. This is obviously an undesired effect, but a filter with a sharper roll-off should address this problem quite easily. In figure 5.16 it can be seen that no AWGN has been added yet, and this is why the noise floor is still at approximately −180 dBm.

The desired signal at 310 kHz was measured at a power level of −21.35 dBm, which, recalling the output of the LPF on the Q channel at −21.88 dBm, indicates this the quadrature-mixed signal was amplified by a gain of 0.5 dB — exactly the gain set for the quadrature mixer. If we actually go back and look at the levels observed for the hardware setup measurements in figure 5.3 and compare this to the emulated setup's measurements, we will see that there is a difference of 1.02 dB. This error was however, identified at the output of the upsampling model, which had approximately 1 dB added to its output.

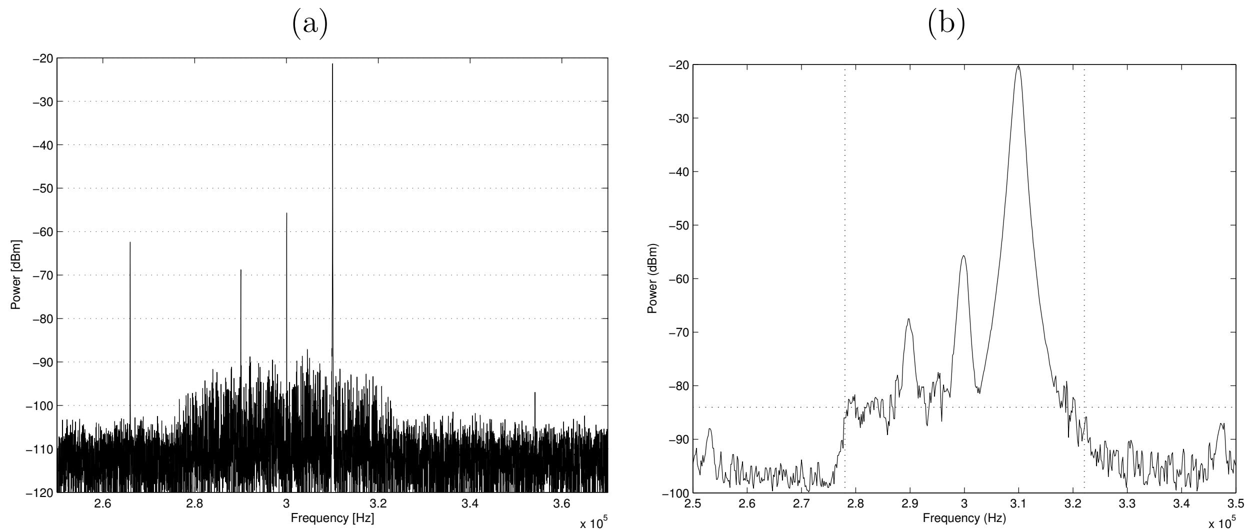


Figure 5.17: *Spectral plot of the emulated quadrature-mixed output (a) showing the inclusion of the AWGN against the previous output of the spectrum analyser (b) from the hardware setup shown earlier. The peaks at either end of the spectrum of the emulated output is as a result of improper filtering of the sample-and-hold harmonics.*

The output of the quadrature mixer illustrates how this error level carried through to the output of the quadrature mixer.

AWGN level measurements

Before the absolute and relative signal levels are studied further, the addition of AWGN must first be taken into account. The quadrature-mixed output with the AWGN added was obtained from the monitor point M7. The AWGN noise floor is shown in figure 5.17 and lies at approximately −105 dBm. This noise level is indifferent to the expected noise level of −95 dBm as measured in the physical hardware setup. Although attempts were made to address this noise level discrepancy, these proved unsuccessful. It is assumed, however, that the error lies somewhere with the usage of the AWGN code and that further analysis for implementation of the AWGN code could possibly resolve the matter.

Final output measurements

Measuring at the absolute levels in figure 5.17 reveals that the desired up-mixed component at 310 kHz lies at −21.35 dBm. Comparing this to the level of -20.3 dBm from the hardware setups output indicates that the two outputs differ by 1.05 dB. As mentioned earlier, this discrepancy was already picked up at the output of the emulated soundcard's LPFs. Therefore, the absolute level of the output was expected to be slightly less that expected. This discrepancy in results can also be attributed to the assumptions needed

by the FFT in order to produce perfect FFT results, of which the signals in the emulator failed to meet. The carrier leakthrough component in figure 5.17 was measured at -55.7 dBm, which measures close to the hardware setups output of -55.6 dBm. The last signal of interest is the spurious component at 290 kHz due to amplitude imbalance in the I and Q signals. The level of the amplitude imbalance spur was measured at -68.77 dBm — 1.3 dB off from the expected level of 67.5 dBm. These results yield the following SFDR measurements for each spurious component with respect to the fundamental signal:

- SFDR for the carrier leakthrough component = 34.4 dB
- SFDR for the amplitude imbalance component = 47.4 dB

The carrier leakthrough SFDR measured for the hardware setup was 34.4 dB and the SFDR for the amplitude imbalance spur was measured at 47.4 dB. Given the possibly of some error introduced as a result of imperfect FFT measurements, the results displayed by the emulator are relatively close to the measurements of the hardware setup, for the same input conditions. Although the additional effects of quadrature impairments need to be included, it is already possible to obtain an idea of the performance of the emulator at this stage.

Addition of carrier leakthrough error

The effects of adding the quadrature mixers impairments will now be illustrated in the emulator. The challenge in this regard was how to determine the values of the input parameters needed to specify the correct value of quadrature impairments that correlate with the levels set by the vector analyser. For example, how does the vector analyser implement 1% of carrier leakthrough?

The only certain way of determining how the vector signal generator interpreted and added the quadrature errors was to study a detailed schematic of the unit. Unfortunately, detailed informating regarding the internal operations of the vector analyser was not available. This point is especially true for the addition of carrier leakthrough error. This restriction, however, does not limit the possibility of using alternate methods to estimate the values of the quadrature mixer impairments needed in the emulator. By observing the spectral output of the hardware setup in figure 5.6, it is possible to use the SFDR level between the fundamental and carrier component to determine a carrier leakthrough value. The process of obtaining the value that will represent 1% carrier leakthrough in the emulator was performed simply as follows: Working with equation 5.2, the SFDR level of 24.33 dB for the fundamental and carrier leakthrough component was used to calculate a value of $2.5 \cdot 10^{-4}$ for the leakthrough level. Subtracting this level by the value obtained from the carrier leakthrough output with no impairments (figure 5.3), yields a resulting error of $1.81 \cdot 10^{-4}$. This resulting error value was used in the quadrature mixer

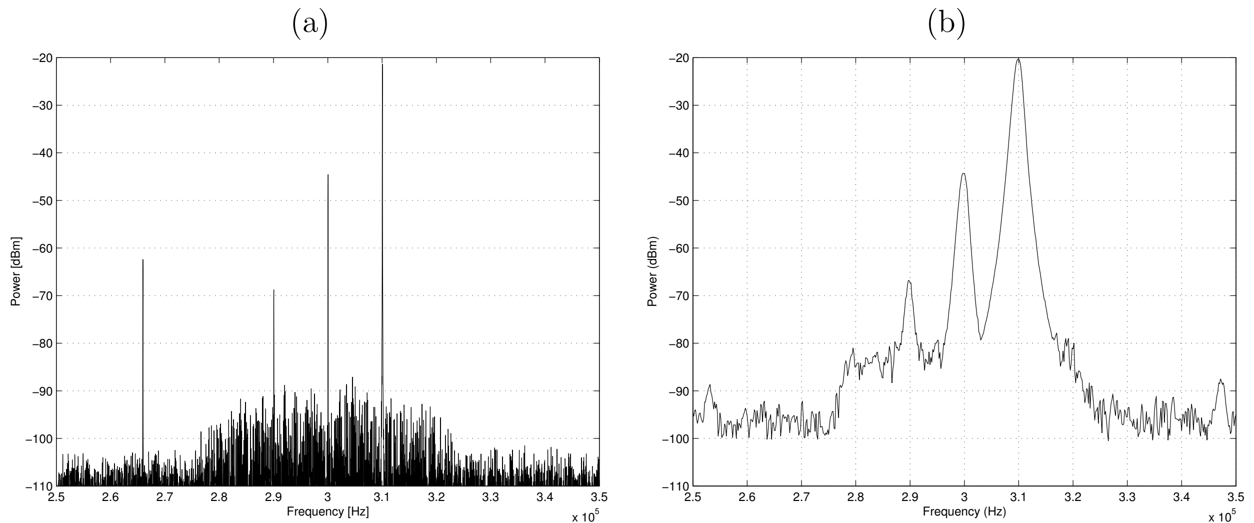


Figure 5.18: *Output of the emulator's quadrature mixer with 1% carrier leakthrough (a). The original output of the physical hardware system setup with the same impairment added (b) is included for comparison.*

model to represent 1% carrier leakthrough error. Figure 5.18 shows the spectral output of the quadrature mixer with 1% carrier leakthrough added. The results are figure 5.18 are summaried in table 5.2.

The differences of 1 dB and greater can be attributed to both, the amplitude error introduced by the upsampling model as well as taking the limitations of the FFT into account. In general, however, the outcome of adding 1% carrier leakthrough in the emulator matches up closely to the hardware measurents. The measurements for 1% amplitude imbalance are investigated next.

Table 5.2: *Comparison of hardware and emulation measurements with 1% carrier leakthrough added*

Measurement	**Hardware**	**Emulation**	**Difference**
Fundamental	−20.34 dBm	−21.34 dBm	1 dB
Carrier peak	−44.33 dBm	−44.55 dBm	0.22 dB
Amp_dev spur	−66.83 dBm	−68.77 dBm	1.94 dB
Carrier SFDR	24.33 dB	23.21 dB	1.12 dB
Amp_dev SFDR	46.49 dB	47.43 dB	0.94 dB

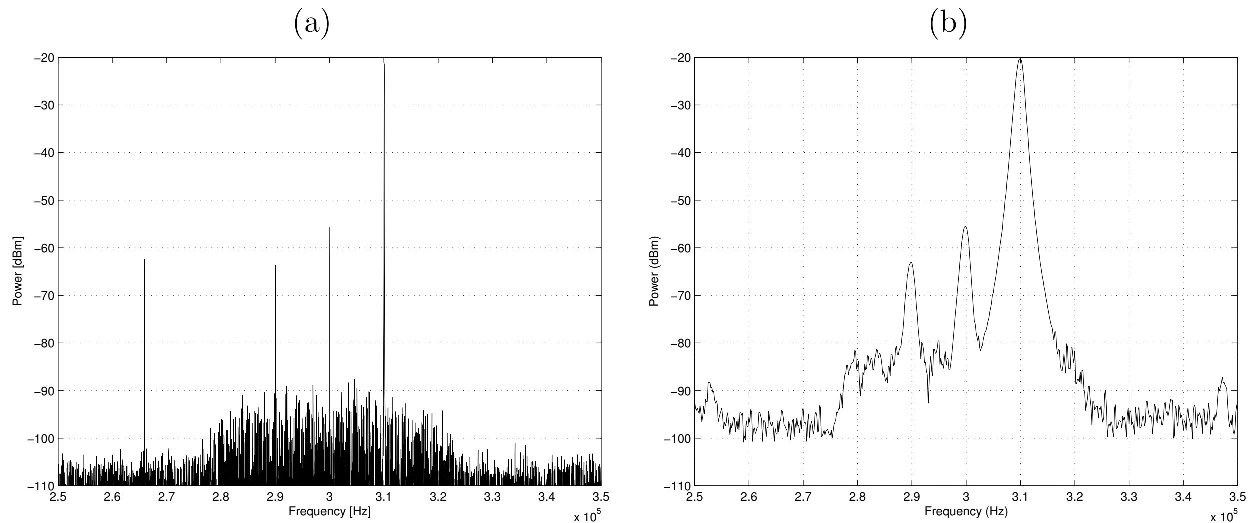


Figure 5.19: *Output of the quadrature mixer model (a), showing the effects of 1% amplitude imbalance by a increase in the power level of the spurious component at 290 kHz. The original output of the physical hardware system setup with the same impairment added (b) is included for comparison.*

Addition of amplitude deviation error

Applying the same method used to obtain the carrier leakthrough level needed, acquiring the amplitude deviation value from figure 5.7 yielded a result of 0.007, which translates to 0.7% — approaching the expected value of 1%. This indicates that the vector analyser utilises a similar method as the quadrature mixer model in order to add amplitude error to the signal. The emulator simply does this by adjusting the level of the I-channel's carrier waveform amplitude by a specified error level. With all other quadrature impairments set to zero again, figure 5.19 shows the output of the quadrature mixer with 1% amplitude deviation error added. Table 5.3 summaries the absolute and relative signal power levels of figure 5.19. Generally, the difference error values in table 5.3 indicate that hardware and emulated setup measurements show only minor level discrepencies. However, the cause of a 0.81 dB difference between the SFDR of the amplitude deviation spur and fundamental component could be as a result of the original level error of the fundamental component.

Addition of phase error

Using figure 5.20, the final quadrature mixing impairment that will be looked at is the addition of 3° phase error. Again, for the setup of the quadrature mixer, all other impairments were set to zero. Table 5.4 once again summaries the main observations in figure 5.20.

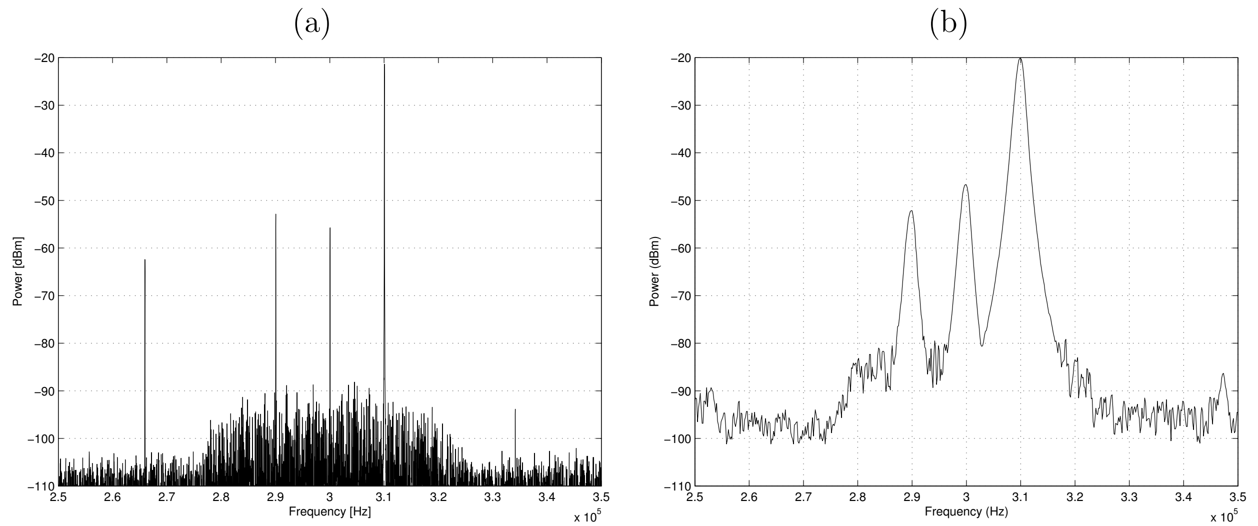


Figure 5.20: *Output of the quadrature mixer model (a), showing the effects of 3° of phase imbalance by an increase in the power level of the spurious component at 290 kHz. The original output of the physical hardware system setup with the same impairment added (b) is included for comparison.*

The hardware measurement outcomes in table 5.4 was obtained from figure 5.8. Some of the results from table 5.4 warrant further explanation concerning the difference error levels of 9.07 dB and 8.08 dB for the carrier leakthrough component. The reason for this large difference is because vector analyser measurement actually shows that the carrier leakthrough component decreased for the addition of 3° phase error — theoretically something that should not happen.

As mentioned earlier, a likely explanation to why this happens is the possibility of the vector analyser having some type of internal compensation circuitry that attempts to keep the total power of the output signal constant. This is, however, at best just a speculation, and a more thorough explanation can only be given if a schematic detailing

Table 5.3: *Comparison of hardware and emulation measurements with 1% amplitude imbalance added*

Measurement	Hardware	Emulation	Difference
Fundamental	−20.34 dBm	−21.32 dBm	0.98 dB
Carrier peak	−55.5 dBm	−55.67 dBm	0.17 dB
Amp_dev spur	−63 dBm	−63.7 dBm	0.7 dB
Carrier SFDR	35.16 dB	34.35 dB	0.81 dB
Amp_dev SFDR	42.66 dB	42.38 dB	0.28 dB

the internal operations of the vector analyser is used.

The emulator results indicate that the carrier component remained unchanged, which is what is theoretically expected. This is also a good example of precisely why emulation is such a valuable tool — the designer has the capabilities to see the effects that individual quadrature inaccuracies have on a signal without having to worry about interfering signals affecting the measurements.

Combination of quadrature impairments

The last observation that will be made is for the combination of all three quadrature impairments as shown in figure 5.21. The results of table 5.5 for figure 5.21 indicate the expected: an increase in level for the carrier leakthrough and amplitude imbalance spur. Again, the results for the carrier leakthrough SFDR does not directly correlate with the measurements of the hardware setup. This effect was observed in figure 5.20 showed up once again with the combination of all the quadrature impairments.

5.3 Conclusions and summary of results

An interesting aspect of using the particular hardware setup in chapter 5, is that in real life systems it is not always possible to measure the data at every point in the system. Therefore, it was necessary to work backwards from results of the spectrum analyser measurements. Of course, this will at best, only approximate the system since the more you know about the system, the more accurate the results will be. This is, however, the very purpose of the emulator — to be able to build and test a system without having to completely represent the internal makeup of the system.

The results of the comparison between hardware and emulated measurements yielded interesting results. Apart from the amplitude error generated by the upsampling model, the levels in the emulation corresponded very closely to the equivalent hardware setup

Table 5.4: *Comparison of hardware and emulation measurements with 3° phase imbalance added*

Measurement	**Hardware**	**Emulation**	**Difference**
Fundamental	−20.34 dBm	−21.35 dBm	1.01 dB
Carrier peak	−46.66 dBm	−55.73 dBm	9.07 dB
Amp_dev spur	−52.16 dBm	−52.87 dBm	0.71 dB
Carrier SFDR	26.32 dB	34.4 dB	8.08 dB
Amp_dev SFDR	31.82 dB	31.52 dB	0.3 dB

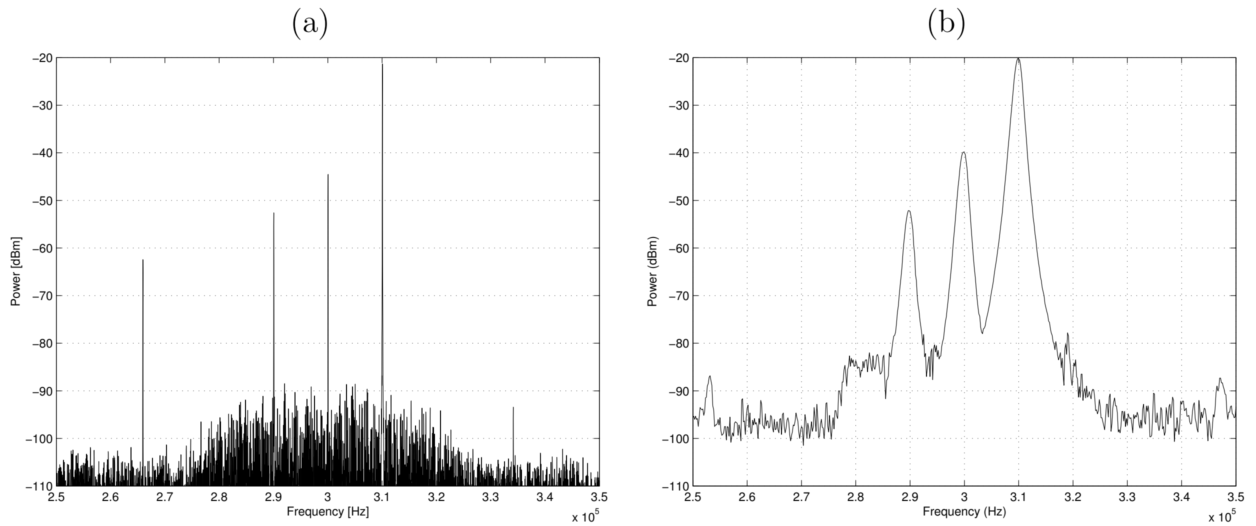


Figure 5.21: *Output of the quadrature mixer model, showing the combined effects of all three quadrature impairments. The original output of the physical hardware system setup with all the same three impairments added (b) is included for comparison.*

output levels. Although the output levels are important in terms of assessing the performance of the algorithms used in the models, another outcome is even more important. This is confirmation that the architecture of the emulator is operating correctly in terms of:

- The setting up and linking of various components.
- The retrieval of waveform data for post-processing.
- The correct streaming and buffer control of samples in the emulated system.

Although the basic operation of the hardware emulator was demonstrated to operate as expected, there is still much room for improvement. Chapter 6 to follow will look at

Table 5.5: *Comparison of hardware and emulation measurements with the combined effect of all three quadrature impairments*

Measurement	Hardware	Emulation	Difference
Fundamental	−20.34 dBm	−21.32 dBm	0.98 dB
Carrier peak	−39.8 dBm	−44.53 dBm	4.73 dB
Amp_dev spur	−52.16 dBm	−52.6 dBm	0.44 dB
Carrier SFDR	19.46 dB	22.93 dB	3.47 dB
Amp_dev SFDR	31.82 dB	31 dB	0.82 dB

some of the possibilities.

Chapter 6

Conclusion and recommendations

6.1 Background and Aim

The many advantages of SDR (as highlighted in chapter 2) make SDR technology an attractive alternative to its hardware-based radio counterpart. The drawback however mentioned was the ever present non-standardised hardware front-end, since extensive RF hardware is often needed in the SDR analogue front-end. The power of software testability is one of the key advantages in a SDR because it allows the designer the freedom to rapidly implement and test new conceptualised ideas and algorithms. The primary aim of this thesis is to take this property of software testability over to the hardware front-end of a SDR. This became the basis for the conception and development of the SDR hardware front-end emulator testbed for a SDR.

This thesis has focused on both the theoretical design and practical implementation of a hardware emulator testbed. The focus has been placed on developing a software architecture that supports a high degree of interchangeability amount the modeled components. Only hardware components present in a typical SDR RF front-end were selected for prototyping and modeling. This was performed by:

- Studying the background theory of the non-ideal behaviours for the selected components.
- Considering and selecting the optimal way to model the components and their non-idealities.
- The development of MATLAB prototype models in order to verify the algorithms used for the modeled components.

In order for the emulator architecture to be functionally effective, it was designed in C++ using a technique to allow a high degree of re-use and interchangeability amongst modeled components. The principle idea behind the technique was to make use of the idea of a generic "converter" object that can receive, process (in its own well-defined way)

and deliver a stream of samples. These samples can be analysed at various points in the emulated system using data visualisation software, which was MATLAB in this case.

6.2 Achievements of thesis

Comparative test outcomes from Chapter 5 reveal promising results that indicate a relatively high degree of accuracy for the sample emulated system. The tests are by no means an exhaustive analysis of the emulator but serve to illustrate the basic functionality of the hardware emulator testbed for a given sample system. The results of the first version of the emulator therefore conclude that the modeled components can be easily linked in order to collectively operate so as to form a basic emulated system. The accuracy of the resulting emulated waveforms are thus chiefly depended on the precision of the component models. The development, however, of more elaborate and complex models are left as a suggestion for further work and investigation. The objective of developing a hardware emulator testbed for a SDR was further divided into many sub-problems. These sub-problems formed the objectives set in Chapter 1, and include the following:

6.2.1 The development of an abstract base class

Chapter 4 introduced the concept of the abstract base class called "Converter". The Converter base class defines a generic component in that the Converter object works by receiving samples, processing the samples and then delivering the samples out. The exact processing that get performed is determined by the descendant classes implementation of the processing function. This high level design readily facilitates the addition of new component models.

6.2.2 The development of sample modules

Before the sample modules were developed, the necessary background theory was studied in order to gain an understanding of the real-world behaviour for the identified hardware components. Thereafter, MATLAB prototype models were developed in order to verify the algorithms used in the models. Once the operation of the MATLAB prototype models were verified, the equivalent C++ models were developed for inclusion in the hardware emulator architecture.

6.2.3 Software architecture development

Chapter 4 was devoted to meeting this objective. All the components in the emulator architecture are initialised as a special type of pointer — a reference counted pointer

(RCPtr). The components are contained within a single common vector. This makes the components easily interchangeable since standard vector operations can be applied. The use of the RCPtr object enables the successful linking of components in the emulator with minimal complexity. A text-based front-end application was developed in order to facilitate the design and testing of the sample emulated system. It should be noted that the system architecture developed here is not only applicable to hardware emulation — this is because the software side of a SDR can be seen as a form of hardware emulation since traditional analogue signal processing is being performed in software. Therefore, many of the architectural concepts presented by the emulator can be applied to the software side of a SDR. An example of this would be the inclusion of a "modulator" or "demodulator" model as a descendant of the converter base class.

6.2.4 Verification using a physical setup

The physical hardware setup in chapter 5 was used as a bench-mark to determine the performance of the emulator. The setup comprised of equipment that was readily available, such as the computer's sound card and a vector signal generator.

6.2.5 Post-processing and visualisation tools

In order to acquire the waveform data in the emulator, data files known as monitor files were employed. MATLAB was used as the post-processing tool of choice. The data files were loaded into the MATLAB workspace and a script file was written in order to visualise the data in frequency domain.

6.3 Application of the results

The results obtained by this research present the following applications:

- The idea of including the front-end hardware into the SDR architecture on the same software platform is a new paradigm. This promises more rapid and reliable communications system design. It therefore brings the larger part of the development process into the digital domain, thereby making an entire SDR system software testable.

- Certain of the basic design principles on which the hardware emulator's architecture is based, serve as an important foundation for the development of the one of the undertakings for the overall SDR project — the development of the primary SDR architecture.

- The hardware emulator provides a simple platform on which to design and test complex model algorithms. This is because, due to the ease in which new models can be accommodated in the emulator, the designer can concentrate more on algorithm development and less on compatibility issues.

- One of the primary goals of SDR in general is to bring the software side of a SDR as close to the antenna as possible, thereby minimising the amount of analogue hardware used. One of the key features of utilising the software-based hardware emulator, is that it allows the designer to freely experiment with the position of the bridge that partitions the digital and analogue domain of a SDR.

6.4 Future work

The development of software-based hardware emulator for a SDR is a fairly novel idea and therefore still a somewhat unexplored area of research. The possibility of a completed, fully-functional SDR hardware emulator that readily integrates and forms part of a SDR software library is a very exciting prospect. The work done in this thesis has formed the initial developments required to obtaining this goal. There is, therefore, still much research to be done and some suggestions for potential further work is outlined below:

- The various possibles must be looked at on how best to integrate the hardware emulator to the remaining SDR. This can be done by either extending the current emulator's library to include SDR software components or to look at the required modifications needed in order to integrate it to existing SDR systems.

- Currently, the interface used for the front-end application is text-based. The development of a GUI (graphical user interface) would certainly make operating the hardware emulator more user-friendly. Similarly, a GUI for MATLAB could also be made to facilitate the operation of post-processing the waveform data.

- The library of emulated components can be extended and upgraded by studying current SDR hardware front-end design trends. The current models can also be improved upon by placing them within a software wrapper in order to make utilising the models more user-friendly.

The research conducted in this thesis has shown the value of hardware emulation in a software-defined radio. However, in order for this research to prove more worthwhile, still much work is needed in this rather untapped area of SDR development. Although the design and development of a SDR presents many challenges, further research in hardware emulation can make the hardware emulator an important part of SDR design.

6.5 Publications in connection with thesis

WITKOWSKY, J. and VAN ROOYEN, G-J., "'A hardware emulator testbed for software defined radio."' *Proceedings of IEEE Africon 2002*, October 2002, Vol. 1, No. 6, pp. 383-388.

Bibliography

AHLQUIST, G. 1999: Error Control Coding in Software Radios: An FPGA Approach, IEEE Personal Communications.

ARNOTT, R. *et al.*, 1998: Advanced Base Station Technology, IEEE Journal on selected areas of Communication.

BOSE, V. *et al.*, 1999: Virtual Radios, IEEE Journal on selected areas of Communication, Vol.17, no.4, 591–602.

BURACCHINI, E. 2000: The Software Radio Concept, IEEE Communications Magazine .

CANDY, J. C. & TEMES, G. C. 1992, Oversampling Methods for A/D and D/A Conversion, IEEE Press, Vol.10, no.5, 1–29.

CERNA, M. & HARVEY, F. 2000: The Fundamentals of FFT-Based Signal Analysis and Measurement.
URL: *http://www.science.unitn.it/ bassi/Signal/NInotes/an041.pdf*

CHEN, W. K. 1995: The Circuits and Filters Handbook, CRC Press, England.

COUCH II, L. 1995: Modern Communication Systems, Prentice Hall, Englewood Cliffs.

CROOK, D. & CUSHING, R. 1998: Sources of Spurious Components in a DDS/DAC System, RF Design pp. 28–42.

DU PREEZ, J. 2001: Reference counted pointers (RCPtrs) header file, University of Stellenbosch.

ECKEL, B. 2000: Thinking in C++, 1st edition, Prenctice Hall, New Jersey.

EGAN, W. F. 2000: Frequency Synthesis by Phase Lock, 2nd edition, Wiley-Interscience, New York.

EICHENBERGER, C. & GUGGENBUHL,W. 1991: Charge injection of analogue CMOS switches, IEE Proceedings G (Circuits, Devices and Systems, Vol.138, no.2,pp. 155–159.

FARIA, D. DUNLEAVY, L. & SVENSEN, T. 1995: The Use of Intermodulation Tables for Mixer Simulations, Microwave Journal .

GAMMA, E. *et al.*, 1995: Design Patterns: Elements of reusable object-orientated software, Addison-Wesley, Massachusetts.

GARCIA, J. & LAJEUNESSE, G. 1995, Understanding Glitch in a High Speed D/A converter, Intersil Technical Brief .

GUNN, E. *et al.*, 1999: A Low-Power DSP Core-based Software Radio Architecture, IEEE Journal on selected areas of Communication, Vol.17, no.4,pp. 574–590.

HJORTH, M. & HVITTFEDT, B. 2002: Modelling an RF Converter in Matlab, Master's thesis, Linkoping University Sweden.

HOROWITZ, P. & HILL, W. 1989: The art of electronics, 2nd edition, Cambridge University Press, Cambridge.

HOSKING, R. 1998: Digital Receiver Handbook, 2nd edition, Pentek, New Jersey.

KENINGTON, P. B. 2000: High-linearity RF amplifier design, Artech House, Norwood.

KESTER, W. 1997: Undersampling applications.
URL: *http://www.analog.com/UploadedFiles/Associated_Docs/213399675.pdf*

KROUPA, V. F. 1973: Frequency Synthesis - Theory, Design & Applications, Griffin, London.

KUC, R. 1988: Introduction to digital signal processing, 1st edition, McGraw-Hill, New York.

LACKEY, R. J. & UPMAL, D. W. 1995, Speakeasy: The Military Software Radio, IEEE Communications Magazine .

MALOBERTI, F. et al 1992: Behavioral modeling and simulations of data converters.
URL: *www.mathsworks.com/products/dsp_comm/pdfs/Behavioural.pdf*

MILLER, G. 2001: Modern Electronic Communication, 5th edition, Prentice-Hall, Englewood Cliffs.

MITOLA III, J. 1995: The Software Radio Architecture, IEEE Communications Magazine .

MITOLA III, J. 1999*a*: Software Radio Architecture: A Mathematical Perspective, IEEE Journal on selected areas in communications, Vol.17, no.4, pp. 514–538.

MITOLA III, J. 1999*b*: Technical Challenges in the Globalization of Software Radio, IEEE Communications Magazine, Vol.37, pp. 84–89.

MITOLA III, J. 2000: Software Radio Architecture Evolution: Foundations, Technology Tradeoffs, and Architecture Implications, IEEE Journal on selected areas of Communication, Vol.83, no.6, pp. 1165–1173.

PRESS, H. *et al.*, 1992: Numerical Recipes in C: The Art of Scientific Computing, 2nd edition, Cambridge University Press, Cambridge.

PROAKIS, J. G. & MANOLAKIS, D. G. 1996: Digital Signal Processing: Principles, Algorithms, and Applications, 3rd edition, Prentice-Hall, New Jersey.

Pucker, L. 2001: Paving paths to Software Radio Design.
URL: *http://www.csdmag.com/story/OEG20010521S0118*

REED, J. 2002: Software Radio: A Modern Approach to Radio Engineering, 1st edition, Prentice-Hall, Englewood Cliffs.

REICHHART, P. *et al.*, 1999: The Software Radio Development System, IEEE Journal on selected areas of Communication.

RHODE, U. & WHITAKER, J. 2001: Communication Receivers - DSP, Software radios and design, 2nd edition, McGraw-Hill, New York.

RHO 2003: SMBIQ 04B User's Manual.

SALKINTZIS, K. *et al.*, 1999: ADC and DSP challenges in the Development of Software Radio Base Stations, IEEE Journal on selected areas of Communication .

SDRF 2002: Software Defined Radio Forum.
URL: *www.sdrforum.org*

SELIC, B. *et al.*, 1998: Real-Time Object-Oriented Modeling, 1st edition, Wiley, New York.

Texas Instruments 1995: Understanding Data Converters.
URL: *www.hit.bme.hu/people/papay/edu/Acrobat/Dataconv.pdf*

THEDE, L. 1996: Analog and Digital Filter Design using C, 1st edition, Prentice Hall, New Jersey.

TURLETTI, E. & TENNENHOUSE, D. 1999: Complexity of a Software GSM Base Station, IEEE Journal on selected areas of Communication.

VAN ROOYEN, G.-J. 2000: An Analysis of Quadrature-Baseband Direct Digital Synthesis, Master's thesis, University of Stellenbosch.

VAN ROOYEN, G.-J. 2003: Quadrature-Baseband Compensation Principles for Arbitrary-Accuracy Signal Conversion and Processing, PhD thesis, University of Stellenbosch.

WALDEN, H. 1999*a*: Analog-to-Digital Converter Survey and Analysis, IEEE Journal on selected areas in communications, Vol.17, no.4, pp. 539–549.

WALDEN, R. H. 1999*b*: Performance Trends for Analogue-to-Digital Converters, IEEE Journal on selected areas of Communication.

WEISSTEIN, E.W 2003: Cramer's Rule.
URL: *http://mathworld.wolfram.com/CramersRule.html*

WITKOWSKY, J. & VAN ROOYEN, G.-J. 2002: A hardware emulator testbed for software defined radio, Proceedings of IEEE Africon 2002, Vol.1, no.6, pp. 383–388.

ZANGI, K. & KOILPILLAI, R. 1999: Software Radio Issues in Cellular Base Stations, IEEE Journal on selected areas of Communication, Vol.17, no.4, pp. 561–573.

ZIEMER, R. & TRANTER, W. 1995: Principles of Communications - Systems, Modulation and Noise, Wiley, New York.

Appendix A

Source code

A.1 MATLAB simulation models

The prototype models were designed and tested in MATLAB, in order to determine the models operation prior to developing the C++ models.

A.1.1 Data converter simulation models

- The model `quantisation.m` is used to quantise a signal according to a specified number of bits.
- The INL pre-distortion model, `inl.m` is used in order to produce the typical non-linear conversion transfer function found in practical data converters.
- The concept of upsampling is important for changing the effective sampling rate between devices. Upsampling a signal also ensures that momentary effects in a DAC, such as glitch impulse can be observed. The `upsample.m` model performs this function.
- The `glitch.m` model is used to add glitch impulse to the upsampled signal.
- In order for the glitch impulse effects to seem random, a deterministic function, `hash.m`, is employed.

A.1.2 Amplifier simulation models

- The saturation model, `saturate.m`, clipps the input signal as the specified limits.
- The Taylor amplifier model, `Taylor.m`, is based of the I and Q channel Taylor coeffients that describe the nonlinear behaviour of the amplifier.

A.1.3 Mixer simulation models

- The model `mixer.m` is used to model the nonideal mixing process using Taylor amplifiers. The coeffients of the Taylor amplifier can be calculated by knowing what the values of the IMT are that describe a particular mixer.

- I and Q signals can be quadrature mixed in the `qmixer.m` model which also allows the user to add quadrature impairments.

A.1.4 Filter simulation models

- The function `poly2quads.m` is used to break a high order polynomial into second order polynomials building blocks. These second order polynomials are used directly in the quadratic filter.

- Using the second order polynomials descibed above, the `quadratic_filter.m` model performs filtering on the input signal.

A.1.5 Post-processing routines

- The samples obtained from the monitor files describe the time-domain signal. Visualisation of this data is simply performed by plotting the samples in MATLAB. The function `freqplot.m` is required to observe and analyse the frequency content of the signal of interest.

- The routine `spur.m` was employed in order to accuratly measure the level of the spurious component in frequency plots (van Rooyen 2000).

```
function y = quantisation(x,n)
% quantisation   Performs n-bit quatisation
%                Syntax: y = quantisation(x,n)
%                   x is the input signal to be quantised according to the
%                   number of bits, n. The input signal must be in the range
%                   of -1 <= x <= 1 for the code to work correctly. The
%                   quantised output is symmetrical around but does not
%                   include zero.
%                See also: inl.m, upsample.m, glitch.m

if (nargin < 2)
    error('At least two parameters are required');
end;

if (max(x) > 1 | max(-x) < -1)
    error('Input must be equal to or in the range +1 and -1');
end;

if ((n/round(n)) ~= 1)
    error('Number of bits must be a integer value');
end;

q = (2^n);
y = sign(x).*(ceil(abs(x)*(q-1)/(2) - 1 + eps) + 0.5) / (q-1) * 2;
```

```
function INLOut = inl(IN,x1,y1)
% INLOut    Model to introduce INL predistortion
%           Syntax: INLOut = INl(IN,x1,y1)
%               x1 and x2 are vectors containing the normalised
%               transfer function co-ordinates that specify the
%               shape of the INL curve. The vector IN contains
%               the input signal that is predistorted using
%               MATLAB's cubic spline function. The function also
%               plots the distorted output over the ideal transfer
%               function curve.
%           See also: quantisation.m, glitch.m, upsample.m

if (nargin < 3)
    error('At least three parameters are required');
end;

if (length(x1) ~= length(y1))
    error('The length of vectors x1 and x2 must be equal');
end;

cs = spline(x1,y1);
INLOut = ppval(cs,IN);

xx = linspace(-max(x1),max(x1),200);
value = ppval(cs,xx);

plot(x1,y1,'o',xx,value,'-',[-max(x1) max(x1)],[-max(x1) max(x1)]);
title('INL Predistortion curve vs Ideal output');
xlabel('Input signal level');
ylabel('Output signal level');
grid on;
```

```
function [us] = upsample(x, RATE)
% upsample      Function used to increase the effective sample rate
%               Syntax: us = upsample(x, RATE)
%                   The upsample function simply takes the input signal
%                   'x' and repeats it by the integer number specfied by
%                   'RATE'. The upsampled signal sampling rate, 'us',
%                    becomes the product of original sampling rate and
%                    the value specified by 'RATE'.
%               See also: glitch.m

if (ceil(RATE)/RATE ~= 1)
    error('The value of the RATE must be an integer number');
end;

if (nargin < 2)
    error('At least two parameters are required');
end;

us = kron(x,ones(1,RATE));
```

```
function [y] = glitch(x, seed, USR, bits, p)
% glitch    Part of the DAC model that generates glitch impulse
%           Syntax: y = glitch(x, seed, USR, bits, p)
%               The input signal x must be an upsampled signal
%               specified by the integer value USR. The seed
%               parameters can be adjusted to change the values
%               generated by the hash function. The value p is a
%               parameter that can be used to specifiy the intesisty
%               of the glitch impulse so that when p = 100 means that
%               there will be 100% glitch impulse on the output.
%           See also: upsample.m

h_intensity = p/100;

q = 2^bits - 1;

xx = round(x*q);

last = 0;
width_counter = 1;

for n = 1:length(x)
    if width_counter ~= 1;
        y(n) = current_out;
        width_counter = width_counter - 1;

    elseif x(n) ~= last;
        [w(n), h(n)] = hash(last, xx(n), seed);

        if w(n) == 0
            w(n) = 0.1;
        end

        temp = last;
        width = w(n);
        width_counter = ceil((width.*USR)/2);
        last = x(n);
```

```
        y(n) = x(n) + h(n).*(h_intensity);
        current_out = y(n);

    else
          w(n) = 0;
          h(n) = 0;
          y(n) = x(n);
          current_out = y(n);
    end

end
```

```
function [w, h] = hash(last, current, seed)
% hash      Hashing function used with glitch model
%           Syntax: [w, h] = hash(last, current, seed)
%               The hashing function generates a height
%               and width value for the glitch impulse
%               model. The height value ranges from -0.5
%               to +0.5. The width value goes from 0 to
%               1. The height and width values depend
%               on the values of last and current
%               quantisation value, as well as a seed
%               value. The hash function is deterministic
%               in that it will produce the exact same
%               output values for the same values of last,
%               current and seed.
%           See also: glitch.m

if (nargin < 3)
    error('At least three input parameters are required');
end;

w = mod(round(last^15 + current^7 + seed^5), 100) / 100;
h = ((mod(round(last^13 + current^5 + seed^3), 100) / 100) - 0.5)*2;
```

```
function [y] = saturate(x,Vsat)

% saturate       Saturates the signal x at the level specified by Vsat
%                Syntax: y = saturate(x,Vsat)
%                    The input signal x is saturated corresponding to the
%                    (equal postive and negative) saturation levels set
%                    buy the value of Vsat.
%                See also: Taylor.m

if (nargin < 2)
    error('At least two parameters are required');
end;

y = [];

for i = 1:length(x);

    if x(i) > Vsat
        y(i) = Vsat;

    elseif x(i) < -Vsat
        y(i) = -Vsat;

    else
        y(i) = x(i);

    end

end
```

```
function Y = Taylor(X,I,Q)
% Taylor    Performs quadrature Taylor series amplification
%           Syntax: Y = Taylor(X,I,Q)
%               The vectors I and Q hold the coefficients for the
%               I and Q channel Taylor series respectively. Given
%               both the I and Q Taylor coeffiencts, both the
%               amplitude and phase response of an amplifier can
%               be obtained. The I and Q vectors must be of equal
%               length. X is the vector that contains the input
%               signal to be amplified.
%           Syntax: Y = Taylor(X,I)
%               If only the amplitude response of a given amplifier
%               is required, then only the I Taylor series coefficeints
%               are needed.
%           See also: saturation.m

if (nargin < 2)
    error('At least two parameters are required');
end;

if (nargin < 3)
    Q = zeros(1,length(I));
end;

if (nargin == 3)
    if (length(I) ~= length(Q))
        error('Parameters I and Q must have the same length');
    end
end

Vin = hilbert(X);

n = length(I);
k = length(X);

V_I = zeros(1,k);
V_Q = zeros(1,k);
```

```
Itemp = 0;
Qtemp = 0;

for i = 1:length(X)

    for j = 1:n

        Itemp = I(j).*(real(Vin(i)).^(j-1));
        V_I(i) = V_I(i) + Itemp;

        Qtemp = Q(j).*(imag(Vin(i)).^(j-1));
        V_Q(i) = V_Q(i) + Qtemp;

    end;

end;

Y = V_I + V_Q;
```

```
% mixer_script.m
% Script used to illustrate how a mixer is modelled from its IMT.
% The IMT is decomposed using SVD and then the remaining matrices
% are modelled as the product of Taylor series pairs. See section
% 3.6.7 in Chapter 3 of the thesis for an in-depth discussion of
% this method.

to model 4th order mixer from IMT

A = [99 39 42 46; 25 0 39 11; 68 67 76 67; 63 58 65 60]; % IMT matrix

AA = 10.^(-A/10);   % convert from dBm to absolute value

AA = AA/AA(2,2);    % scale values with reference to fundamental signal amplitude
                    % i.e. values are now a expressed as a ratio of the fundamental

[U,S,V] = svd(AA);  % Singular value decomposition

v1 = U(:,1)*S(1,1); % Extract harmonics in first row of U matrix. Multiply by
                    % first eigenvalue
v2 = V(:,1);        % Extract harmonics in first row of V matrix

% Assign harmonic values
h0 = v1(1)/2
h1 = v1(2)
h2 = v1(3)
h3 = v1(4)

m0 = v2(1)/2
m1 = v2(2)
m2 = v2(3)
m3 = v2(4)

% Transform harmonic amplitudes into Taylor coefficients
% These formulas are the resulting relationships obtained
% from the written calculations
a0 = h0 - h2       % dc component
a1 = h1 - 3*h3     % 2nd Taylor coefficient
```

```
a2 = 2*h2           % 3rd Taylor coefficient
a3 = 4*h3           % 4th Taylor coefficient

% transform harmonic amplitudes into second Taylor coefficients
b0 = m0 - m2        % dc component
b1 = m1 - 3*m3      % 2nd Taylor coefficient
b2 = 2*m2           % 3rd Taylor coefficient
b3 = 4*m3           % 4th Taylor coefficient

%-------- Input waveform -----------
% generate input (IF) waveform
f = 25;
fs = 5e3;
Ts = 1/fs;
n = (1:3000)*Ts;
A = -50;             % dBm value specified by IMT data

% Convert dBm to a level
Power = 10^((A-30)/10)
x = Power*cos(2*pi*f.*n);

% generate LO waveform
fLO = 500;
B = 7;               % dBm value

% Convert dBm to a level
NPower = 10^((B-30)/10)
x2 = NPower*cos(2*pi*fLO.*n);

% --------- Perform mixing ------------
% calculate Taylor 1 output
x = x/Power;
y1 = a0 + a1*x + a2*x.^2 + a3*x.^3;

% calculate Taylor 2 output
x2 = x2/NPower;
y2 = b0 + b1*x2 + b2*x2.^2 + b3*x2.^3;
```

```
% multiple the outputs of the Taylor series
y = y1.*y2;

%===================================
%--- Second mixing segment
v3 = U(:,2)*S(2,2); % harmonics of first vector
v4 = V(:,2);        % harmonics of second vector

h0 = v3(1)/2;
h1 = v3(2);
h2 = v3(3);
h3 = v3(4);

m0 = v4(1)/2;
m1 = v4(2);
m2 = v4(3);
m3 = v4(4);

% Transform harmonic amplitudes into Taylor coefficients
a0 = h0 - h2;   % dc component
a1 = h1 - 3*h3; % 2nd Taylor coefficient
a2 = 2*h2;      % 3rd Taylor coefficient
a3 = 4*h3;      % 4th Taylor coefficient

% Transform harmonic amplitudes into Taylor 2 coefficients
b0 = m0 - m2;   % dc component
b1 = m1 - 3*m3; % 2nd Taylor coefficient
b2 = 2*m2;      % 3rd Taylor coefficient
b3 = 4*m3;      % 4th Taylor coefficient
%---------

% Calculate Taylor 1 output
y3 = a0 + a1*x + a2*x.^2 + a3*x.^3;

% Calculate Taylor 2 output
y4 = b0 + b1*x2 + b2*x2.^2 + b3*x2.^3;

yy = y3.*y4;    % multiple the outputs
```

```
%==============================================================
% 3rd mixing segment
v5 = U(:,3)*S(3,3); % harmonics of first vector
v6 = V(:,3);        % harmonics of second vector

h0 = v5(1)/2;
h1 = v5(2);
h2 = v5(3);
h3 = v5(4);

m0 = v6(1)/2;
m1 = v6(2);
m2 = v6(3);
m3 = v6(4);

% Transform harmonic amplitudes into Taylor coefficients
a0 = h0 - h2;   % dc component
a1 = h1 - 3*h3; % 2nd Taylor coefficient
a2 = 2*h2;      % 3rd Taylor coefficient
a3 = 4*h3;      % 4th Taylor coefficient

% Transform harmonic amplitudes into Taylor 2 coefficients
b0 = m0 - m2;   % dc component
b1 = m1 - 3*m3; % 2nd Taylor coefficient
b2 = 2*m2;      % 3rd Taylor coefficient
b3 = 4*m3;      % 4th Taylor coefficient

%---------
% Calculate Taylor 1 output
y5 = a0 + a1*x + a2*x.^2 + a3*x.^3;

% Calculate Taylor 2 output
y6 = b0 + b1*x2 + b2*x2.^2 + b3*x2.^3;

yyy = y5.*y6; % multiple the outputs

%==============================================================
%--- 4th (final) mixing segment
```

```
v7 = U(:,4)*S(4,4); % harmonics of first vector
v8 = V(:,4);        % harmonics of second vector

h0 = v7(1)/2;
h1 = v7(2);
h2 = v7(3);
h3 = v7(4);

m0 = v8(1)/2;
m1 = v8(2);
m2 = v8(3);
m3 = v8(4);

% Transform harmonic amplitudes into Taylor coefficients
a0 = h0 - h2;   % dc component
a1 = h1 - 3*h3; % 2nd Taylor coefficient
a2 = 2*h2;      % 3rd Taylor coefficient
a3 = 4*h3;      % 4th Taylor coefficient

% Transform harmonic amplitudes into taylor 2 coefficients
b0 = m0 - m2;   % dc component
b1 = m1 - 3*m3; % 2nd Taylor coefficient
b2 = 2*m2;      % 3rd Taylor coefficient
b3 = 4*m3;      % 4th Taylor coefficient
%---------

% Calculate Taylor 1 output
y7 = a0 + a1*x + a2*x.^2 + a3*x.^3;

% Calculate Taylor 2 output
y8 = b0 + b1*x2 + b2*x2.^2 + b3*x2.^3;

yyyy = y7.*y8; % Multiple the outputs

% Sum the outputs of all the Taylor product pairs
y_out = y + yy + yyy + yyyy;

%================================================
```

```
% Frequency plot of mixed signal
[a,b] = freqplot(y_out, fs, 1);

k = max(b);             % set 0dB reference
b = 10*log10(b/k);      % dB plot

plot(a,b)

% Condition output figure axis
XMIN = 0;
XMAX = fs/2;
YMIN = -80;
YMAX = 10;
axis([XMIN XMAX YMIN YMAX]);
ylabel('Spectrum [dB]')
xlabel('Frequency [Hz]')
grid on;
```

```
function [RF] = qmixer(I,Q,Fc,LTAmp,LTPhase)
% qmixer    Performs quadrature mixing
%           Syntax: RF = qmixer(I,Q,Fc)
%               The vectors I and Q contain the quadrature
%               signals to be mixed. The carrier frequency
%               is specified by Fc in Hz. RF is the vector
%               that contains the quadrature mixed signal.
%               A 1uS sampling period is used.
%           Syntax: RF = qmixer(I,Q,Fc,LTAmp,LTPhase)
%               The effects of carrier leathrough is added.
%               Both, the leakthrough amplitude and phase,
%               can be adjusted.

if (nargin < 3)
    error('At least three parameters are required');
end

if (nargin < 4)
    LTAmp = 0;
end

if (nargin < 5)
    LTPhase = 0;
end

N = length(I);
n = 0:1e-6:(N-1)*1e-6;

RF = I.*cos(2*pi*Fc*n) - Q.*sin(2*pi*Fc*n) + ...
    LTAmp.*cos(2*pi*Fc*n + LTPhase*pi/180);
```

```
function [num2,den2,gain] = poly2quads(num,den)
% poly2quads    Convert n-th order polynomial into 2-nd order quadratics
%               Syntax: num2,den2,gain = poly2quads(num,den)
%                   The vectors num and den are the numerator and
%                   denominator filter coeffients respectively,
%                   generated using MATLABs built-in filter designers,
%                   such as butter.m. The n-th order polynomial is
%                   partitioned into 2nd-order polynomials as required by
%                   the quadratic_filter.m function. The gain is also
%                   extracted.
%               See also: quadratic_filter.m

if (nargin < 2)
    error('At least two parameters are required');
end;

if (length(num) ~= length(den))
    error('The length of the input vectors must be equal');
end;

gain = num(1);

numL = length(num);
test_sign = rem(numL-1,2);

if (test_sign == 1);
    num(numL+1) = 0;
    den(numL+1) = 0;
end

num_roots = roots(num);
den_roots = roots(den);

num_pair = cplxpair(num_roots);
den_pair = cplxpair(den_roots);

order = length(num)-1;
num2 = [];
```

```
den2 = [];
k = 1;

for n = 1:(order/2);

    x = poly([num_pair(k) num_pair(k+1)]);
    y = poly([den_pair(k) den_pair(k+1)]);

    num2(k) = x(2);
    num2(k+1) = x(3);

    den2(k) = y(2);
    den2(k+1) = y(3);

    k = k+2;
end
```

```
function [y] = quadratic_filter(a,b,g,x)
% quadratic_filter      Second-order filter function
%                       Syntax: y = quadratic_filter(a,b,g,x)
%                           The quadratic filter functions assumes that
%                           the filter numerator and denominator coeffients
%                           supplied are factored as 2nd order polynomials.
%                           MATLABs roots.m and poly.m function can be used
%                           to obtain the 2nd order polynomials. Alternatively,
%                           the function poly2quad.m can be used to automated
%                           the process of generating the 2nd order terms. The
%                           variable 'g', the first term of the numertor, is
%                           gain factor. The input signal 'x' is filtered
%                           according to these parameters to produce the
%                           filtered signal, 'y'.
%                       See also: poly2quads.m

gain = g;

number_of_quads = (length(a))/2;
total_w = (length(a)/2)*2;
w = zeros(1,total_w);

for m = 1:length(x)
    o = x(m) * gain;

    for n = 1:number_of_quads
        nn = (n-1).*2;
        ww = o + -b(nn+1)*w(nn+1) + -b(nn+2)*w(nn+2);
        o = ww + a(nn+1)*w(nn+1) + a(nn+2)*w(nn+2);
        w(nn+2) = w(nn+1);
        w(nn+1) = ww;
    end

    y(m) = o;
end
```

```
function [a,b] = freqplot(X, fs, k)
% Freqplot  Frequency plot of time domain signal.
%           Syntax: [a,b] = freqplot(X, fs, k)
%               The function freqplot takes the time signal,
%               X, for a specified sampling frequency, fs,
%               and generates its FFT. If k is 0, then a
%               double-sided FFT is given. For k = 1, a
%               single-sided plot is generated. The variables
%               'a' and 'b' are the frequency points and FFT
%               vector respectively. The units of 'b' are
%               specified as a rms voltage value. To convert
%               the output to a dBm value, use the following
%               equation:   dBm_out = 10*log10((b.^2)/1e-3);
%           See also: spur.m

if nargin < 3
   k = 1;
end

Nl = length(X);
df = fs/Nl;
XX = abs(fft(X)) / Nl;

N = floor(length(X)/2);

if k == 0
    a = (-(N):(N-1))*df;
    b = fftshift(XX);

   elseif k == 1
            a = (0:(N-1))*df;
       b = XX(1:N).*2;
            b(1) = b(1)/2;
            BL = length(b);
            b(2:BL) = b(2:BL)./sqrt(2);

        else
            warning on;
```

```
        warning ('Please enter a valid value for k - either 0 ...
        or 1. Type help freqplot for help');
        warning off;
    end
```

```
function [S,I] = spur(Y)
% SPUR      Finds the highest spur in a spectrum
%           Syntax: [S,I] = spur(Y)
%               SPUR ignores the global maximum of a samples spectrum, and
%               finds the next-highest peak in the spectrum. S is the
%               amplitude of the highest spur, and I is its index. Y is
%               the spectrum to be analysed.
% Author: G-J van Rooyen

[SMax, IMax] = max(Y);
Minim = min(Y);
Last = SMax;
n = IMax;
Y(n) = Minim;

while n < length(Y),
    n = n+1;
    if Y(n) <= Last
        Last = Y(n);
        Y(n) = Minim;
    else
        break;
    end;
end;

Last = SMax;
n = IMax;

while n >1,
    n = n - 1;
    if Y(n) <= Last
        Last = Y(n);
        Y(n) = Minim;
    else
        break;
    end;
end;
```

```
[S,I] = max(Y);
```

A.2 C++ code listing

A.2.1 Architecture source code

- The two listed source files `Converter.hpp` and `Converter.cpp` form the main structure of the emulator architecture. This is known as the base class. All the decendant classes include the model classes that function within the emulator. All the accompaning source files, which include the models as well as utility files, can be found on the accompaning CD included at the end of the thesis.

```
/* Converter.hpp The header file for the Converter base class. */

#ifndef CONVERTER_H
#define CONVERTER_H
#include <vector>
#include "rcptr.hpp"

using namespace std;

class Converter {
 protected:
   typedef struct OutputPort{
RCPtr<Converter> Module;
int BufferNumber;
   } OutputPort;

   /* define vector of structures called output_port */
   vector <OutputPort> output_port;
   vector <vector<float> > input_buffer; /* define input buffers */
   int samples_in; /* number of samples in input buffer at a time */
   int samples_out; /* the number of samples to deliver at a time */
   int output(int port_number, float sample);/* method to write to next object*/

 public:
   Converter() {samples_in = 1; samples_out = 1; }; /* default values */
   virtual ~Converter() {} ; /* destructor */

   float read(int buffer_number); /* read samples from an input bufer */
   /* write samples to an input buffer */
   int write(int buffer_number, float sample);
   /* determine the size of an input buffer */
   int buffer_size(int buffer_number);
   bool ready();/* check to see whether component is ready to process samples*/

   /* define a virtual process that is unique to each component */
   virtual int process() = 0;
   /* ID tag that gets defined by decendant classes */
   virtual const char* isA() const = 0;
```

```
   virtual void reset() = 0; /* method used to reset input buffers */
   int setup(); /* setup the parameters for each decendant class */

   /* Method used to setup a link between to objects */
   int link(int output_port_number, RCPtr<Converter> destination_module,
    int destination_portnumber);

   int numInputPorts(); /* returns number of input ports for a component */
   int numOutputPorts(); /* returns number of output ports for a component */
   /* Method to find out details (type and port number) about
   linked destination object */
   int query_link(int output_port_number, RCPtr<Converter>* destination_module,
    int* destination_portnumber);
};

#endif
```

```
/* Converter.cpp */

#include <iostream>
#include <vector>
#include <assert.h>
#include "Converter.hpp"

using namespace std;

/* Method used to place processed sample into the
linked components specified input buffer */
int Converter::output(int port_number, float sample)
{
  /* First check if specified port_number is valid */
  if (port_number + 1 <= output_port.size())
   {
    RCPtr<Converter> DestinationModule = output_port[port_number].Module;
    int DestinationBuffer = output_port[port_number].BufferNumber;

   if (DestinationModule == 0)
     {
         return 2;
     }
   else
     {
        /* write samples to target input buffer */
DestinationModule->write(DestinationBuffer, sample);
        return 1;
     }
   }

   else
   {
    cout << "Warning - invalid port number! Maximum number of output ports is "
    << output_port.size() << endl;
    return 0;
   }
}
```

```
/* Read from a specified buffer then delete sample from buffer to prevent
overflow */
float Converter::read(int buffer_number)
{
   float sample;
   /* Check contents of buffer */
      if (input_buffer[buffer_number].size() > 0)
      {

      sample = input_buffer[buffer_number][0];

      input_buffer[buffer_number].erase(input_buffer[buffer_number].begin());
      return sample;

      } else

       {
        return 0.0;
       }
}

/* write samples to a specified input buffer */
int Converter::write(int buffer_number, float sample)
{
   input_buffer[buffer_number].push_back(sample);
   return 1;
}

/* Ensure that buffer has samples for processing */
bool Converter::ready()
{
   return (input_buffer[0].size() >= samples_in);
}

/* determine the size of the buffer */
int Converter::buffer_size(int buffer_number)
{
   return input_buffer[buffer_number].size();
```

```
}

/* setup parameters can be defined in decendant classes */
int setup()
{
   return 1;
}

/* member function that sets up connection between two components */
int Converter::link(int output_port_number, RCPtr<Converter> destination_module,
                    int destination_portnumber)
{
   assert(output_port_number + 1 <= output_port.size());

    output_port[output_port_number].Module = destination_module;
    output_port[output_port_number].BufferNumber = destination_portnumber;
    return 1;
}

/* Retrieve information about the linked object for a given output port */
int Converter::query_link(int output_port_number,
        RCPtr<Converter>* destination_module, int* destination_portnumber)
{

*destination_module = output_port[output_port_number].Module;
*destination_portnumber = output_port[output_port_number].BufferNumber;

return 1;

}

/* return total number of input ports */
int Converter::numInputPorts()
{
return input_buffer.size();
}

/* return total number of output ports */
```

```
int Converter::numOutputPorts()
{
 return output_port.size();
}
```