Reusable Software Defined Radio Platform for Micro-satellites

John Foster van Wyk

Thesis presented in partial fulfilment of the requirements for the degree Master of Science in Engineering (Electronic Engineering with Computer Science) at the Stellenbosch University

Supervisor: Dr G-J van Rooyen

March 2008

Declaration

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

Signature

Date

Abstract

This thesis describes the design and implementation of a software platform for software defined radio (SDR). This platform was to form part of an experimental satellite payload. Several other experiments were also housed on this platfrom and subsequently had to be incorporated into the design. The hardware components of the payload were already determined at the start of the project but firmware had to be created as part of the project. The software platform was based on the Linux kernel. Device drivers had to be designed for the hardware and firmware components. These drivers were designed so that standard Unix utilities could be used to interact with them. This allowed for easy testing of the system and the programs running on it. The use of the platform for modulation and demodulation of analogue signals was demonstrated using a proof-of-concept SDR application.

Opsomming

In hierdie tesis word die ontwerp en implimentasie van 'n sagteware gedefinieerde radio (SDR) bespreek. Die platform het deel uitgemaak van 'n eksperimentele satelliet loonvrag. Verskeie ander eksperimente het ook die platform gedeel, en daar moes gevolglik voorsiening vir hulle gemaak word gedurende die ontwerp. Hardeware vir die stelsel was buite die projek bepaal, maar fermware moes geskep word as deel van die projek. Die sagteware platform is op die Linux bedryfstelsel gebasseer. Drywers moes gevolgelik vir die hardeware en fermware komponente ontwerp word. Die drywers is so ontwerp dat die standaard Unix stelsel programme gebruik kon word om daarmee te kommunikeer. Dit het tot gevolg gehad dat die drywers maklik getoets kon word, so ook die programme wat op hierdie drywers staatmaak. Die gebruik van die platform vir modulasie en demodulasie van analoog seine is gedemonstreer deur van 'n toets SDR program gebruik te maak.

Acknowledgements

I would like to thank:

- My supervisor, Gert-Jan van Rooyen, for his guidance,
- My parents, for their understanding support,
- My Heavenly Father, for the wonderful opportunities he has given me.

Contents

Nomenclature **x**

Terms of Reference **xi**

1 Introduction **1**
1.1 Origin of project . . . 1
1.2 Objectives of this project . . . 1
1.3 Structure of this thesis . . . 2

2 Background **4**
2.1 Software Defined Radio . . . 4
2.1.1 Defining Software Defined Radio . . . 4
2.1.2 Benefits . . . 5
2.1.3 Challenges . . . 5
2.1.4 Processing Hardware . . . 9
2.2 Stellenbosch SDR . . . 11
2.3 Operating System . . . 13
2.3.1 Unix . . . 13
2.3.2 Linux Kernel . . . 15
2.3.3 Cross-Platform Development . . . 15
2.3.4 Linux Device Drivers . . . 17

3 High-level design **21**
3.1 Overview . . . 21
3.2 Background . . . 21
3.2.1 Experimental Payload Hardware Components . . . 21
3.2.2 Experiments . . . 23
3.2.3 Daughterboard . . . 24
3.2.4 OBC . . . 28
3.3 Requirements Analysis . . . 30
3.3.1 Design Methodology . . . 30
3.3.2 Functional Decomposition . . . 31
3.3.3 Required behaviour . . . 32

3.4 System Design 34
3.4.1 Firmware 34
3.4.2 OBC Software 36

4 SIC Firmware Design **39**
4.1 Introduction 39
4.2 Host Interface 39
4.2.1 Registers 39
4.2.2 Sample Buffering 42
4.2.3 Dataflow 43
4.3 Subsystems 44
4.3.1 Peripheral Sampling Subsystem 44
4.3.2 Quadrature Sampling Subsystem 45
4.3.3 Quadrature Transmission Subsystem 46
4.3.4 General IO Ports 49

5 OBC Software **50**
5.1 Overview 50
5.2 Operating System 50
5.2.1 Choice of Operating System 50
5.2.2 Kernel Configuration 51
5.2.3 Bootloader 53
5.2.4 Root File system 54
5.3 SIC Drivers 56
5.3.1 Experimental Payload Drivers 56
5.3.2 IQ sampling and transmission 60
5.3.3 DDS proof-of-concept driver 65
5.4 OBC Drivers 65
5.4.1 CAN Driver 65
5.4.2 NAND Flash Interface 67
5.4.3 LVDS Port Interface 68
5.5 User space 69
5.5.1 User space services 69
5.5.2 Experimental Payload Interface 70
5.5.3 IP over CAN 70

6 Evaluation **72**
6.1 Software Environment 72
6.1.1 Bootloader 72
6.1.2 Operating System 72
6.1.3 NAND Flash 74

6.1.4 CAN . . . 76
6.1.5 IP over CAN tunnelling . . . 77
6.2 Firmware . . . 78
6.2.1 Register Input and Output . . . 78
6.2.2 Peripheral ADCs . . . 79
6.2.3 Peripheral Sampling system . . . 81
6.2.4 IQ DACs . . . 82
6.2.5 NCO . . . 84
6.2.6 IQ sampling . . . 85
6.2.7 IQ transmission . . . 87
6.3 SDR Proof-of-concept application . . . 89
6.3.1 OBC to PC using the IQ transmission path. . . . 89
6.3.2 PC to OBC . . . 91
6.3.3 OBC to PC using the NCO . . . 92
6.3.4 Result . . . 94

7 Conclusion **95**
7.1 Thesis Results . . . 95
7.2 Future Improvements . . . 96
7.3 Final remarks . . . 97

Bibliography **98**

A SIC register descriptions **101**
A.1 Peripheral Sampling Subsystem . . . 101
A.2 Quadrature Sampling Subsystem . . . 103
A.3 Quadrature Transmission Subsystem . . . 104

B Source Code **105**

List of Figures

2.1 Direct conversion front-end. 7
2.2 Graphical representation of the SU SDR framework. 12
2.3 Graphical representation of the layers in a Unix system 14

3.1 Block diagram of experimental payload with important connections. 27
3.2 Block diagram of SH4 OBC. 29
3.3 Block diagram of supplied components. 30
3.4 Use case diagram for experiments. 33
3.5 Top-level block diagram of firmware. 35
3.6 Software components of SDR system. 36
3.7 Software components for experiments. 38

4.1 Clock and data gating 41
4.2 Gated clock trigger circuit. 42
4.3 Top-level design of the peripheral sampling subsystem. 45
4.4 Top-level design of the quadrature sampling subsystem. 46
4.5 Top-level design of the quadrature transmission subsystem. 47
4.6 Block diagram of the NCO's input-synchronisation circuit. 48
4.7 General purpose IO block diagram. 49

5.1 Flowchart of peripheral sampling driver's read system call. 58
5.2 Flowchart of busy-waiting IQ-transmission driver's write system call. 61
5.3 Flowchart of interrupt driven IQ-sampling driver's interrupt handler. 62
5.4 Flowchart of interrupt driven IQ-sampling driver's read system call. 63
5.5 Flowchart of interrupt driven IQ transmission driver's interrupt handler. . . 64
5.6 Flowchart of interrupt driven IQ transmission driver's write system call. . . . 64

6.1 Writing data into an FPGA register. 78
6.2 Reading from an FPGA register. 79
6.3 Peripheral ADC signalling. 80
6.4 Sampling a 1 kHz sinusoid on a peripheral ADC. 82
6.5 Sampling the Magnetosphere experiment's magnetic sensors. 83
6.6 Quadrature output of the NCO. 85
6.7 NCO output with altered, phase, frequency and amplitude and DC-offset. . . 86

6.8 Quadrature signal sampled on the IQ ADCs. 86
6.9 Ramp waveform output, generated on the OBC. 88
6.10 Sinewave output, generated on the OBC. 88
6.11 Input and output of the reconstruction filter. 90
6.12 Busy-waiting IQ transmission driver states. 91
6.13 Interrupt driven IQ transmission driver states. 92
6.14 Stem plot of FSK signal sampled on the daughterboard. 93
6.15 Busy-waiting IQ sampling driver states. 93
6.16 Interrupt driven IQ sampling driver states. 94

List of Tables

4.1 IQ sampling word format. 46

5.1 Peripheral sampling driver's output word format. 59
5.2 CAN Message identifier format. 65
5.3 CAN Driver's packet format. 66

6.1 STREAM benchmark results on SH4 OBC. 74
6.2 Transfer speeds of NAND partition 76
6.3 Signal descriptions for logic analyzer output. 80
6.4 Measured DAC output voltage. 84
6.5 Debug signal descriptions for modem tests. 90

B.1 Contents of the accompanying disk. 105

Nomenclature

Acronyms

ADC	analogue-to-digital converter
DAC	digital-to-analog converter
DSP	Digital signal processor
GPP	General purpose processor
OS	operating system
NCO	numerically controlled oscillator
FPGA	field programmable gate array
CPU	Central processing unit
OBC	On-board computer
SIC	Signal interface card
CORDIC	Co-ordinate rotation digital computer
DDS	Direct digital synthesis
EDAC	Error detection and correction
RAM	Random access memory
SRAM	Static RAM
SDRAM	Synchronous dynamic RAM
VHDL	VHSIC hardware design language
VHSIC	Very high speed integrated circuit
XML	Extensible markup language
SDR	Software defined radio
CAN	Controller area network
LVDS	Low voltage differential signaling
RTOS	Real-time OS
FIFO	First in, first out
FHS	Filesystem Hierarchy Standard

Terms of Reference

This project was commissioned by Dr G-J van Rooyen of the Digital Signal Processing Group at the Department of Electrical and Electronic Engineering at the University of Stellenbosch. The instructions were as follows:

- Create a software platform on the satellite hardware that can host experiments created with the University of Stellenbosch's Software Defined Radio architecture (SU SDR).
- Create suitable firmware for the signal interface card (SIC), being developed at the University of Stellenbosch (SU), so that it may perform the above and also be reusable in future versions of the SIC.
- Create the necessary software modules to enable the above system to host other satellite experiments as well.
- Demonstrate the system by hosting a software defined radio on it.

Because of limited resources, the project was subject to the following conditions:

- All software were to be designed to run on the SH4 based on-board computer (OBC) board, designed and supplied by SunSpace.
- The Signal interface card (SIC), which was being designed at the Signal Processing group at Stellenbosch University, was to be used in conjunction with the OBC supplied by SunSpace.
- The hardware would not be available at the start of the project. Development was to begin without access to hardware.
- The specifications of the hardware would not be finalised until after the completion of the project.
- SunSpace supplied firmware on the OBC would be subject to change at the discretion of SunSpace.
- Exclusive access to the OBC was not available during development. This hardware had to be shared with another project that was also utilising a SunSpace supplied on-board computer.

- Access to the SIC was shared with another project.
- The satellite integration team had priority access to all hardware, including development hardware not supplied by SunSpace.

Chapter 1

Introduction

1.1 Origin of project

In 2006 the SA Government's Department of Science and Technology commissioned Stellenbosch University for the design and construction of a micro-satellite [15]. The primary objective would be to take high-resolution images, for agricultural and environmental purposes, by means of a multi-spectral imager.

The satellite was built by Sun Space and Information Systems Pty Ltd (SunSpace), but provision was made for a small experimental payload to be built at the University. This payload consists of the following components:

The radiation experiment, which endeavours to measure the effects of space radiation on commercial-off-the-shelf components and investigate radiation mitigation techniques.

The string experiment, proposed by the Nelson Mandela Metropolitan University. This experiment attempts to measure non-linear effects of string dynamics in micro-gravity.

The magnetosphere experiment, proposed by the University of KwaZulu-Natal. An attempt will be made to map the earth's magnetosphere.

The SA-AMSAT payload. This component was supplied by the Southern African Amateur Radio Satellite Association [6] and features a digipeater, parrot and voice beacon.

The SDR experiment, proposed by the Signal Processing (SP) Group of Stellenbosch University. Its goal is to provide a real-world test bed for the SP Group's software defined radio (SDR) research [31].

1.2 Objectives of this project

All the experiments listed above are integrated into a single payload, called the experimental payload. This unit would in turn be integrated into the rest of the satellite. The purpose of this project is to design and implement the software that will run on the experimental payload, enabling it to execute the listed experiments.

The payload consists of a single-board computer, acting as processing platform, and a daughterboard which provides signal interfacing to the experiments. The hardware of the single-board computer is identical to the satellite's primary on-board computer (OBC), and was be supplied by SunSpace. The daughterboard was to be designed and built concurrently with this project.

The objectives for this project can now be outlined:

- Establish a software environment on the OBC on which the software necessary for the experiments can be executed.
- Implement the necessary firmware components for the Field Programmable Gate Array (FPGA) on the daughterboard.
- Implement the necessary device drivers to allow the OBC to access the components on the daughterboard.
- Integrate the system with SunSpace's systems to allow the satellite to control the experimental payload.
- Implement a proof-of-concept application to demonstrate the use of the platform for SDR.

1.3 Structure of this thesis

This thesis is structured as follows:

- Chapter 2 begins by discussing the principles of SDR. The software of the SP Group's SDR project is then briefly discussed. The chapter concludes by providing some background information about the operating system kernel chosen for the experimental payload.
- In Chapter 3 the experimental payload hardware and the experiments are discussed. The software requirements of the experiments are analysed and functional requirements are extracted. The software components are then designed to meet those requirements.
- In Chapter 4 the design and implementation of the firmware components for the daughterboard is discussed.
- Chapter 5 describes the implementation of the software environment on board the OBC and explains the design considerations that were made. The design of the drivers for the daughterboard and the OBC's peripherals are discussed next. Finally, the structure of the software responsible for integration of the experiments with SunSpace's systems, are explained.

- The system is evaluated in Chapter 6. The functionality of the software environment, firmware components and their drivers are verified. Finally, a proof-of-concept SDR application is tested.
- Chapter 7 gives a review of the project and discusses possible improvements to the system. Finally, concluding remarks are given.

Chapter 2

Background

2.1 Software Defined Radio

2.1.1 Defining Software Defined Radio

Radios are devices which transmit and receive signals over a distance by transmitting and receiving radio waves. Radio waves do not rely on a physical medium for propagation; this property of radios has made them essential for modern communication. Radios accomplish their function by radiating electromagnetic waves, which are modulated by some message signal on the transmission side and by demodulating the received waves on the reception side. The message signal can be anything from music, played on a local radio station, to digital weather maps transmitted from a weather satellite.

Radios differ mainly in two areas. The first is concerned with the wavelength of electromagnetic energy that is radiated. This determines which part of the electromagnetic spectrum the radio functions in. The second is concerned with how this electromagnetic wave, called the carrier, is modulated to allow a message signal to be transmitted. For two radios to be compatible, both of these have to match up. Commercial FM radio stations, for example, use the same modulation technique as traditional television broadcasts to transmit sound, but FM radios cannot play the sound from television broadcasts because the two systems operate in different parts of the radio spectrum. Similarly even though Bluetooth and 802.11b Wireless LAN devices operate in the same part of the radio spectrum (2.4GHz ISM Band), they are not interoperable at the physical layer level because they employ different modulation[1] techniques.

A software defined radio (SDR) can now be defined as a radio where modulation or demodulation of a transmitted signal is performed in software on a digital signal processing platform. This differs from conventional radio where processing of the signal was usually accomplished in the analogue domain by application-specific circuits. A typical SDR requires

[1]Bluetooth radios use Gaussian Frequency Shift Keying, while 802.11b radios use a technique known as complementary code keying.

a radio front-end (this will be discussed in more detail shortly), a conversion stage and a processing platform. In comparison, conventional radios also require some form of front-end, for instance the tuning circuit in a FM radio, but a comparatively simple dedicated electronic circuit replaces the conversion and processing stages.

Processing baseband signals in the digital domain does not necessarily constitute a SDR. The approach taken by Reed [24] is the following: software radios represent a paradigm shift from fixed, hardware-intensive radios to multi-band, multi-mode, software-intensive radios. This is the definition adopted by this thesis.

2.1.2 Benefits

The most notable advantage of SDR over the conventional approach is the ease of reconfiguring the radio. Because all SDRs require virtually the same hardware components, the only thing required for a SDR to support a new waveform is, in theory, a software change. Where conventional radio would require some sort of physical intervention to change the circuits, a SDR could in principle be reconfigured by simply uploading new software into the device.

Another potential benefit is the cost savings that could result from hardware platforms becoming standardised. Mass-produced software radio platforms could reduce the engineering effort required to design and build a new hardware radios for specialised applications. The roll-out of new air interfaces would also be less costly as the recurring costs associated with reprogramming software radios could conceivably be almost nothing.

The SDR Forum [5] lists the following as the key benefits of SDR:

- Standard architecture for a wide range of communications products
- Non-restrictive wireless roaming for consumers by extending the capabilities of current and emerging commercial air-interface standards
- Uniform communication across commercial, civil, federal and military organisations
- Flexibility and adaptability
- Potential for significant life-cycle cost reductions
- Over-the-air downloads of new features and services as well as software patches
- Advanced networking capabilities to allow truly “portable” networks

2.1.3 Challenges

Building the ideal SDR[2] might not be possible in practise. It is however helpful to consider the ideal SDR as a benchmark to compare practical systems with. Desirable features of this radio include the following:

[2]The term used by the SDR Forum to describe the ideal SDR is Ultimate Software Radio

- Signals are digitised in the radio frequency (RF) band. This allows the radio to function over an arbitrarily wide bandwidth and support potentially limitless air interfaces.
- All processing of waveforms are done in software. The ideal SDR does not use application specific integrated circuits (ASIC).
- The processing platform has sufficient resources to handle both current and future waveforms.

Antennas

In order for a SDR support a variety of air interfaces, it requires an antenna that can function in all the bands of interest. This might require an antenna with an unattainably high bandwidth. In base station applications the problem may be addressed by utilising several antennas to cover the whole spectrum of interest and a switch to change between them. Therefore this is of more concern in ideal handheld devices.

Converters

Ideally the signal would be digitised directly at the antenna. This would however require signal conversion devices working at extremely high sample rates. Nyquist theorem requires that a signal be sampled at more than twice its rate of change in order to preserve all the information in the signal. In practise at least 2.5 times the bandwidth is required [14]. An 802.11b[3] receiver, would, for example, require the analogue-to-digital converter (ADC) to operate at a sample rate of around 6 GHz. At the time of writing, ADCs capable of sampling at these speeds were quite exotic devices having power requirements such that they could not be considered for mobile, battery-powered devices. The major hurdle to increasing sampling rates is aperture jitter. Aperture delay is the amount of time it takes for the ADC to take a sample after receiving the signal to do so. If this delay were constant it would only amount to an inconsequentially small phase shift, but this delay has a random component. The variance of the aperture delay is known as aperture jitter and has the effect of diminishing the converter's resolution.

Another problem that arises when considering digitising RF is the effect of adjacent channels. In conventional radios a front-end would filter out unwanted signals in nearby bands before the desired signal is processed. In the ideal SDR this filtering, or channel selection, is another function performed in software. It may happen that the signal of interest for a particular radio is weak when compared to a signal in an adjacent channel. The converter would have to have sufficient dynamic range to accommodate the large signal while still adequately digitising the smaller signal. It is therefore required that converters have

[3] 802.11b operates at carrier frequencies around 2.4GHz

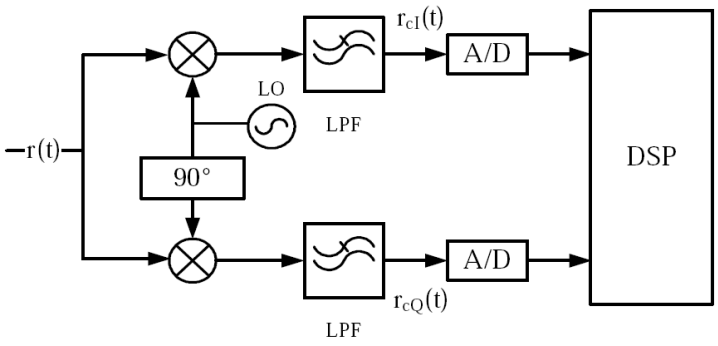


Figure 2.1: *Direct conversion front-end.*

higher dynamic range than would be required for conventional digital radios with channel selection implemented in the front-end.

While it can reasonably be assumed that the speed of converters will increase as technology progresses, the limitations of fabrication technology will for the foreseeable future be a barrier to the creation of the ideal SDR. Until then, trade-offs will have to be made between bandwidth, dynamic range, power consumption, and cost when designing SDR systems [24].

Front-ends

An alternative to sampling signals at RF is to translate the signal to a lower intermediate frequency (IF) by employing a radio front-end. This allows the conversion process to occur at a lower frequency. This is advantageous because of the limitations of available converter technology and often allows for converters with better spurious free dynamic range to be used. The use of RF front-ends is, however, still a compromise as they have to be designed with a specific band of interest in mind.

The superheterodyne architecture is an RF front-end that is suitable for SDR. Down-mixing may occur in multiple stages depending on frequency requirements. An RF filter is required to eliminate image frequencies that result from the signal being mixed down in several stages. To reconfigure the carrier frequency or signal bandwidth requires that the RF or IF bandpass-filters be, respectively, reconfigured. Designing these filters to be reconfigurable over a wide bandwidth is difficult to near-impossible and requires large circuit layouts. This limits the flexibility of the superheterodyne front-end for SDR applications. Despite its limitations it remains one of the leading solutions used in practical SDRs.

Another method is to translate the signal down to baseband in a single stage. This front-end is know as the direct conversion front-end or zero-IF mixing. This architecture uses quadrature mixing to translate the signal of interest to baseband. The resulting baseband signal is complex, requiring separate converters for the in-phase, I, and quadrature, Q, components. The advantage of this architecture is that it is image free and therefore does not

need the image rejection filters that limit the superheterodyne's flexibility. Direct conversion does, however, have several drawbacks [28]. It is highly sensitive to imbalances between the I and Q channels as well as the phase offset between the local oscillators. Furthermore, mixing the signal down to baseband means that $1/f$ noise, also known as flicker noise, becomes a problem: $1/f$ noise is one of the largest known contributors to noise in CMOS circuits [14, p. 94]. Because its intensity increases as frequency decreases, it is at its worse at baseband. Lastly, having the local oscillator at the carrier frequency means that carrier leak-through compromises the DC performance of the receiver. Because of these difficulties the front-end has historically not received much attention. The relatively simple construction, and therefore small size, of the front-end and the application of digital compensation techniques has however contributed to renewed interest.

Processing power required

When considering the ideal SDR, all signals would also have to be processed at the incoming sample rate, which will be in the RF range. It is estimated that GPRS requires approximately 100 million instructions per second[4] (MIPS) and EDGE up to 1000 MIPS to deliver its full data rate when processed at baseband. Processing these protocols at RF would increase the computational load considerably. Processing more advanced protocols, such as 3G, at RF becomes infeasible with current processing techniques [21]. Furthermore, power consumption of processors increases proportionately to their speed. This is counter to the power efficiency required for handheld devices with limited battery power. In base station applications power consumption is less problematic, and therefore processing power is less constrained.

Software

As the name implies, software is central to the functioning of any SDR. Ideally, the software would be completely independent from the hardware on which it runs. The hardware in an ideal software radio would therefore serve the software running on it. In reality the situation is reversed: the constraints imposed by hardware governs much of the software written for SDR.

A famous example of this dependence on hardware is the Speakeasy radio [10]. Developed for the US Department of Defence, it was to support 10 different waveforms. To accomplish its objectives, the designers had to use several of the most powerful DSPs available at the time. The software was written specifically for these processors to extract the maximum performance from the available architecture. By the time the design was completed, newer, more powerful DSPs had emerged. The software, tailored to the older DSP architecture, was, however, unable to benefit from the advancements in DSP technology.

[4]The MIPS concept is somewhat inadequate as the amount of work done by an instruction isn't constant over different architectures. It is only used here as a ballpark figure.

The important lesson to be learnt from the Speakeasy project is that portability of software is more valuable than the performance gains of highly optimised, platform specific software. When considered in the short term, this conclusion might at first seem counterintuitive. The pace at which semiconductor technology advances, often attributed to Moore's Law [17], leads to the conclusion that portability is of extreme importance.

In order to leverage the advances in hardware, the software for SDR needs to be decoupled from the hardware. It is desirable to abstract the hardware to such a degree that the SDR programmer only has to focus on the actual algorithms needed to define waveforms. To this, end software architectures, or frameworks, have been defined to aid the SDR developer in writing portable code. Gnuradio [1], and SCA [21] are examples of such systems. The University of Stellenbosch's SDR project has also defined such a framework to aid in SDR research conducted there.

2.1.4 Processing Hardware

While SDR focuses mostly on the software aspect of systems, it cannot be completely removed from the hardware that it is designed for. The following is a brief discussion of processing hardware suitable for SDR. Application-specific circuits (ASICs) are not considered as suitable for the ideal software radio due to their inherent inflexibility. While ASICs may be designed so that some of their parameters may be adjusted, they can only perform one task (albeit perform that one task well).

DSP

Digital signal processors (DSP) are in essence specialised microprocessors. They are optimised to perform computations by executing instructions in a tight loop, since the nature of digital signal processing functions lends itself to this arrangement. The most important feature that sets DSPs apart from other microprocessors is the inclusion of highly efficient multiply and accumulate (MAC) instructions, since this is an operation that is central to most DSP tasks [20]. DSPs also have high memory bandwidth, usually employing a Harvard architecture or an architecture derived from it. This allows them to fetch data and instructions simultaneously.

The term DSP encompass a wide range of processors. The most important distinctions between these are whether they operate on fixed-point or floating-point data. The other important specification is the size of the accumulator, therefore its precision. DSP varieties are further differentiated into devices optimised for performance and devices optimised for efficient power consumption [20].

DSPs are often employed without operating systems and programmed in assembly language. High level language tools, especially C compilers, are available for most DSPs, but performance critical sections are usually still programmed in assembly language [9]. Execution time for instructions is usually highly deterministic. This coupled with the lack of an

underlying operating system allows the DSP programmer to achieve very precise instruction timing at the cost of code portability.

Due to their excellent signal processing qualities, DSPs are a highly suitable platform for SDR. Code portability is the biggest drawback from an SDR point of view.

GPP

General-purpose processors (GPPs) like the versions found in workstation personal computers (PCs) are not as efficient as DSP platforms for SDR purposes. What makes GPPs especially attractive is their relatively low cost. With PCs becoming almost ubiquitous, the cost of relatively powerful workstations have fallen markable.

GPPs are usually used together with an operating system (OS). This has the advantage of making software more portable because of the hardware abstraction afforded by the OS. By using GPPs, the SDR author is effectively exploiting the massive growth in GPP performance often attributed to Moore's Law [10]. The use of an underlying OS does, however, have the drawback that execution times become hard to predict. The main factors contributing to this are other tasks competing for CPU time and contention arising from many peripherals sharing the same bus. GPP execution is also inherently less predictable as a result of the use of data and instruction caches to speed up execution.

This execution jitter can be addressed by buffering samples and thereby decoupling the real-time stream from the processing of samples. This temporal decoupling has the added advantage that more efficient, longer-running algorithms can be used than would have been possible had execution time been bound by the limitations of lock-step processing. Although the processing power required for such a system is higher than for a more conventional tightly coupled system, it is capable of real-time operation as long as the average processing speed is higher than the required rate. High buffer lengths unfortunately leads to higher latencies, which is undesirable. Buffer lengths therefore require much attention and tuning in such a system to ensure correct operation while minimising latency.

Many software tools are available for developing on GPPs. The tools are also more mature than for many other platforms [20]. A foreseeable problem with GPP architectures is their limited memory bandwidth compared to DSPs.

FPGA

Field Programmable Gate Arrays are configurable integrated circuits. They consist of configurable logic blocks, configurable routing and configurable pads at the periphery that enable interaction with the outside [20]. By configuring the logical blocks and the interconnections between them, FPGAs can be used to synthesise complicated logic circuits. In addition to these configurable logic blocks, some devices contain embedded system blocks that perform specialised functions, like multipliers and RAM.

The logic elements usually contains several inputs and outputs; the number depends on the specific model. The logical functions between these inputs and outputs are configurable

and may include memory elements like flip-flops. The complexity of the functions that may be achieved is highly dependant on the specific device. Some devices can synthesise complicated logical functions with latching outputs within single blocks, while other devices may only achieve simple logical functions or single flip-flops. It also follows that the number of blocks on a single device varies widely between different vendors and device families.

FPGAs are available in one-time-programmable as well as reprogrammable varieties. Only the reprogrammable devices are of interest here because of the flexibility they afford. Configuration data for reprogrammable FPGAs are stored in memory elements: the popular options are Flash-based and SRAM-based FPGAs. The Flash-based devices have the advantage that their configuration is preserved when power to the device is removed. They are therefore immediately functional at power-up. SRAM devices require that the configuration data be uploaded to the device before it can be used. This, however, presents a beneficial advantage over Flash-based devices, in that SRAM devices can be very easily reconfigured, even while powered up. It is even possible to reconfigure part of a SRAM device while another part is functioning [20].

A very popular application of FPGAs are as ASIC replacements. The reconfigurable nature of FPGAs allows easy development without the delays and costs associated with ASIC fabrication. As a result, most FPGA software tools are geared toward this application. The result is that these tools support only one-time configuration of FPGAs. Utilising the dynamic reconfiguration ability of FPGAs is therefore hampered by the lack of tools. Such tools are, however, beginning to emerge [20].

Because FPGAs are in essence configurable circuits, they are capable of ASIC-like performance. FPGAs can be configured to perform different functions in parallel and the FPGA programmer can exploit parallelism in algorithms to achieve performance unheard of with microprocessors. Programming FPGAs to perform complex signal processing functions is, however, still a daunting task. This is partially because of the relative immaturity of FPGA tools compared to the vast amount of tools available for microprocessors.

Despite the difficulty associated with programming FPGAs, they are a highly suitable platform for SDR. Their greatest strengths are their very high speed operation and relatively low power consumption.

2.2 Stellenbosch SDR

The SU SDR project aims to provide a platform-independent software framework for implementing software defined radios. A layered approach is taken with each layer providing a more abstracted view of the radio. The lowest layer is know as the converter layer. Each converter is a signal processing block with signal inputs and outputs as well as controls implemented as attributes. The second layer, known as the subcontroller layer, combines a group of subconverters into larger unit, the subcontroller. The subcontroller is responsible for scheduling the execution of the converters. At the top is the SDR application, comprising

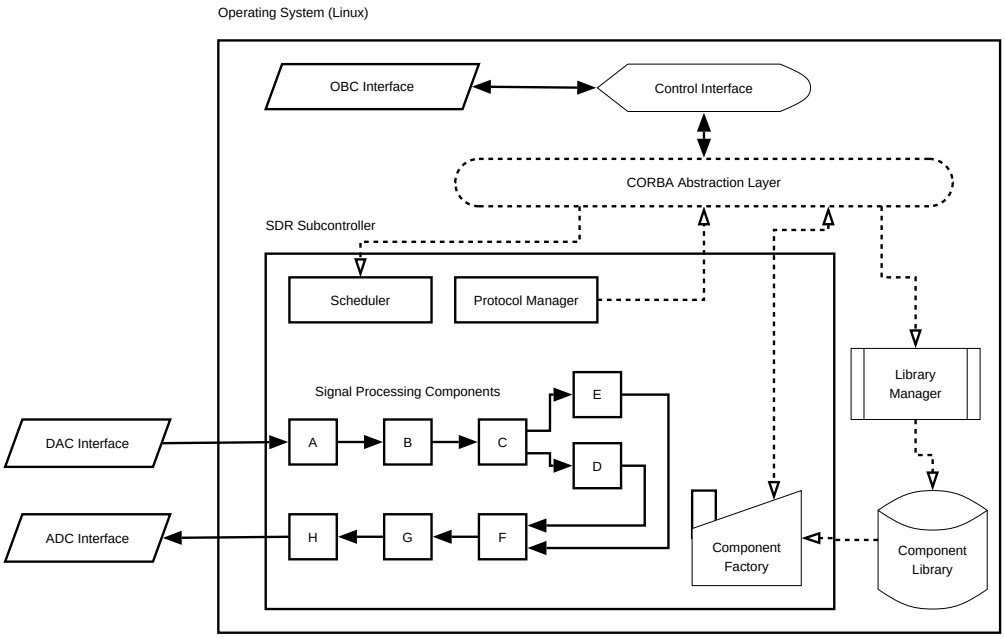


Figure 2.2: *Graphical representation of the SU SDR framework.*

a collection of subcontrollers. To address portability over hardware platforms, converters are described in a domain-specific language based on the extensible markup language (XML). These generic converters are then transformed into native software when compiled for a specific platform. The first compilation step is to transform the generic, XML-based definitions into an intermediate language by an XSLT processor. Thereafter, native software is built in a manner consistent with the requirements of the target platform. Research has been done on compilation paths for GPP [13] and FPGA [9] architectures, with functional proofs of concept for both.

The GPP path is of particular importance for this project as the project platform is based on a SH4 microprocessor which falls in the GPP class.

Software generated by the GPP compilation path is implemented in C++ and makes use of the object-oriented design paradigm. All converters are derived from a base converter class. Signal sources and sinks, which are ultimately the reception and transmission antennas, are also abstracted to converters. These special converters are specific to the target platform and therefore not defined in the generic high level language. The interconnection of these converters with the generic derived converters are, however, transparent to the SDR application.

The GPP implementation was developed on the SU DSP lab computers which use the Unix-like Linux-based operating systems. At the time of writing, no attempt has been made to port the architecture to other operating systems. This imposes the requirement that the satellite platform must provide a similar interface to that of these machines—specifically a POSIX-compliant environment (see section 2.3.1) .

2.3 Operating System

The software platform developed in the project is based on the Linux kernel and is therefore a Unix-like system. Before discussing the software designed in this project, some background information on Unix systems and Linux will be provided. This section is intended to provide sufficient background information for later chapters.

2.3.1 Unix

Unix was created in 1969 at Bell Laboratories and became the successor to the Multics time-sharing operating system. Multics was an extremely ambitious attempt to create a highly advanced operating system. Due to its complexity, it ultimately failed, but it contributed many valuable ideas to operating system design. While Unix incorporated concepts from Multics, its underlying principle was undoubtedly simplicity. While Multics had thousands of pages of technical specifications, the first implementation of Unix was reportedly written in two days by its principle inventor, Ken Thompson [23].

The design of Unix proved to be extremely durable. After almost 40 years, the basic design is still recognisable. It has also influenced the design of many systems developed after it. Several different versions of Unix exist today, but they all share the same basic ideas and structure. To maintain a level of compliance between the different Unix versions, the Institute of Electrical and Electronics Engineers (IEEE) defined a family of standards named POSIX. POSIX stands for Portable Operating System Interface and has been given the designation IEEE 1003 [27]. The 1003.1 interface standard is of particular importance, it defines a set of interfaces that all POSIX-compliant systems must provide; and therefore ensures a measure of portability between systems.

Unix systems can all be partitioned into three layers (Fig. 2.3). At the bottom is the hardware. On top of the hardware layer is the Unix kernel, which is responsible for allocating the hardware resources to the layer above it. The top layer, known as user space, is where application programs, known as processes, are executed. This is the layer that does the useful work; the lower layers exists to serve the user space processes.

The software interface between processes and the kernel is called the system call interface. In addition to the system call interface, processes also have access to a set of commonly used library functions. These are run on top of the system call interface and is therefore also executed in user space. Processes are usually built on top of the system libraries, but may access the kernel directly through the system call interface. The system calls and library functions were traditionally described in sections two and three, respectively, of the Unix Programming Manual[5]. It can still be found in the online documentation of many modern Unixes under the same sections. The difference between code executing in kernel space or

[5]For the nostalgic, digital images of the first edition of the Unix Programmer's Manual can still be found at http://cm.bell-labs.com/cm/cs/who/dmr/1stEdman.html.

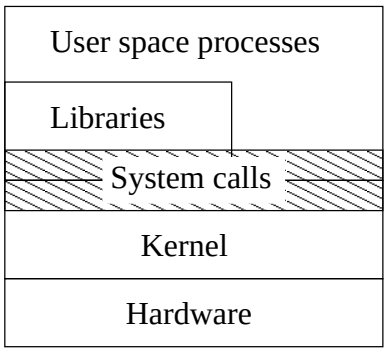


Figure 2.3: *Graphical representation of the layers in a Unix system*

in user space lies in the privileges that the code exerts. Kernel code may access any part of a system while user code can only access the system through the kernel. This distinction allows many processes to share the same hardware at once; it also allows the kernel to protect user space programs from malfunctions in other user programs.

In essence, the kernel's function is to respond to resource requests from user space, be it execution time on a CPU or access to a peripheral device. The functions of the kernel can be divided into several subsystems, each with a well-defined purpose:

Process management. This entails scheduling of processes for execution on CPUs and providing methods for processes to communicate, know as inter-process communication (IPC).

Memory management. Each process is executed in its own virtual address space; this protects processes from malfunctions (or malicious intent) in other processes.

File system. The file system plays a central role in a Unix system. All information (programs and data) is stored in files and directories. The concept of files is so important that many devices emulate the behaviour of regular files in order to integrate better into the system.

Device control. An important role of the kernel is to provide programs access to the functionality of peripheral devices without exposing them to the complexities of interfacing these devices. Software known as device drivers are employed to handle hardware devices.

Networking. As with device drivers, the kernel provides programs a means to utilise network connections, without exposing them to the intricacies of handling network packets.

2.3.2 Linux Kernel

Linux is an implementation of a Unix-like kernel. The first version was released by Linus Torvalds in 1991. It is currently being developed and maintained by a diverse community of volunteers headed by its original author.

Linux is a monolithic kernel—this means that the whole kernel (all the services it provides) is implemented as a signal piece of code that executes in the same memory space. All the code in the kernel has the same privileges. The rival architecture is the microkernel-based systems, in which services are provided by separate processes, running in different address spaces. An important feature of Linux is that a running kernel's functionality may be expanded by loading pieces of object code, known as kernel modules. The modules are linked into the running kernel and effectively becomes part of the running kernel. They therefore still share the same address space as the rest of the kernel.

Loadable modules are often used for device drivers. Drivers may, however, also be directly compiled into the kernel as is sometimes required. The drivers needed to mount the root file system (where other drivers may be found) must be compiled into the kernel, for instance. Allowing drivers to be distributed as loadable modules helps to limit the size of a deployed kernel as unneeded drivers are simply left out. It may also save memory since the modules may be unloaded when they are not required.

Modern versions of the kernel are designed to be highly portable. The kernel code is divided into core operating system code and platform-specific code. Platform-specific code is further divided into a core architecture and sub-architecture parts. The latter is know in Linux terminology as boards and can be found in a subdirectory of an architecture (also named */boards*). This modularity simplifies the process of porting Linux to new architectures and is certainly a key reason for its popularity among systems ranging from large multiprocessor mainframes to small embedded systems.

2.3.3 Cross-Platform Development

When developing software for an embedded system, one is often faced with the situation where the target platform and the host platform are not binary compatible. The embedded system for which software is being developed is known as the target platform, the workstation on which the software is created is the host platform. Binary compatibility means that the same software binary files may be executed on both. This was the case with this project where the host systems were Intel x86 compatible systems (also referred to as i386) and the target was based on a Renesas SuperH, specifically a SH-4. In order to produce executable binary programs for one platform on another, one requires software tools known as cross-platform development tools, sometimes also called cross-development tools.

The cross-development tools used for this project is based on the GNU toolchain, avail-

able in source form from the Free Software Foundation (FSF)[6]. The toolchain consists of the binary utilities, distributed in the *binutils* package, the compiler, *gcc* and the standard libraries. The binary utilities include programs like the linker *ld*, assembler *as* and other programs to manipulate binary and object files. The compiler, *gcc*, known as the GNU Compiler Collection contains a C compiler as well as support for several other languages including C++. Instead of using the GNU C-library[7], glibc, this project used uClibc. uClibc is a standard library specifically aimed at embedded systems. Whereas glibc would have required more space on the target system than available, uClibc consumes less than a megabyte.

The toolchain for this project was created with the aid of an automated system known as Buildroot. While the toolchain for this project was not built 'by hand', it is still enlightening to give an overview of the steps necessary to build a cross-development toolchain on the host[8].

- Binary utilities are compiled with the host system's compiler. The *TARGET* variable is set to the target architecture. In this case it would be set to *sh* for the SuperH architecture. This builds binary utilities that run on the host system and manipulate binaries intended for the target system.

- A stripped-down version of the compiler is built, know as the bootstrap compiler. This will be a cross-compiler, in other words it will run on the host system and produce object files for the target system. Only the C language will be supported. The function of this compiler is to compile the standard libraries which are necessary to build the final compiler.

- The kernel headers are generated by configuring a Linux kernel for the target system. The headers are necessary to build the standard libraries as well as the final compiler.

- The standard libraries are compiled with the bootstrap compiler. The libraries are built against the kernel headers prepared in the previous step. Both static libraries and dynamic libraries are produced.

- The final cross-compiler is built. It is build against the system headers produced by building the standard libraries. This compiler may have support for other languages like C++.

The cross compilation toolchain is usually not installed alongside the host's native compiler. The main reason for this is that it is often useful to have several cross compilation toolchains and installing one in the system path could cause unnecessary complications when

---

[6] ftp://ftp.gnu.org/gnu

[7] glibc is somewhat mistakenly referred to as the C-library as it is a collection of standard libraries including the C-library.

[8] It is interesting to note that it is possible to build a toolchain on yet a third platform that is neither compatible with the host or the target. This is however far beyond the scope of this discussion.

calling a specific one. The tools are distinguished from the native tools by a prefix. The tools used in this project used the prefix *sh4-linux-*. The compiler, for instance, is named *sh4-linux-gcc*.

2.3.4 Linux Device Drivers

The purpose of the hardware device drivers is to allow programs to access hardware devices without needing intimate knowledge of how these devices function. Device drivers accomplish this task by presenting the device through a well-defined interface that is not specific to the device. In the Unix paradigm that Linux follows, this interface is often a file-like interface.

Driver types

Devices can fall into one of three groups in Linux. These are the character devices, block devices and network interfaces, each with its corresponding class of driver. It should be noted that a driver does not have to fit neatly into one of the above types. It is entirely possible for a single driver to implement, for instance, both a network interface and a character device to access its hardware. Some drivers use a layered approach where only functions are exported to the kernel symbol table so that other drivers may be stacked upon it. An example would be the USB driver stack where several layers are used to implement different USB host adaptors, core functionality and endpoints.

Character and block devices are represented by special files known as a device nodes. These are usually located in the file system under the */dev* directory, but this is purely convention and is not in any way enforced by the kernel. Through these special files, user space programs may gain access to the driver, and therefore to the device it represents. It is quite often not even necessary for a user space process to know whether it is addressing a hardware device or a regular file on the file system. This is highly desirable as it allows much flexibility in how programs are applied.

Character devices are usually devices that can be represented by a stream of data. An example would be a serial port. These devices can often only be read from or written to in a sequential manner, without any provision for seeking (also known as indexing) into the stream. If the device allows for indexing it may however be implemented in the driver.

Block devices differ from character devices in that they represent a big block of memory that may be indexed at will. Hard drives are prime examples of block devices. To user space processes in Linux, block devices appear similar to character devices in that any number of bytes may be read or written. The implementation of block device drivers differ considerably from that of character devices. This is because these devices can only be written to or read from in complete blocks, often 512 bytes, at the hardware level. Block drivers have their own kernel interface that is concerned with the writing and reading of these blocks, while the kernel handles the translation to file-like read and write operations.

Network interfaces are unique in that they are not represented by a device node some-

where in the file system. This is because the way in which network interfaces are used is fundamentally different from that of character of block devices. Network interfaces are concerned with the transfer of data packets; the processes using them are usually interested in data streams over socket connections. The interface between network interface drivers and the kernel is concerned with the transmission of packets. The kernel's networking subsystem is responsible for the translation of these packets to socket connections and data streams. In the same manner that character and block devices are given unique node names, network interfaces are also given unique names to distinguish them, for example *eth0*.

Kernel space restrictions

Even though drivers contain code specific to the hardware for which they are written, there are many functions that are common to all drivers. All drivers, for instance, need to allocate memory for internal variables or register their services. Furthermore, unlike user space programs, kernel drivers cannot rely on the functions provided by the standard library, as those functions are implemented on top of the kernel. There exists a library of common functions written into the kernel that is called the Linux Kernel API [2]. It includes, amongst others, special versions of the string manipulation functions (*strcpy*, *strcmp*, etc.) and string printing function *printk* that behaves similarly to the standard library versions.

All kernel drivers run in superuser context (the highest privilege level) and therefore have access to all system resources. The driver programmer therefore has direct access to all kernel data structures; knowledge of these structures should, however, not be used to access kernel data directly because most structures in the kernel isn't guaranteed to be stable. This means that a structure may change in future versions of the kernel, but perhaps more importantly amongst different architectures. To promote stability over kernel versions, and when porting to new architectures, many structures have macro functions associated with them to access data on behalf of the driver.

A further restriction imposed on drivers are that they are required to be re-entrant. Because a kernel may run on systems with more than one CPU, there can be no guarantee that the same piece of code will not be executing in different contexts simultaneously. Even on single-processor systems this cannot be guaranteed because the kernel may preemptively reschedule the CPU while executing in kernel space. If concurrent execution of a piece of code may lead to a race condition, mutual exclusion techniques must be employed. These include among others the semaphore operations provided by the kernel.

Character drivers

As explained above, character drivers provide a interface to user space applications that is similar to that of a regular file. The kernel provides a very thin layer between the system calls from user space and the functions implemented in the driver code. The functions of that layer include tasks like checking whether the calling process has the right permissions to access the device node.

Whenever a file (a device node is a file) is opened, the kernel creates a *file* structure. This structure is passed to the driver whenever a call to driver is made. It contains two fields that are important to this discussion. The first is the *private_data*; this is a pointer that the kernel reserves for the driver. The driver may store any information in the field that it requires. If a device driver serves many physical devices, this field allows the driver to differentiate between them. The other field of interest here is *f_op*, which is of type *struct file_operations*. This structure contains function pointers to operations that may be performed on the file, in the case of the driver it will contain pointers to driver code. Other fields include the current file position, flags indicating for example blocking operation or read-only, and many others. Because the kernel creates the structure, its exact form is not that important to the driver writer.

When the device driver is initialised, it has to pass a *file_operations* structure to the kernel to tell the kernel which operations it can perform. If a certain operation is not supported, a value of NULL is returned. Depending on the function, the kernel may substitute an appropriate action for a NULL value or else refuse when the operation is requested from user space. It is noteworthy that this substitution of functions is similar to the object-oriented approach of inheritance.

The following is a list of the important file operations that the driver may define:

open The *open* function is called when the device node is opened by a user space process. The driver will likely use this function to initialise the device. If not defined by the driver, the kernel will not notify it when the device is opened.

release Called when the device node is released, in other words after all copies of the node are closed.

read and **write** The *read* and *write* operations are used to transfer data buffers between user space and kernel space. The direction is defined from the user space side. A read therefore passes information from kernel space (the driver) to the user space application.

poll Used for non-blocking IO. The kernel calls this function to ascertain whether a *read* or *write* would block. In other words, the function is used to tell the kernel that data is available for reading, or that the device is ready to accept data. This function also supplies the kernel with information on how to wait (sleep) until the device is ready.

ioctl This function is generally used for controlling the device. Control data that cannot easily be integrated into the *read* and *write* paradigm is passed from user space to the driver with the *ioctl* call. An example of its use is the setting of a serial port's baud rate.

mmap This is used to map a region of a device's memory into a user space program's memory space.

There are several other operations that may be defined. These are, however, the most important operations for this discussion. For a complete list as well as implementation details, the reader is referred to [12].

Chapter 3

High-level design

3.1 Overview

In this chapter the structural design of the software components for the experimental payload will be discussed. This chapter starts by providing a background of the experiments and their hardware. The requirements of the experiments are then discussed. Finally the design considerations are expressed and a design proposed.

3.2 Background

3.2.1 Experimental Payload Hardware Components

This section aims to create the context wherein the requirements of the design can be expressed. The following is an overview of the hardware that constitutes the experimental payload. It is hoped that knowledge of the hardware will help clarify the purpose of the software with which this project is concerned.

The experimental payload is housed in an aluminium enclosure that is bolted into place inside the main satellite body. The enclosure consists of two stacked trays, with an aluminium lid on top. This results in two enclosed spaces in which the experimental electronics can be mounted. The purpose of the trays are to provide mechanical support for the circuit boards within, as well as to prove shielding against radiation. One of the trays houses a single-board computer and daughterboard. The other tray houses a special version of SunSpace's VHF UHF Communication Unit (VUCU).

The single-board computer, also procured from SunSpace, is of the same design as the one used for the satellite's main on-board computer (OBC). The SunSpace OBC was chosen to simplify the hardware design because it would, by definition, already be compliant with SunSpace's electrical and mechanical requirements. The availability of an expansion port for a daughterboard meant that the additional electronics required for the experiments could be connected to it fairly easily. The software running on the OBC's main processor was developed as part of this thesis and will be discussed in the following chapters.

As mentioned above, the daughterboard connected to the OBC contains the components needed to execute the experiments. The daughterboard is based on the SU SDR group's Signal Interface Card (SIC), which is a general-purpose interface card. Extra components were added to fulfil the specialised functions of the experiments [32]. The firmware that controls this board was developed as part of this thesis. This board will therefore be discussed in more detail in later sections.

The VUCU was designed by SunSpace as their primary control conduit to the satellite. A VUCU on the ground station would communicate with a VUCU on the satellite the relay commands and telemetry information between the satellite and the ground station. As the name suggests, it is capable of operating in the UHF and VHF parts of the radio spectrum and could therefore be used with the allocated amateur band frequencies. The frequencies assigned to the experimental payload were 144.1-148.0 MHz for the downlink and 435.0-438.0 MHz for the uplink [18]. Following the same rationale as for the OBC, the VUCU was a logical choice as an front-end for the AMSAT and SDR experiments. Furthermore, it would also be able to act as a backup for the satellite's main communication unit.

The VUCU used in the experimental payload had to be adapted to accommodate the two experiments. Normally a VUCU is essentially used as an interface between the satellite's main communication bus and RF. It was adapted to provide intermediate frequency (IF) access, to the transmission path, and radio frequency (RF) access, to the reception path, of the SDR experiment. A mode was therefore added to allow it to act as transmission and reception amplifier for the SDR experiment. A local oscillator (LO) signal was also tapped and provided to the SDR experiment to help simplify hardware development. Furthermore, it supplies power to the AMSAT experiment. Lastly, audio frequency (AF) access was provided to the AMSAT and the SDR experiment.

The experimental payload has several external appendages mounted to the outside of the satellite. These are the uplink and downlink antennas, sensors for the KZN experiment and an enclosure housing the string experiment's string and its sensors.

The systems of the satellite are connected through a central controller area network (CAN) bus. All commands and telemetry data are transported over this bus. The experimental payload's OBC and VUCU are both connected to this bus. The preferred CAN-bus data rate for the experimental payload is less than 10 kB/s [18]; it is therefore not well suited for bulk transfer of experimental data.

Experimental data that is to be transmitted back to earth, first has to be transported to a 24 GB central store before being transmitted over the satellite's wide-band downlink. A unidirectional LVDS bus provides the conduit between the OBC and the central store. Data rates of up to 1 Mb/s is feasible on this bus [18].

The OBC and VUCU are fed by separate power nodes from the satellite's power distribution system. They can be controlled independently by satellite's main OBC.

3.2.2 Experiments

The experiments hosted on the experimental payload and their respective goals have been described in the introduction. The following section will explain what functions they require from the experimental payload.

Radiation Experiment

The radiation experiment is designed to measure two of the known effects of radiation on metal oxide semiconductor field effect transistors (MOSFETs). The first is a shift in the threshold voltage. Three test transistors are used for this measurement; one is always biased when the experiment is running, one is intermittently biased by a slow oscillator running at about 10 Hz at a duty cycle of about 50%, and the last one is only biased while measurements are being taken.

The other effect to be measured is the increase in bias current with the increase of radiation dosage. For this measurement the currents being supplied to the daughterboard's FPGA are measured. Separate readings are taken for the FPGA logic core and the input/output periphery. The current sensors provide a voltage that is proportional to the measured current.

All the signals mentioned above are multiplexed by an analogue multiplexer to provide a single voltage that has to be measured by a sampling system on the experimental payload. Signals to control the power to the experiment and to control the analogue multiplexer is also required—these signals are simple binary logic.

String experiment

The string experiment consists of a tensioned string, with sensors to measure its deflection and tension. Two sensors measure the deflection in the x and y direction respectively, and a strain gauge measures the tension in the string. A small actuator is attached to the string to excite it.

The experiment requires that all three signals be sampled at once. A signal also has to be provided to drive the actuator, and a sinusoidal wavefrom with adjustable frequency is required so that measurements can be taken at frequencies from below to above resonance. A signal ranging from 0 to 1 kHz is expected to be more than sufficient as the resonance of the string is expected to be far below this. Lastly, a binary signal is required to control power to the experiment.

Magnetosphere experiment

The Kwazulu-Natal experiment's measurements of the magnetosphere comprises vector measurement of the magnetic field and a scalar measurement of the electric field strength. This is accomplished by three current loops positioned perpendicular to each other so that they might measure the magnetic field strength in the x, y and z directions respectively. Electric

field strength is measured by suspending a conductive disc above the aluminium body of the satellite and measuring the potential difference between the two conductors. By factoring in the distance between the conductors, an indication of the E-field strength around the satellite can be estimated.

The experiment requires that all four signals be sampled simultaneously. The highest frequency of interest is at 30 kHz. This experiment also requires a binary power control signal.

SA AMSAT

The AMSAT experiment was designed by SA AMSAT to function independently from the experimental payload. When powered, it waits for certain baseband tones to be received over the VUCU and responds by sending baseband signals back over the VUCU. Power is supplied from the VUCU, it can therefore operate without any intervention from the experimental payload.

The AMSAT unit is controlled by a micro-controller. This micro-controller has a serial programming interface. It was originally requested that provision be made to allow for in-flight reprogramming of the micro-controller by the experimental payload. Binary input and output to the programming port is therefore needed.

SDR experiment

The SDR experiment's main requirement is access to sample streams to enable reception and transmission of modulated baseband signals. In addition to those facilities, a signal switching network was also devised that would allow the signals from the DACs to be routed, through the analogue reconstruction filters, back to the ADCs. In theory, this would allow imbalances in the filters to be characterised so that the imbalances can be corrected in software.

3.2.3 Daughterboard

The specialised hardware components required by the experiments have been discussed in the previous section. The rest of the hardware that constitutes the daughterboard will now be discussed.

SIC Background

As stated previously, the bulk of the daughterboard consists of a design, conceived by the SU SDR group, named the signal interface card (SIC). The design of the SIC was only first specified at the start of this project. The concept of the SIC itself was therefore also still under development during the course of this project. The SIC is not a physical piece of hardware, it was conceived to be a design that can be easily implemented whenever the group needed to convert signals from the analogue to digital domain. The SIC design comprises a basic circuit board layout as well as the accompanying software to make it work. This

software is in the form of firmware, residing on the SIC itself, and drivers that allow the host system to communicate with it.

In order to promote portability, both the hardware and software components would be designed as modules. The modules that manage the communication between the SIC and the host system would be specific to that instance. When porting to new host interfaces, only the affected modules would need to be addressed.

An example of a possible future application of the SIC design is a general purpose signal interface for a workstation computer. To accomplish this goal the circuit board layout would first be moved to a expansion card, possibly a Peripheral Component Interconnect (PCI) card. The parts of the circuit that interfaces with the host system bus would be an example of the instance specific parts of the SIC that would have to be redesigned for this application. The rest of the design would then be reused as is. As with the hardware, parts of the software would fall in the instance specific category and would need to be redesigned. The firmware on the FPGA is an example of this. The host side software drivers would likely also need to be adapted to function with the new bus interface.

The advantage envisioned by using the SIC is that, even though new development would be necessary for new applications, the core functions of the SIC would remain essentially the same. Importantly, the software interface would remain the same because the same drivers would be used, possibly with a changed hardware interfacing back-end. This, it is hoped, will help make SU SDR applications rapidly portable to new architectures.

SIC Specification

The first specification of the SIC is outlined here. Future versions of the SIC will undoubtedly add to and refine this specification. This specification looks at the inputs and outputs provided by the SIC. Internally the SIC will be controlled by an FPGA, the firmware of which was developed as part of this thesis.

Primary converters. As the name implies, the SIC is primarily a signal interfacing device. Designed by the SU SDR, its primary envisioned function is as part of a hardware front-end for SDR. Direct conversion front-ends are employed in both transmission and reception paths. This choice is motivated by the relatively simple hareware requirements and the lower samplerate. It therefore has two main DACs and two main ADCs, as well as the accompanying quadrature modulator and demodulator. The local oscillators driving the mixers can be adjustable, in which case their control will be handled by the general-purpose IO subsystem (discussed below).

Neither the signal bandwidth nor the band of interest is specified. These parameters will be dictated by the requirements of the system in which the SIC is employed. As a result, the conversion rate and local oscillator frequencies are not specified as part of the SIC architecture specification. Nor are the specifications for the reconstruction filters. In order to promote portability, the software should therefore not make any assumptions about these parameters.

Peripheral converters. Besides the main converters, there are two sets of general-purpose ADC inputs. These are implemented as a single bank of four ADCs that can be sampled synchronously. An analogue multiplexer in front of these converters allows a controlling application to take samples from either of the input sets, designated peripheral bank A and bank B. Samples can therefore not be taken from both bank A and B simultaneously.

General purpose IO. In addition to the abovementioned analogue signals, the SIC also makes provision for general-purpose control signals in the form of logic-level inputs and outputs. These signals may be unidirectional or bi-directional as the application requires. The exact number of signals will be dictated by the application and the available resources on the FPGA.

Host Interface Port. The signals described above constitute the core functions of the SIC. In addition to those, an instance-specific host interface port will have to be implemented for each intended host architecture that the SIC is to be connected to. This interface is expected to generally take the form of some bus interface found on the host system.

Daughterboard connections

The primary component of the daughterboard is the SIC. It is responsible for interfacing with the host system and converting the analog signals from the experiments into the digital domain. This section will complete the description of the daughterboard by listing the connections between the SIC and the various experiments (see Fig. 3.1).

Power is supplied from the host system. It is distributed to the various experiments on the daughterboard by a set of digitally controlled power switches. Each of these switches is connected to one of the SICs general purpose outputs.

The radiation experiment. The analogue multiplexor's control inputs are connected to four general-purpose outputs. The analogue output from the experiment is connected to the fourth input of peripheral bank B (B4). There are two power switches involved with this experiment. The first one is a dedicated power switch to allow the designated devices-under-test to be powered for controlled period's while the experimental payload is active. The other switch is the one that powers the peripheral sampling system. It is used to power the device that only gets biased while measurements are taken.

String experiment. The three inputs from the string experiment (x, y deflection and string tension) are connected to the remaining inputs on peripheral bank B (B1-3). The excitation signal is generated by the I channel of the primary DAC. SDR transmission and the string experiment will therefore not be able run concurrently. There is a power switch dedicated to this experiment that powers the actuator and the sensors in the string unit.

Magnetosphere experiment. The three signals from the magnetic field sensors (x, y and z) are connected to peripheral bank A's first three inputs (A1-3). The E-probe is connected to the last input on this bank (A4). The experiment has its own power switch to power the amplifiers and filters that condition the sensor inputs.

SA AMSAT. As explained previously, this experiment can function without intervention

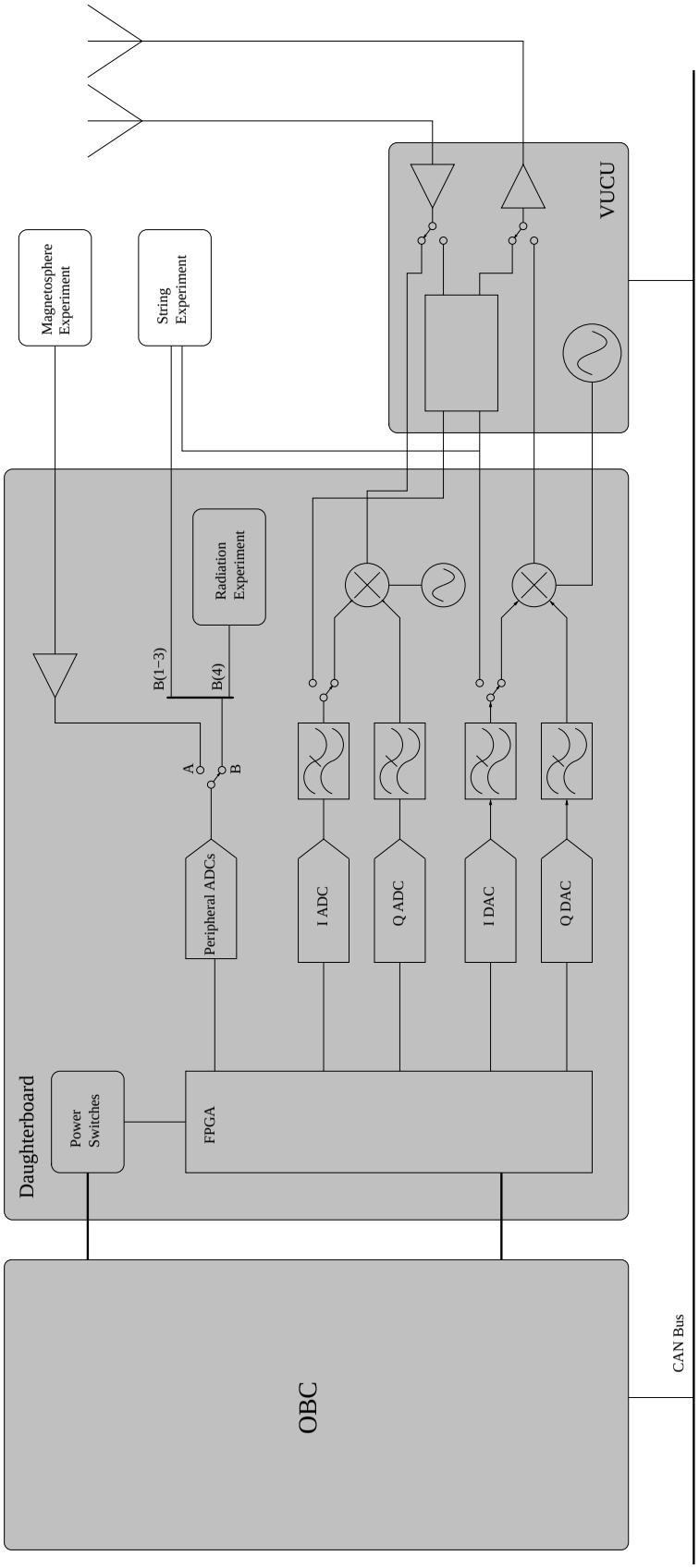


Figure 3.1: *Block diagram of experimental payload with important connections.*

from the daughterboard systems. The programming port of the micro-controller is connected to five general purpose IO lines.

SDR experiment. As the SIC was developed with SDR experiments in mind, the hardware concerned with the this experiment can be considered part of the SIC. The switching network between the converters and the mixers can be seen as a specialisation of the SIC that employs general purpose outputs.

3.2.4 OBC

As described above, the single-board computer used in the experimental-payload is the SunSpace OBC. The board has several peripherals that are of interest to the experimental payload. These can be seen in figure 3.2.

The following is a description of roles these peripheral components were envisioned to fulfil in the experimental payload:

NAND Flash. Experiments may require data to be stored in between runs. The NAND flash can be used as non-volatile storage for these experiments.

NOR Flash banks. The operating system and permanent application software would reside here. It can be reprogrammed over the CAN bus, without intervention from the SH4 processor.

EDAC SRAM. The error detection and correction (EDAC) static random access memory (SRAM) could be used as primary volatile memory for applications and the operating system running on the experimental payload. The size of this memory is 8 MB.

SDRAM. The size of the protected memory is limited. The synchronous dynamic random access memory (SDRAM) could also be considered as primary memory. This size of this memory is 64 MB.

CAN nodes. As explained above, all the satellite subsystems communicate through a central CAN bus. These nodes provide an interface to that bus. There are two CAN nodes on the OBC.

SH4 Processor. The central processing unit (CPU) used for the experimental payload.

LVDS Transmitter. As explained above, the CAN bus is not suitable for bulk transfer of experimental data. The low-voltage differential signalling (LVDS) transmitter provides a conduit for sending experimental data to the satellite's main storage unit. From there data can be transmitted to earth.

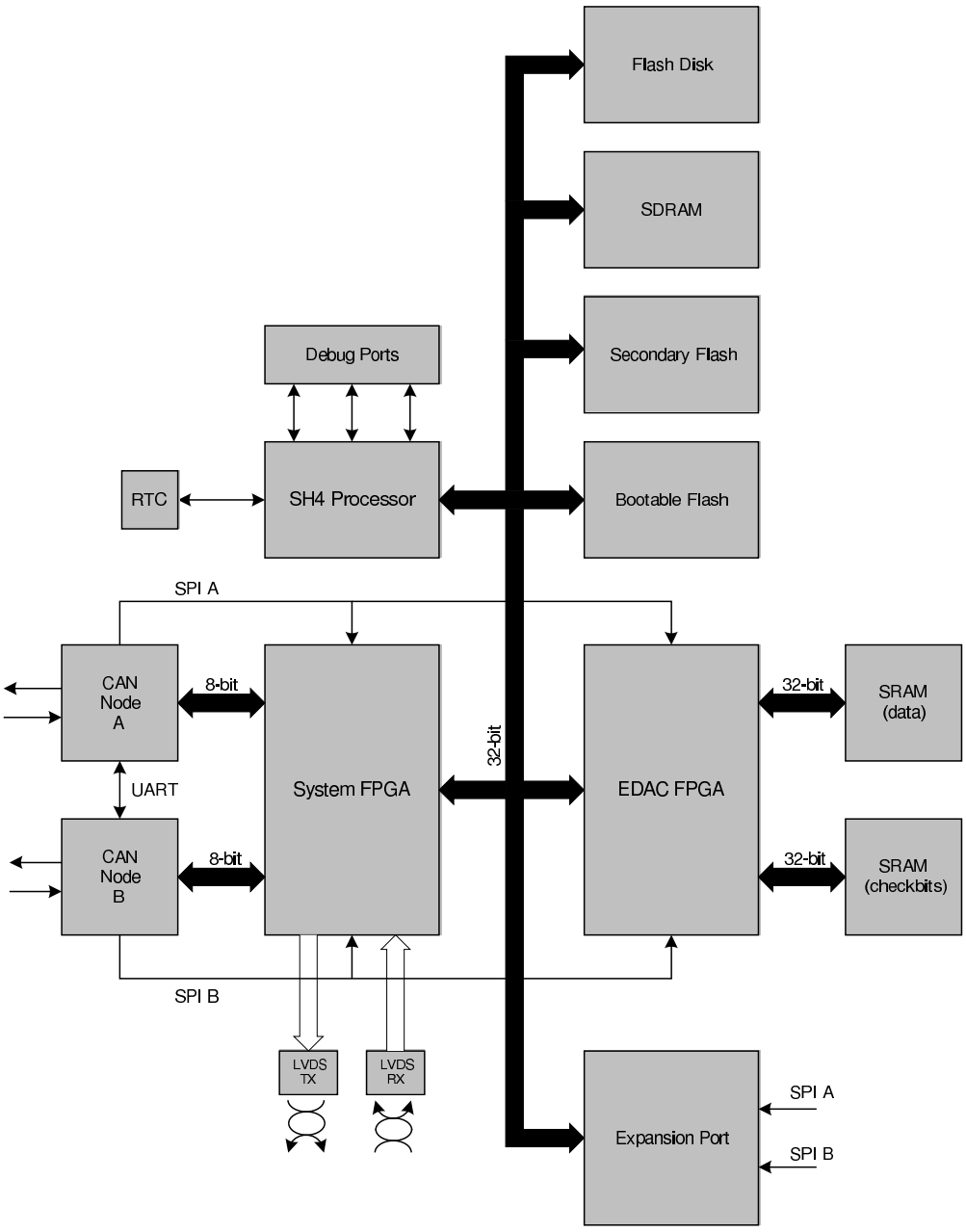


Figure 3.2: *Block diagram of SH4 OBC.*

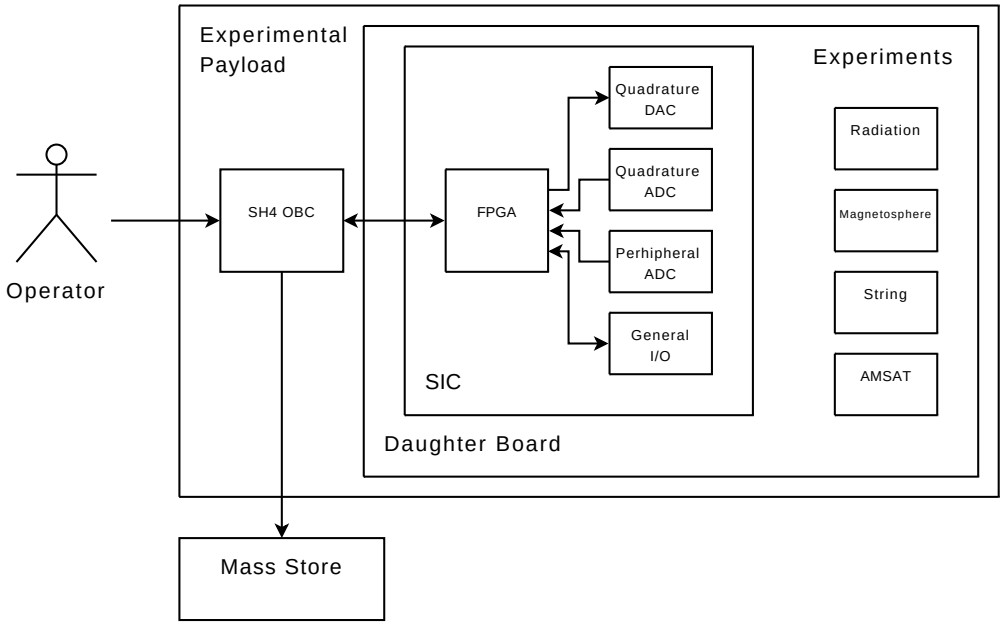


Figure 3.3: *Block diagram of supplied components.*

3.3 Requirements Analysis

3.3.1 Design Methodology

The hardware components of the experimental payload have been designed or acquired outside of this project and was briefly discussed in the previous sections. In this section the design of the software components for the OBC as well as the daughterboard will be discussed. These components, which are required to complete the experimental payload, will consist of firmware for the daughterboard and a complete operating system with user space applications.

In order to evaluate the required behaviour of the system, a bottom-up approach is taken. This is necessary because the design of the system will, in part, be dictated by the components already chosen outside of this project.

- First the complete system, as seen by the software, will be decomposed into functional blocks. The software components will be treated as black boxes at this stage.
- Secondly the desired behaviour of the system will be discussed. Both functional and non-functional requirements will be discussed.
- Finally a design will be proposed that attempts to address the requirements.

3.3.2 Functional Decomposition

The following discussion is based on the functional representation in figure 3.3. The functional blocks identified as being of interest to the software design are the following:

Operator. This block represents the command structure of the satellite. Any device connected to the satellite's CAN bus could qualify, but it is expected that only ground station operators or the satellite's main OBC will send requests to the system.

Mass Store. The satellite's primary storage device. Experimental data is temporarily stored here before being transmitted to earth.

SH4 OBC. This represents the single-board computer supplied by SunSpace. For the purpose of this decomposition, this is essentially a placeholder for the software components running on the OBC.

FPGA. The FPGA on the daughterboard. This represents all functionality supplied by the firmware.

General-purpose IO. All components that are directly connected to the SIC's general purpose IOs. These include all the power switches and other experimental hardware discussed previously.

Quadrature DACs and ADCs . Primary SIC signal outputs, intended for use with SDR experiments. Also used by the string experiment.

Peripheral ADCs. Auxiliary SIC signal inputs. These are used to sample the experiments' data.

The functional behaviour of the software components are, in part, dictated by the interfaces they have to conform to. The following interfaces were identified to have a direct impact on the software components:

CAN Bus. The primary communication conduit in the satellite.

LVDS. SunSpace's data transfer protocol, it is mostly implemented in hardware on the OBC.

SH4 BUS. OBC's primary bus. SH4 processor's data bus. All peripherals connected to the SH4 processor are connected to this bus.

GPIO. Low-voltage, transistor-transistor logic (LV TTL) logic levels. These may includes inputs and outputs as well as bi-directional signals.

DAC. Two serial data streams. The clock driving the conversion process also has to be supplied by the FPGA.

ADC. Two serial data streams. The clock driving the conversion process also has to be supplied by the FPGA.

P.DAC. One to four serial data streams. The clock driving the conversion process also has to be supplied by the FPGA.

3.3.3 Required behaviour

Use cases

It was requested, by the project management, that sampling of data and transmission of the data be separate events. It was therefore necessary to implement some form of local storage on the OBC. An additional functional block, local memory, is therefore added (to the list given in section 3.3.2) for the purpose of this discussion. The integration team also requested that an option be added to allow captured data to be transmitted over the CAN bus. This was used to conduct integration testing using workstations, running SunSpace Lua Command Interface (LCI) sessions over CAN, that did not have LVDS interfaces. An LCI session interfaces with a device called the Ground Support Equipment Ethernet Interface (GSE-EI), which in turn interfaces with the CAN bus. Finally, the integration team also requested that some form of low-level access be provided to the components on the daughterboard—specifically the ability to access registers on the SIC.

The AMSAT payload is designed to operate without intervention from the rest of the experimental payload and was therefore not considered during the software design. The SDR experiment does not have an easily definable use case; this is due to the reconfigurable nature of SDR. From the discussion of SDR (in chapter two) it can be deduced that the SDR experiment will require access to data streams from the quadrature converters and a POSIX platform on which to run. A non-volatile file system will also be required to store the SDR software components.

Based on the above, as well as the discussion of the experimental payload's components (see section 3.2.1), a list of requirement for the software can now be defined. From the perspective of the software, all the experiments share a set of common requirements. These are the following:

- Switching power switches on the daughterboard.
- Acquiring samples from the peripheral sampling system and temporarity storing them on the OBC.
- Sending those samples to an LCI session, using the CAN bus.
- Sending the samples to the satellites mass store, using the LVDS transmitter on the OBC.

In addition to the above, the String experiment requires a facility to adjust the parameter of its excitation signal and the Radiation experiment requires a facility to adjust the setting

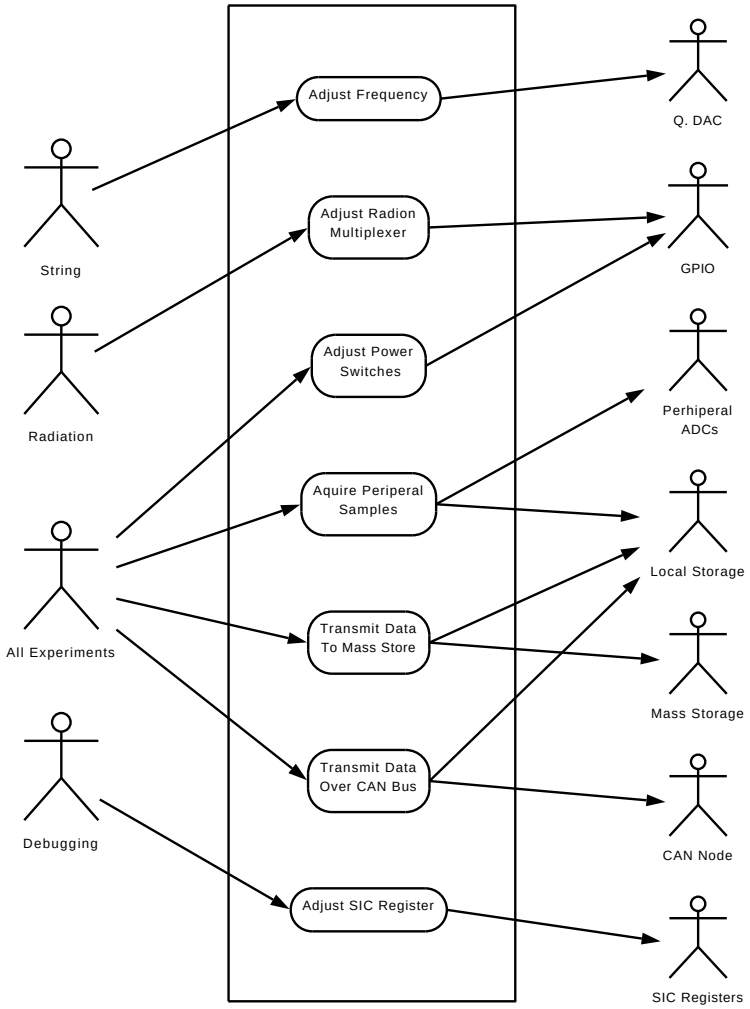


Figure 3.4: *Use case diagram for experiments.*

of its input multiplexor. The required behaviour of the system is expressed in a use case diagram (Fig. 3.4).

Non functional requirements

In addition to performing the required functions, the following desireable features were also identified:

Reusability. The SDR Group's ultimate goal for the SIC is a reusable platform, this goal has to be considered when making design decisions.

Reliability. Once launched, no physical intervention will be possible; reprogramming the FPGA will therefore not be possible.

The reusability consideration dictated that the SIC's design should be made modular, so that parts of the system could be easily replaced if necessary without having to change

the whole. Reliance on vendor specific feature should also be avoided as much as possible; when they have to be used, a well defined interface should be in place so that they may be reimplemented in other systems.

In order to improve the reliability of the system, it should be attempted to simplify its components as much as possible. Implementing functionality on the reprogrammable part of the system (OBC) should take preference in order to shift complexity away from the non-reprogrammable (FPGA) part.

3.4 System Design

3.4.1 Firmware

The primary purpose of the firmware is to enable the controlling OBC board to access the hardware on the SIC daughter board. A secondary goal of the firmware is to assist the CPU on the OBC to carry out signal processing tasks.

The firmware should be designed in a modular way to simplify porting of the SIC to other platforms. If this goal is met, it should be possible to port the SIC to another platform by simply replacing the parts that interface to the host. This should simplify porting by limiting the number of changes to the code base in both the firmware as well as the software drivers.

In order for the system to execute a certain function, a certain amount of complexity is inevitable. This complexity has to be partitioned between the SIC and the host. It is also frequently true that with increased complexity the number of latent bugs increase. Therefore, in order to make the system as robust as possible, the structure of the firmware should be kept as simple as possible, leaving more complicated operations to the controlling system and its drivers. This is dictated by the constraint that the firmware cannot be changed after the satellite is launched. By shifting more complexity into the host system, it is hoped that any latent bugs discovered after launch may be addressed. The following is an overview of the design, identifying the main components and their interconnections.

Each of the four interfaces of the SIC (see section 3.2.3) has a subsystem dedicated to it in the firmware; the host interface is not counted here as it will form an integral part of the whole system. These subsystems operate independently from each other. This is done in the hope that it will increase robustness, as failures in one system are less likely to affect others. Communication between the OBC and daughterboard should be stateless as far as these subsystems are concerned. In other words, a request or command from the host should be an atomic instruction. This will help keep complexity of both drivers and firmware to a minimum. The conceptual layout of the firmware is depicted in figure 3.5.

Quadrature DAC

This subsystem is responsible for transfer of samples from the host OBC to the DACs on the daughterboard. Samples will be buffered on the FPGA to help smooth jitter created by the

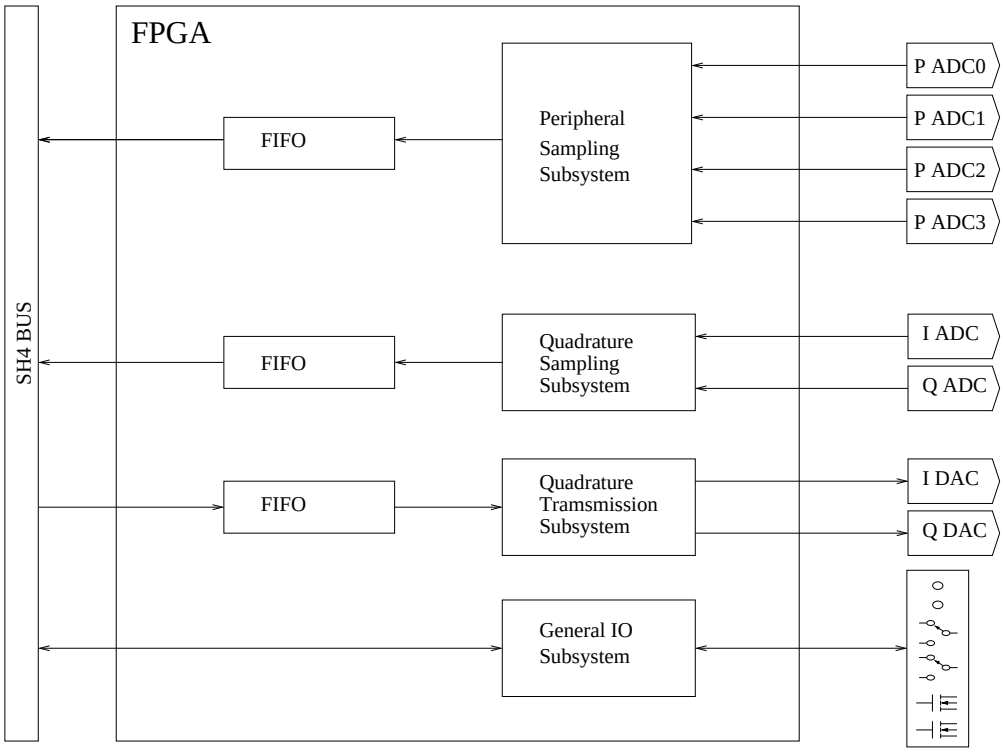


Figure 3.5: *Top-level block diagram of firmware.*

host system and to alleviate the overhead associated with data transfer over the processor's data bus.

Because of early concerns about the SH4 system's ability to generate samples at the specified sample rate, it was decided to incorporate a facility on the FPGA that could generate tones for the String experiment. This device, called a numerically controlled oscillator (NCO), would serve as a backup in the event that samples could not be generated on the OBC. The frequency and amplitude of the NCO should be adjustable for the purposes of the experiment; this meant that the NCO could also be used for direct digital synthesis in SDR experiments. Therefore phase and DC-offset adjustment was also added to the NCO and two of them were instantiated on the FPGA—one for each channel.

Quadrature ADC

This subsystem is conceptually just the inverse of the quadrature DAC subsystem. Samples should also be buffered to compensate for jitter in the host's consumption rate.

Peripheral ADC

This subsystem is responsible for transfer of samples from the peripheral ADCs to the host system. Buffering is again provided on the FPGA. Sampling of the four ADCs will always occur synchronously. A masking register controls which of the ADCs are active so that only

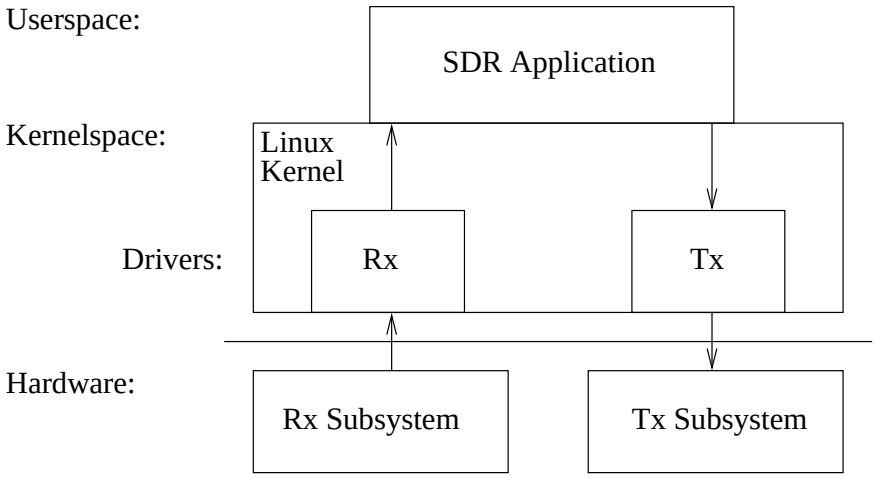


Figure 3.6: *Software components of SDR system.*

a subset of the bank may be sampled at once to conserve bus bandwidth when required.

General purpose IO

All general purpose outputs are implemented as registers in the host systems memory range. These registers will latch any data written to them by the host processor. The outputs of these registers will drive the GPIO outputs. Bidirectional IOs will have an extra register to control the direction of the GPIOs. Inputs will again be mapped to the host's memory range and be read back as register values.

3.4.2 OBC Software

As mentioned in section 2.3, the Linux operating system kernel was used on the OBC. This required that Linux device drivers be developed for the OBC's on-board peripherals and the SIC. The choice of kernel is discussed in section 5.2.1.

SIC drivers

The drivers for the SIC follow the usual Unix convention of exposing devices as files. Each subsystem of the SIC will be represented by a character device node. Additionally, device nodes for each register on the daughterboard are to be implemented to provide low-level register access to the SIC (see Fig. 3.7).

The register device nodes communicate to user space using a textual representation, in hexadecimal notation, of the register contents. This was chosen to simplify user intervention with the hardware. The standard Unix user space tools can therefore be used to communicate directly with the hardware. To write the value 127 (7F in hexadecimal), for instance, into the register named *x1*, the following command can be typed into a shell on the OBC:

```
echo 7f > /dev/x1
```

To read the contents of the same register, to following can be used:

```
dd if=/dev/x1 count=1
```

The *dd* program is used because using *cat* would result in the register being read continuously. The *count=* parameter of *dd* allows the exact number of reads the be specified.

Drivers will be implemented to allow sample streams to be written to, or read from, the converters on the SIC. The interaction between these drivers and the SDR application is shown in Fig. 3.6. A proof-of-concept NCO interface driver will also be implemented to demonstrate the use of the NCO in SDR applications.

Root File system

The root file system should allow for a execution environment in which SDR applications can be executed. This will include all the necessary user space programs and libraries to establish a software environment in which experimental application may be executed. Furthermore, a method for updating this environment must be devised; the experimental software components must be replaceable without necessitating that stable components be re-uploaded unnecessarily.

A ramdisk will be instantiated in the SDRAM to store sampled data temporarily. The rest of the filesystem can therefore be made read-only to prevent accidental damage to the root filesystem.

NAND Flash

Non-volatile storage should also be implemented. Future SDR applications may require storage space for their components. It should be possible to write to this store without requiring changes to the read-only parts of the file-system. A filesystem on the NAND flash will be implemented for this purpose, the Linux Memory Technology Devices (MTD) subsystem (see section 5.4.2) will be used to accomplish the goal.

CAN Driver

A device driver will be implemented to allow applications to transmit messages over the CAN bus.

In order to simplify the design of this driver, it was decided not to implement any parsing of the CAN messages in the driver code. Decoding of message identifiers are to be handled by a user space program, as programs running in user space are easier to write and debug. It was also decided to have the driver communicate to user space in hexadecimal text. This would allow the workings of the driver and the user space process to be easily observed and debugged without having to write more software, which would also need to be debugged.

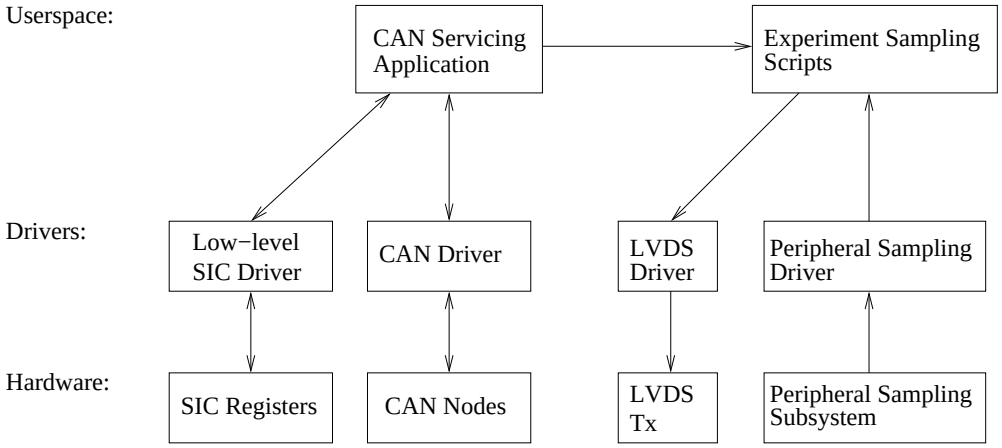


Figure 3.7: *Software components for experiments.*

The overhead of converting all messages to the text based representation (and back) would be offset by low frequency at which messages are expected to arrive.

The CAN servicing application will be responsible for decoding the CAN messages and taking appropriate actions. For simple requests it will access the daughterboard directly using the low-level driver, experimental data will be gathered by calling spawing subprocesses which in turn will access the daughter board through the SIC drivers. These interactions is depicted are Fig. 3.7.

Chapter 4

SIC Firmware Design

4.1 Introduction

The firmware can be divided into four subsystems that function essentially independent of each other. Dictated by the portability objective, these subsystems can also be partitioned into host interface and core function parts. The subsystems are depicted in Fig. 3.5 and have the following functions:

- The peripheral sampling subsystem is responsible for acquiring samples from the four peripheral ADCs and presenting them to the host system.
- The quadrature sampling subsystem is responsible for acquiring samples from the two quadrature ADCs and presenting them to the host system.
- The quadrature transmission subsystem is responsible for driving the two quadrature DACs. It either passes samples unchanged from the host to the DACs or interprets modulation commands from the host while performing direct digital synthesis (DDS) (see section 3.4.1).
- The general purpose input and output ports are primarily used for controlling on-board signal and power switches. Because the SIC was conceived and designed to be used as foundation for other systems, it also contains general purpose bidirectional ports. On the experimental payload, these were, for example, connected to the AMSAT payload's micro-controller.

4.2 Host Interface

4.2.1 Registers

Registers are the primary interface between the SIC and the host system. Their purpose is simply to store a single data word as received over the data-bus and directed to their unique address by the address-bus. The word stored in the register is available as an output port that may be read by any logic within the FPGA that requires it.

Porting

Because of their role as interfaces, their own interface to the rest of the SIC is abstracted and their implementation details are hidden from the rest of the system. This should simplify the process of porting to a new host system, since the interface between the registers and the rest of the SIC and would remain constant.

Width

All registers in this system are 32 bits wide since this is the bus width of the systems the SIC was designed to interface with. Generally 32-bit systems can address 8, 16 or 32-bit wide memory locations [12]. It was, however, decided not to implement this extra functionality as it did not present any appreciable gain. The increased complexity of decoding addresses with different word sizes would also increase footprint size of the registers. The unused bits of the 32-bit registers do not impose a footprint size penalty (when compared to 8 or 16 bit registers) since the unused logic will simply be discarded by synthesiser when the design is built.

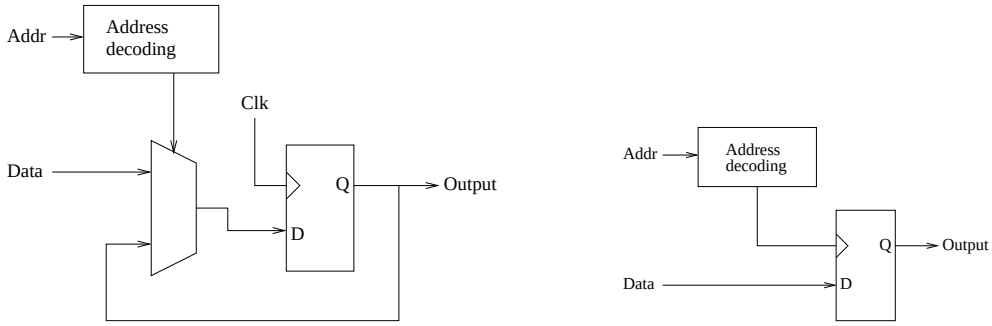
Architecture-specific implementation

The major concern identified during the implementation of the registers is the timely decoding of register addresses. A second concern was the footprint size of the registers: A separate project, which ran concurrently with this one, was charged with developing a sample rate conversion module for the daughterboard. This module would allow the OBC to operate at lower sample rates than the converters. The initial results of that project showed that its module would consume a large number of the daughterboard's FPGA blocks[1]. It was therefore decided to attempt limiting the footprint size of this projects component in order to preserve space of the sample rate converter.

The SH4's bus timings are controlled by a module (integrated into the CPU) called the bus state controller (BSC) [25, p. 317]. The BSC allows devices with different interface specifications to use the bus simultaneously. This is accomplished by partitioning the address space into 8 segments, each with its own chip-select line. The daughterboard is given segment 2 and the accompanying chip select line, *nCS2*; similarly, the EDAC SRAM's FPGA controller (see section 3.2.4) on the OBC has a dedicated segment. It was decided to configure the daughterboard's segment with the same parameters as that of the EDAC controller, since they are both based on ProAsic Plus FPGAs.

VHDL written for synthesis usually uses synchronous design methods with regard to a clock signal. This causes the synthesiser to place a multiplexer in front of each register (see figure 4.1(a)). The overhead incurred by this multiplexer is usually of little concern as it will be integrated into other logic by the optimising stage of synthesis [26]. Because of the

[1]The sample rate converter was ultimately too large, by itself, to fit on the daughterboard.



(a) Data gated register. (b) Clock gated register.

Figure 4.1: *Clock and data gating*

layout of the ProASIC Plus logic blocks, this results in two blocks being used for every bit in a register. A single 32-bit register therefore consumes in excess of 64 cells (some logic is required for address decoding), and it was decided to explore options for reducing the footprint of the registers.

The result was to define the register closer to the hardware level and to do away with the multiplexer. This requires controlling the clock input of the registers instead of the data input, a technique known as clock gating [26] Besides reducing the footprint by doing away with the multiplexer (see Fig. 4.1), it also has the advantage of slightly reducing power consumption by lowering the amount of logic that changes state at each clock cycle.

Clock gating has some notable drawbacks. Synthesis tools cannot guarantee clean clock signals and they cannot measure timing constraints accurately [26]. Careful attention is therefore necessary, as with all hand-written logic, to ensure that the clock signals are clean from glitches and that timing constraints are met.

In order to ensure that the clock input to the register blocks are glitch-free, a latch (see Fig. 4.2) was inserted in the clocking circuit for the SIC registers. The latch should ensure that a constant enable-signal is available during the write-cycle, so that glitches in the address decoding does not induce false triggers. This arrangement worked well during initial development. As the number of registers increased, the increased fan-out on the address lines resulted in the address latch not always having a valid input at the bus clock's rising edge. During later stages it was therefore found that false triggers did occur. The solution adopted at that stage was to instruct the BSC to add an additional, *address hold*, cycle. This allowed the address decoding enough time to settle before a write cycle. This unfortunately also slightly slowed down all the accesses to the FPGA. Because implementation of the SIC registers are completely abstracted from the rest of the firmware, it will be possible to address the problem in a more elegant way in the future by replacing the register implementation.

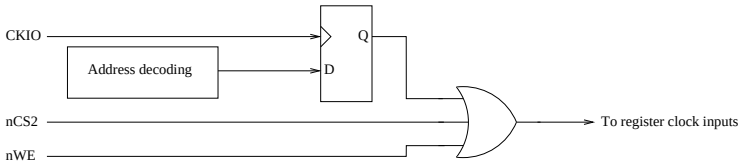


Figure 4.2: *Gated clock trigger circuit.*

4.2.2 Sample Buffering

Samples received over the host system's bus is buffered before being send out over the DACs. This is necessary to compensate for jitter arising from other devices contending for the bus. Buffering also helps reduces overhead, required on the host's side for sending samples over the bus, by grouping samples together and sending them as groups. Finally the buffers present an elegant solution to the problem of crossing different clock domains. The problem arises because the host bus operates at a clock rate that is asynchronous to the rate at which the converters consume samples. In the same manner, samples captured from the ADCs are buffered before being sent to the host.

Many FPGAs contain integrated RAM blocks that may be used to construct FIFO buffers. Since different FPGA models are bound to have different interfaces to their RAM, the sample buffers used in the SIC are abstracted by simple wrapper components. These wrapper components will hide the implementation details of their FPGA RAM blocks by keeping their interface to the outside constant. If internal memory is unavailable in the FPGA, external memory could be added and the wrapper components suitably updated to allow access to the external memory.

The ProASIC Plus FPGA used on the daughter board contains two port memory blocks that may be configured as either two-port RAM or FIFOs. The blocks are comprised of 256 words of memory, each 9 bits wide. The ninth bit may be configured as either a parity bit or a data bit. Larger memories may be constructed by stacking discrete blocks in both length and breadth. The Actel tools accompanying the ProASIC FPGA can generate VHDL code that manages the stacking of these blocks.

The FPGA used on the daughterboard had 32 memory blocks that had to be divided amongst the IQ DACs, IQ ADCs and the four peripheral ADCs. Memory was allocated so that FIFO lengths would be in proportion to sample rate. Because the peripheral ADCs operate at about a tenth of the sample rate of the IQ converters, it would require that their FIFOs be a tenth of the length. The amount of memory available was, however, not sufficient to allow memory to be distributed in this manner. As a result the peripheral ADCs were allocated a single block length (256 words) and the remaining memory was distributed between the IQ converters.

4.2.3 Dataflow

Control registers

For the purpose of the daughterboard firmware, a control register will be defined as a register that allows the host to control the behaviour of the daughterboard. The control register is mapped into the host's memory space and can be written to by the host system. When the host writes to a control register, the information stays latched in the register until the host writes to it again or the system-wide reset signal is received. Internally the register is wired to the firmware components that it controls. Every subsystem on the daughterboard requires control registers, for example the register that controls which power distribution switches on the daughterboard are active.

To implement a control register, an instance of the register component's input is connected the host's bus; and the register's output port is connected to the VHDL signals that manipulate the downstream subsystem that the control register is to be in charge of.

Status registers

In this context, status registers are defined as registers that may be read by the host system. In contrast to the control registers, status information is not latched when the host reads from the register's memory address. The host simply receives a snapshot of the state at the moment the address is read (the information will, of course, be latched into a host CPU register where the host will interpret the contents).

While numerous logic elements may listen to the signals on the host bus (as is the case with all the control registers), only a single entity may output a signal to the bus at any time. In the case of the daughterboard, this requirement is implicitly enforced because all logic is implemented on a single FPGA and therefore share the same physical connection to the bus. There are therefore two separate concerns that need to be addressed when a status register is read by the host. Firstly the FPGA must only enable its output to the bus when it is instructed to do so by the host, in other words, when the host reads an address belonging to the daughterboard. The second is that the correct signal information must be output to the bus with regards to the address requested by the host.

Signal flow

Samples arriving from the host are received at a rate dependent on the clock of the host bus. Similarly, samples being sent to the DACs are clocked at the sample rate of the DACs. As noted in section 4.2.2, the crossing between these clock domains occur inside the FIFO buffers. Samples are written to and read from the FIFOs asynchronously.

From the host's perspective the FIFO appears as yet another memory location. Values written to this memory address are clocked into the FIFO. The same method used to decode the address of registers (see section 4.2.1) is used to trigger the input clock of the FIFO.

The DACs are controlled by a state machine that reads samples as parallel data words from the FIFOs and sends them as a serial data stream to the DACs. This state machine runs at the DAC's sample rate and clocks samples out of the FIFO at this rate.

To inform the host when new samples are required, a trigger in the FIFO buffer activates when data in the FIFO reaches a certain level. This level is adjustable by the host so that the effective length of the FIFOs may be fine tuned if desired.

The ADC's signal flow operates similarly to that of the DACs. Serial data streams are read from the converters and clocked into FIFOs as parallel data words. The host removes samples from the FIFO by reading a memory location assigned to them. In order to clock samples out of the FIFO, a state machine is implemented inside the FIFO wrapper components. This state machine detects when a read cycle occurs on the bus and then triggers its FIFO buffer if the correct bus address is also detected.

Interrupts and DMA

The host system may poll the status registers of the FIFOs to determine if they require servicing. In order to make this process more efficient the daughterboard may generate a hardware interrupt to notify the host that it requires service.

Interrupts for FIFOs are generated from their level triggers. Because the daughterboard only has one hardware interrupt line, the subsystems on the daughterboard have to share the resource. A special register is instantiated that contains the interrupt status of the whole board as a bit mask. This register therefore indicates which systems requires service. The host may clear selected entries in this register by writing a bit mask back to it, indicating which are to be cleared by setting the appropriate bits high. This allows several interrupt service routines running on the host to service the same interrupt signal.

As a further step, direct memory access (DMA) may be implemented to allow samples to be transmitted in the background without CPU intervention. This functionality was, however, not implemented in the experimental payload as its importance was not deemed high enough to spend the constrained resources on. It should be noted that implementation of this functionality would be highly dependent on the bus architecture to which the SIC connects. For example, minimal logic would be required in the case of the OBC, as the SH4 includes an integrated DMA engine that may be used by external hardware. On a PCI bus in a PC, the DMA engine would however have to be implemented on the FPGA.

4.3 Subsystems

4.3.1 Peripheral Sampling Subsystem

The peripheral sampling subsystem consists of two main components (see Fig. 4.2). The first is the peripheral FIFO wrapper components, and the other is the peripheral ADC controller.

Because of the ADC's 18 bit resolution, it was decided to dedicate a host address to each

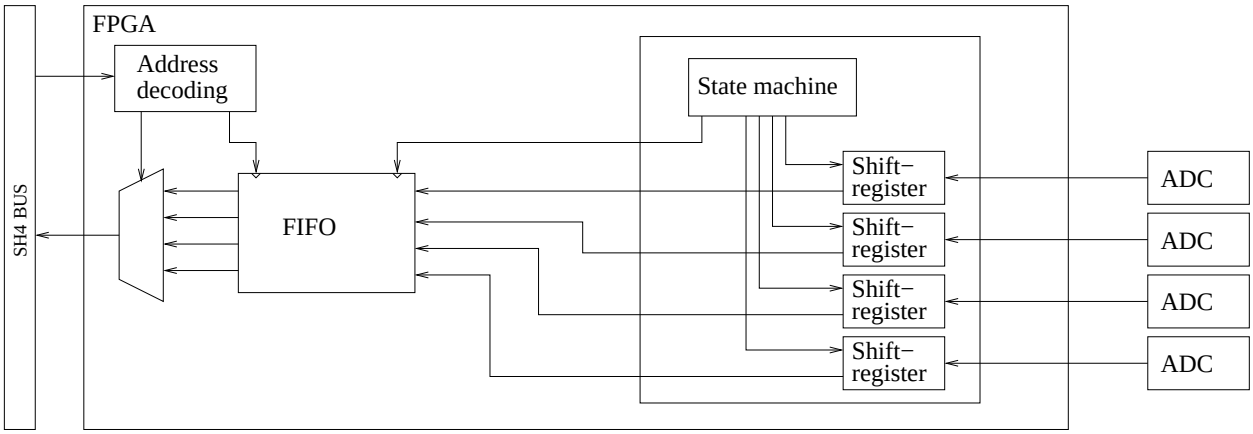


Figure 4.3: *Top-level design of the peripheral sampling subsystem.*

of the converters (32 bits). Reading the first address clocks a sample, for each, converter out of the FIFO. The remaining three addresses are similar to a status register in that no trigger is generated on reading from them.

The state machine controlling the ADCs is driven by a clock signal derived from dividing down the daughterboard's clock. While the ADCs can generate their own sampling clocks, it was decided to operated them in their external clocking mode. This assures that they operate synchronously and, according to the datasheet [19], also provides the best noise performance.

The state machine can be configured to only activate selected ADCs on the bank. This is accomplished by writing a bit mask, indicating which converters are to be active, into its control register.

4.3.2 Quadrature Sampling Subsystem

The quadrature sampling subsystem (see Fig. 4.4) has the same basic structure as the peripheral system. It also consists of a FIFO wrapper component and an ADC controller.

The two signals, from the I and Q signal paths respectively, are packed into a single 32-bit word. This was done to reduce the number of bus cycles required to transfer samples over the bus. Both samples are therefore read from a single host address. Each read to this address results in a set of samples being clocked from the FIFO.

The format of the IQ sample word is expressed in table 4.1. The 16-bit channel data is obtained by left-shifting the 14-bit ADC output two places. The two least significant bits are therefore allways zero. The was done because the outputs of the ADCs are in signed, two's compliment format; and the most significant bit therefore contains sign information. Shifting the word keeps the signed bit in the most significant bit position. This simplifies the software using the samples at no extra cost in the firmware. Considering that the full-scale input to the ADCs are zero to five volts, the relationship between the channel value, V_d, and

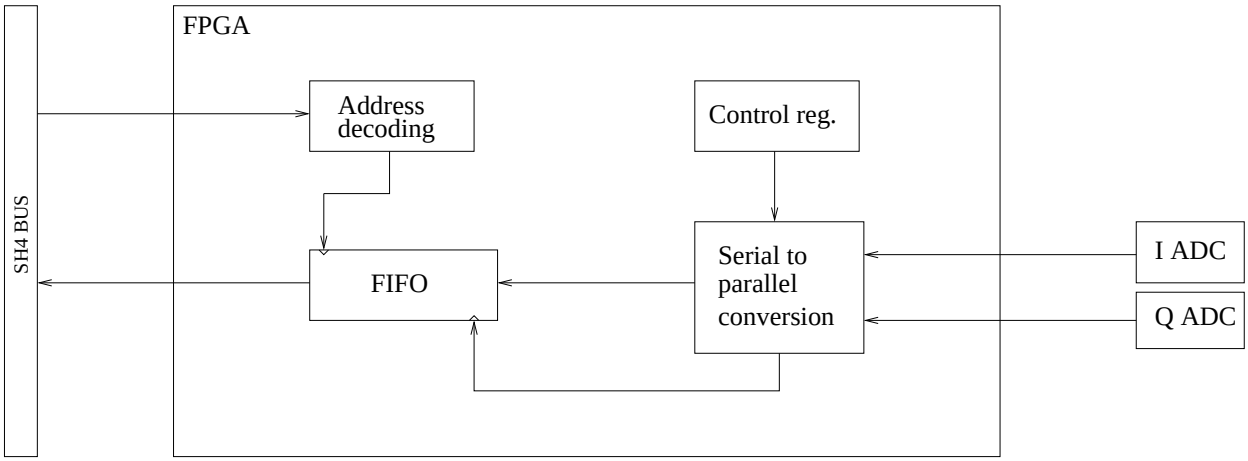


Figure 4.4: *Top-level design of the quadrature sampling subsystem.*

Table 4.1: *IQ sampling word format.*

MSB		LSB
IQ Sample		
32 bits		
16 bits	16 bits	
Q Data	I Data	
31		0

the input voltage, V_{in}, can be expressed as $V_{in} = \frac{5}{2}(1 + \frac{V_d}{2^{15}})$.

The state machine that controls the converters can be driven by any clock signal. It does not have to run synchronously with any other subsystem on the FPGA since the FIFO buffer provides a method for connecting components with different clocking sources. This allows the conversion rate to be controlled by an off-chip clock. For the engineering model, one of the FPGA's on-board phase-locked loops (PLL) was used to generate the required frequency for conversion. In the flight model an approximate frequency was derived by dividing the board clock with a synthesised flip-flop. This was necessary as the PLLs on the flight model were inoperable and it was decided, at the integration team's discretion, not to attempt to fix the problem.

4.3.3 Quadrature Transmission Subsystem

The quadrature transmission subsystem is depicted in figure 4.5. It consists of three major components which are the DAC controller, FIFO wrapper component and a numerically controlled oscillator (NCO).

The function of this subsystem is to transmit a quadrature signal, generated by the host

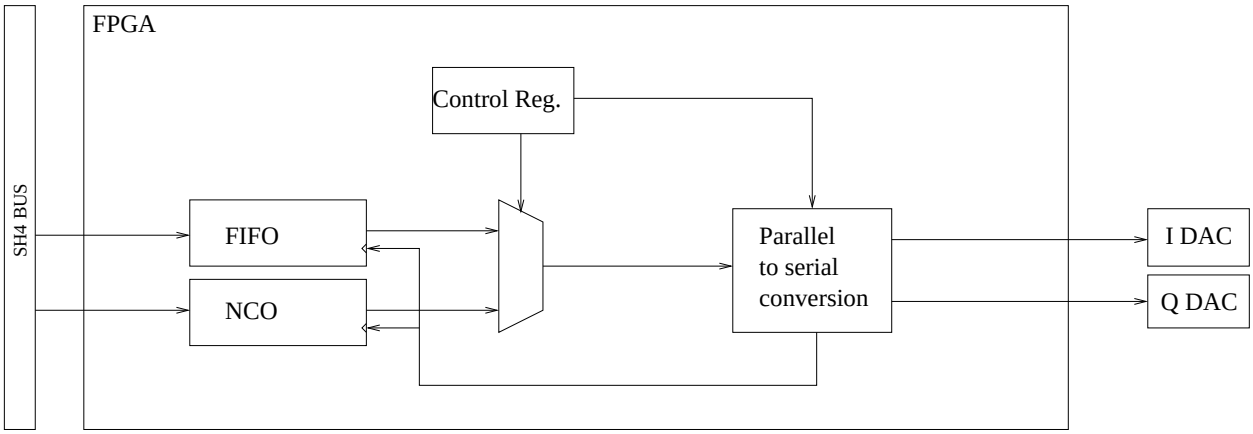


Figure 4.5: *Top-level design of the quadrature transmission subsystem.*

system over the quadrature DACs. The subsystem is responsible for generating the required control signals necessary to drive the DACs as well as to communicate the requests for new samples back to the host system. In addition to transmitting samples acquired from the host, the output of the quadrature NCO can also be output. Data is streamed to the quadrature DACs from either the FIFO or from the NCO, a multiplexer controls which input is used. In either case the data flow is directed by the DAC controller.

Samples sent from the host are written to an address assigned to the FIFO. Writing to this address results in the samples being clocked into the FIFO buffer. As with the IQ reception subsystem, both samples are packed into a single 32 bit word (see table 4.1) The format of the channel data is that same as for the IQ sampling system. The equation describing the relationship between input code and output voltage is also the same. The DACs accept 14-bit unsigned codes [22]; to convert from the 16-bit input values, the most significant bit is inverted and the result right shifted two places.

NCO

The numerically controlled oscillator (NCO) consists of two parts. These are the phase accumulator and the cosine engine. To produce two signals (I and Q), two instances of the cosine engine is instantiated.

By adjusting the NCO's output frequency, amplitude and phase, it can be operated as a direct digital synthesiser (DDS). In addition, DC offset may also be adjusted to negate any DC offset present in the signal path to the mixers. Except for frequency adjustment, all parameters can be adjusted separately for both channels. Frequency is controlled by a single 32-bit register; other parameters are 16-bit values and have both channels' parameters packed into single 32 bit words.

The phase accumulator is implemented as an overflowing counter with adjustable increment. The size of this counter was made 20 bits long, allowing frequencies down to

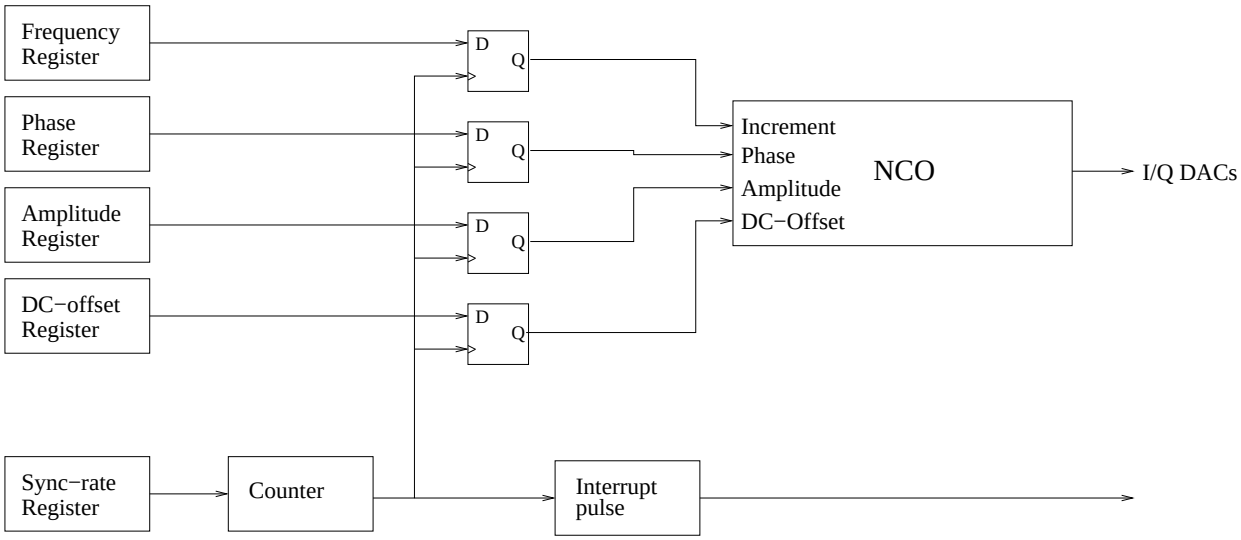


Figure 4.6: *Block diagram of the NCO's input-synchronisation circuit.*

approximately 1 Hz to be synthesised. The higher the increment setting, the faster the output phase changes and the higher the resulting frequency. The output frequency can be expressed as $f_{out} = \frac{2^{20}n}{f_s}$, where n is the increment value and f_s is the sample rate of the DACs.

Phase adjustments are made by adding an two offset phase-values (one for each of the output channels) to the output the phase accumulator. This produces two phase values with are fed to the angle inputs of the cosine engines. The phase offsets are 16-bit signed numbers, which ranges from $-\frac{\pi}{2}$ (0x8000) to $\frac{\pi}{2}$ (0x3FFF).

The cosine engine is an based on the Co-ordinate rotation digital computer (CORDIC) algorithm [8]. CORDIC is an iterative method for rotating a vector using shifts and adds; no multiply operations are needed. About a single bit of accuracy can be gained for each iteration [8]. The implementation used 13 iterations since simulations showed no appreciable accuracy gain for 14.

The implementation of the CORDIC algorithm was based on a pipelined design from the Opencores [4] repository, and was adapted for recursive operation. In order to convert the pipelined stages of the above into a recursive system, two changes had to be made to the code. The recursive implementation allowed for considerable space savings as it only required a single stage of the algorithm to be implemented on the FPGA. The first change was to add a barrel-shifter, this was needed because the original design used static shift operations in each stage. Secondly, a state machine had to be implemented that counts number of iterations through the single CORDIC stage. The barrel shifter's input is also controlled by this state machine.

The size of the input vector to the CORDIC engine will determine the size of the output. The input vector size can therefore be adjusted, using two 16-bit inputs in order to control

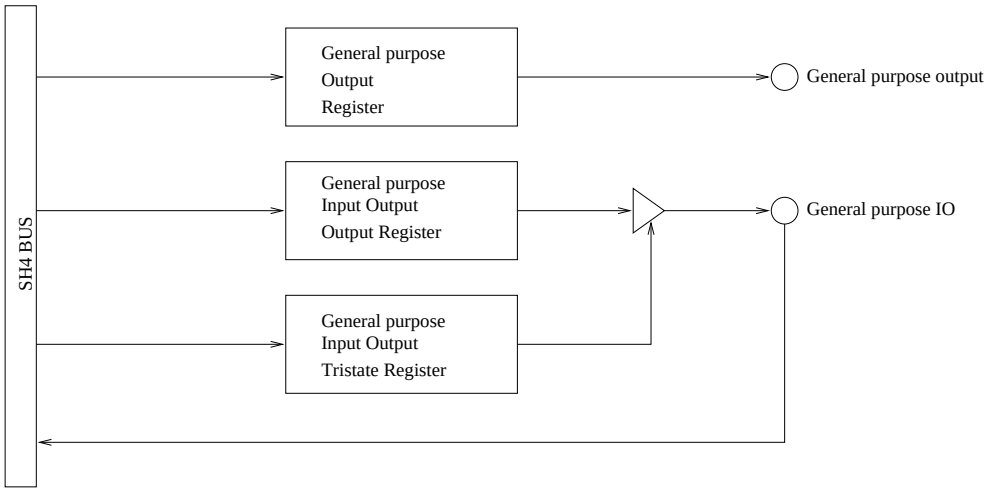


Figure 4.7: *General purpose IO block diagram.*

the amplitude of the NCO. The cosine engine's output is added to a DC-offset value before being fed to the DAC controller.

The rate at which the NCO parameters are updated is controlled by a dedicated NCO input-synchronisation circuit. The rate can be adjusted from once per output sample downwards, but the updating of parameters is optional. A 32-bit counter, similar to the phase accumulator, is used to implement the synchronisation timer. The rate of parameter updates can therefore be expressed as $f_{update} = \frac{2^{32}m}{f_s}$, where m is the increment value, and f_s is again the sample rate. There is also an option to generate a hardware interrupt to inform the host when a parameter update occurred so that new values may be supplied.

4.3.4 General IO Ports

The general IO subsystem is a trivial application of the control and status registers described in section 4.2.3. The general IO pins can either be inputs, outputs or bidirectional.

Output pins are created by configuring the FPGA pad as an output and routing a signal from a control register to it. Similarly, input pins require pads configured as inputs and routed to a status output.

Bi-directional pins require a signal from a control register to a pad's tri-state control input, as well as signals from a second control register to control its output in the low impedance mode. The signal from the pad's input buffer is routed back to a status register to allow the host to read from it.

Chapter 5

OBC Software

5.1 Overview

In this chapter the detail design of the software running on the OBC is discussed. The chapter consists of four sections, each dealing the a different aspect of the software.

5.2 Operating System

In this section the basic layout of the operating system on the OBC is discussed. It consists of three components, these are the bootloader, the kernel and the root filesystem. These components are stored in the NOR Flash banks of the OBC as three separate chunks of data. The components were kept separate to simplify maintenance of the software once the satelite was launched as only the affected components need to be re-uploaded when a change is necessary.

5.2.1 Choice of Operating System

The choice of operating kernel doesn't only affect the outcome of this project, but potentially those of future projects that make use of the SIC. The SIC is of limited use if there is no host software support for it. Development of the SIC therefore also includes development of the OS drivers that form a layer between the application and the hardware. Because the choice of operating system for this project would dictate the type of drivers it would require, some further investigation is warranted.

SDR research at Stellenbosch has primarily focused around the architecture named SU SDR. This architecture is intended to be extended by each project that uses it. It was designed on top of a Linux based OS. It should, however, be possible to port to any POSIX compliant OS as long as the hardware drivers it relies on are available on that system.

Seeing as all hardware drivers for the SIC were yet to be written, the choice of kernel would not be decided by this. SunSpace uses QNX on their SH4 OBCs. QNX is POSIX compliant, so it stands to reason that SU SDR software could be made to work on it.

Choosing another POSIX compliant kernel was not considered as neither the the advantage of compatibility with SU SDR, nor the preconfigured SH4 OBC kernel would be available.

The source code to the Linux kernel is available freely under the GNU General Public License (GPL). The GPL does not impose any restriction on the use (execution) of the kernel. This makes it an excellent choice for a research project, as future work can continue without any legal encumbrances and with full advantage of having all source code available. QNX is a proprietary kernel, some source code is available and its creators allows it to be used, without charge, for research purposes. It is therefore not unsuitable for a research project. The threat of legal encumbrances when research projects has to work with industry partners, for reasons of funding, creates a grey area and makes it less appealing.

QNX is available for several popular platforms, including the x86 based platforms on with most SDR research is conducted as SU. Linux ports, however, are available for a vast number of platforms, compared to QNX, there is even support for systems without memory management units (MMUs). Furthermore, Linux has been purposefully designed to separate platform dependent and independent code to aid with porting to new platforms. Combined with its open source model, this makes Linux extremely portable to new platforms.

Because of the availability of the complete code base and its superior platforms support, Linux was chosen as operating system kernel for this project.

5.2.2 Kernel Configuration

The Linux kernel used on the OBC is largely an unmodified mainline kernel. The changes made will be outlined below.

Version 2.6.9 of the Linux kernel was used in this project. A patch against the 2.6.9 kernel was supplied by Francois Retief of SunSpace. All subsequent work was done with this 2.6.9 kernel as it was a fairly current kernel at the time and performed faultlessly.

The patch added a directory named *sunspace_obc* under the SuperH architecture's */board* branch. In this directory was a simple board initialisation file into which further code could be added and a file containing code to generate a heartbeat pulse on one of the LEDs connected to a GPIO pin on the SH4 processor. Furthermore it patched the kernel Makefiles to add this board to the kernel build system. It also included a kernel configuration file with the address for the EDAC SRAM.

Two additional changes were made to the kernel source to work with the OBC:

- The serial port driver used by *earlyprintk* was changed to work without handshaking lines as these were not present on the OBC and made the kernel startup wait indefinitely.

- Code was added to the board initialisation file to activate the interrupt lines used for the CAN controllers and daughterboard.

The kernel was compiled with most features and drivers turned off to preserve memory. The following options were enabled :

- Loading and unloading of kernel modules.
- The debug feature *earlyprintk*. It was used to help track down problems when trying to boot the kernel for the first time.
- The file system driver for CramFS, the root file system was built into the kernel.
- Support for TCP/IP networking as well as the Point-to-point protocol.

The drivers developed in this project for the OBC peripherals and the daughter board where not compiled into the kernel. They are compiled as separate modules so that they may be easily updated without requiring the whole kernel image to be uploaded to the OBC.

Primary Memory

The OBC has both radiation hardened and conventional memory on-board. There is no clear way to utilise both these memories simultaneously in Linux.

One option is use the EDAC memory exclusively for the critical kernel data and use the unprotected memory for user space applications. That way a bit flip resulting from a single event upset (SEU) would only crash or destabilise a user space application, but the kernel would remain running and the process could conceivably simply be restarted. Implementing this would, however, require intimate knowledge of the Linux virtual memory manager. This option was therefore abandoned due to time constraints.

Another option would be to use both memories together as one larger memory block. This would however leave both kernel space and user space vulnerable to SEUs. Furthermore the EDAC memory is eight times smaller and therefore does not contribute a significant amount to the total memory. (The EDAC system also consumes a considerable amount of power, which would be wasted as the system would not benefit from it.) This option would also require some modification to the kernel, albeit less significant since the mainline kernel already has support for discontinuous memory. This option was discarded as not having any clear benefit.

Implementing a special system call would be another option. EDAC memory would be allocated for normal kernel and user space requests, but sample buffers for SDR applications would use unprotected memory. SDR applications will have to use the special system call to allocate unprotected sample buffers, all other memory allocation requests will receive EDAC RAM. The SDR buffers would have some intrinsic tolerance for radiation induced single event upsets (SEUs) because the effect would be similar to other noise induced in the communications channels. The drawback is that using a non-standard system call would make SDR applications written for the OBC less portable. The expected memory savings from this options was so small that it did not warrant the complication of a non-standard system call.

The remaining option is to use only one of the memories. It was decided to use the unprotected SDRAM as it is substantially larger and would therefore make development of

software easier. The kernel was however also tested on the EDAC memory to verify that it works and can be moved to EDAC if it is found to be necessary.

5.2.3 Bootloader

The bootloader (or bootstrap loader) is a program that is responsible for loading another program, in this case the operating system kernel. In order for the kernel to run properly, certain hardware initialisation is usually necessary. An important example of this is to configure the memory chip interface so that it may write and read from primary memory.

In a typical desktop personal computer (PC), hardware initialisation is carried out by an program called the Basic Input/Output System (BIOS) contained in non-volatile memory on the computer's motherboard. In that case the function of the bootloader is limited to finding the kernel in a file system on a hard disk and copying (usually also decompressing) it into main memory where it is executed. Because of PC design limitations these bootloaders are usually comprised of two separate parts. The first stage being charged with loading the larger second stage, the second stage may also include some form of menu interface which allows the user to choose among different operating systems.

The requirements of the experimental payload OBC is somewhat different from the functionality of a PC described above. The OBC does not contain a BIOS, the bootloader will therefore be charged with setting up the OBC hardware. The kernel will not be stored on top of a file system. The bootloader therefore does not need to understand a file system, the location where the kernel is stored will be supplied to the bootloader at compile time.

Some bootloaders can interface with debugging software, this functionality is known as monitor mode [33]. An example of this is the RedBoot bootloader from the eCos project. This functionality is extremely helpful when porting a kernel to a new platform as instructions can be traced as they execute on the host platform and variables may even be modified in place.

While the RedBoot loader mentioned above would have been a good option if interactive debugging of the kernel was required, the port of Linux to the SuperH platform was already mature. Furthermore, the OBC does not rely on any special hardware outside of the LinuxSH project. It was decided monitor capabilities did not warrant the time required to adapt the complicated bootloader to the OBC hardware.

Instead it was decided to write a minimal bootloader to fulfil the requirements described above. The only debugging feature implemented in the bootloader was to verify that the kernel image in ROM was correctly uploaded into ROM. This was accomplished by calculating a CRC checksum and sending it to the SH4's built-in serial port before attempting to execute the kernel image. The CRC could then be read from a terminal connected to the serial port and compared to the checksum of the original image.

The bootloader for the OBC has the following tasks:

- Configure the SH4's BSC.

- Set the on-board CAN controllers to an inactive state.
- Initialise the EDAC RAM to prevent the false resets.
- Generate a CRC checksum of the loaded kernel and display it a terminal connected to one of the SH4's build-in serial ports.
- Copy the kernel image from the boot ROM into RAM and execute it.

The bootloader was implemented in both SH4 assembly language and C. Much of the assembly code was derived and adapted for the OBC from the sh-boot bootloader's code-base [3]. An assembly language routine initialises the OBC to the point where the RAM may be used. A stack pointer is then set after which subroutines, written in the C language, may be invoked. The rest of the bootloader is then implemented in the C language. The last step is to call an assembly language subroutine which jumps to the start of the kernel image.

When building programs to run on top of an operating system, the linker uses built-in knowledge of the execution environment to place executable code and other segments, such as variables, in the correct places. The bootloader runs before any software environment is established and can therefore not rely on this. Specific instructions has to be given to the linker to ensure that program code and variables are located in the correct places. This is accomplished with a linker command script [16], which allows exact placement of program segments.

As can be seen in listing 5.1, the linker control script defines memory regions and output sections. Two memory regions are defined in the listing, one pointing to the flash ROM where the bootloader is located, and the other pointing to the SRAM. The listing shows the definition of a segment called *.text*[1]. The instructions tells the linker to create symbols at the beginning and end of this segment called *__test_start* and *__text_end* respectively. These symbols may be referred to in the bootloader code as markers to the beginning and ending of the executable code in memory. The linker is instructed to place all segments named *.text* and *.rodata* from all input object files into this segment. Finally the linker is told to place this segment in the *rom* memory region.

Several of these segments are defined to cover the linking of stack data, variables and initialised variables.

5.2.4 Root File system

In order to maintain a level of compatibility between different distributions based on the Linux kernel, and Unix in general, the Filesystem Hierarchy Standard (FHS) was created. It aims to describe, and standardise, the structure of the root file system of compliant Unix

[1].text is a name historically given to executable segments.

Listing 5.1: *An extract from linker command script used to link the bootloader*

```
MEMORY
{
 rom   : o = 0xA0000000, l = 127k
 ram   : o = 0x90000000, l = 8M
}

/* how we're organizing memory sections defined in each module */
SECTIONS
{
 /* code and constants */
 .text :
 {
  __text_start = .;
  *(.text)
  *(.rodata)
  *(.rodata.*)
   __text_end = .;
 } > rom
```

systems. The Linux kernel does, however, not impose many limitations on this structure. It would be possible to configure a system without a structured file system. Many programs that run in user space, however, do rely on the structure and would therefore need to be modified if the structure is changed. The file system used on the OBC is therefore based on the guidelines defined by the FHS.

The root file system used on the OBC slightly differs from the standard. The changes are mostly omissions. The */boot* directory is for instance omitted as the bootloader and kernel are not stored in the file system. One notable difference is the */mod* directory, used to store drivers implemented as loadable modules. Loadable modules are usually stored in subdirectories under */lib/modules/2.6.9/*, where 2.6.9 is the version of the kernel and will differ depending on what version of the kernel is installed. It was deemed unnecessary to create such an elaborate structure to hold only handful of drivers.

CramFS was selected as the file system to house the root file system. It is a compressed read-only file system. It is a lightweight file system specifically targeted at systems with small files and file system footprint. The limitations imposed by the file system is that only read access is available and file sizes are limited to 16 MB. The total file system size is also limited to about 256MB[2]. The limitations result in a much simplified, and small,

[2]Total size may be larger as long as the last file entry starts before 256MB.

implementation. Because files are compressed, they have to be decompressed in memory when read.

A patch was applied to the kernel driver for CramFS that allows files stored in the file system to be executed in place (XIP). This means that the files are not first copied into main memory before it is executed. This function does, however, require that those files be left uncompressed and that they are aligned on page boundaries. A special version of the *mkcramfs* utility was used to create the file system images, stored in the NOR read-only memory bank. This utility allows files to be marked, using the unused sticky bit, for XIP. Those file are then page aligned and left uncompressed. XIP allow for some memory saving as the code segments of running programs does not need to be copied into RAM. It also improves the first time execution speed of programs as the process of copying the code into RAM is bypassed.

The root file system can be updated after the satellite is launched by uploading a new file system image. To speed up the process, the root file system was partitioned into two parts. The first is mounted by the kernel as the root file system at boot-up and contains a minimal system. All the system libraries and binary programs that were considered stable are included in the image. The second image is loaded by the user space *init* process. This image contains the programs and drivers developed as part of this project, in other words the programs that are less well tested and therefore more likely to need updates. The second image is also a CramFS image, allowing the compression to further reduce the size of the upload.

5.3 SIC Drivers

This section describes the driver that allow access the firmware components on the daughter board. These drivers are the low-level register access driver, the peripheral sampling driver, the drivers for the IQ subsystems and a driver to demonstrate the use of the NCO. An overview of these component were given is section 3.4.2.

5.3.1 Experimental Payload Drivers

Register access

The driver allows low level access to the registers on the SIC. All communication with the driver is in the form of hexadecimal strings, as discussed in section 3.4.2. For each register that the driver provides access to, a device node has to be created in the filesystem. On the experimental payload these nodes where placed in the */dev* directory. Nodes are named[3] by concatenating the letter x with the register's decimal number.

[3]The actual node names does, of course, not matter as long as the major and minor numbers are correct.

A node's minor number determines which register it addresses. The major number has to match the driver's major number. The register's number is determined by the upper six bits of the register's eight bit address. The lower two bits are ignored because all addresses fall on 32-bit boundaries (see section 4.2.1). Upon execution of the *open* system call, the driver determines the register's memory address by examining the minor number of the device node.

A sequence of values may be read from a register by invoking the *read* system call multiple times between opening and closing the device node. This allows for an arbitrary number of successive reads. The following is an example of how the *dd* program may be employed to read the contents of register number zero from the command line:

```
dd if=/dev/x0 count=1
```

A sequence of values may also be written to a register by invoking *write* with a buffer containing a list of space-delimited hexadecimal values. The maximum number of values is determined by a constant in the driver code; during development never more than eight values were written in this manner. The *write* system call may however be invoked a arbitrary number of times between opening and closing. The ability to do a sequence of writes during a single system call was added because it writes the sequence considerably faster than multiple system calls would, the reason being that the whole sequence may be translated from hexadecimal to binary values before being sent to the register. To write the sequence a sequence of values to a register, the following may be typed on the command line:

```
echo 108 109 10A 10B 10C 10D 10E 10F > /dev/x10
```

Peripheral sampling

The goal of the peripheral sampling driver is to provide a simple interface to processes requiring data from one of the peripheral sampling banks. The low-level register access driver could in theory be used to accomplish this, but would be very inefficient because all samples would need to be converted to hexadecimal and, most likely, back to a binary representation. Furthermore, a system call would need to be executed for each sample gathered. This amounts to 400 000 system calls per second if all four ADCs are read. Additional system calls would also be needed to read the status of the FIFO buffer. The peripheral sampling driver therefore provides a more efficient interface to acquire binary values in bulk.

The peripheral sampling subsystem on the FPGA provides several sampling options (see section 4.3.1). These include the option to sample a selected subset of the ADCs and an option to format the output as binary two's compliment or unsigned data. In order to simplify the driver, only a single mode was implemented: sampling all four ADCs and format output as two's compliment, signed data.

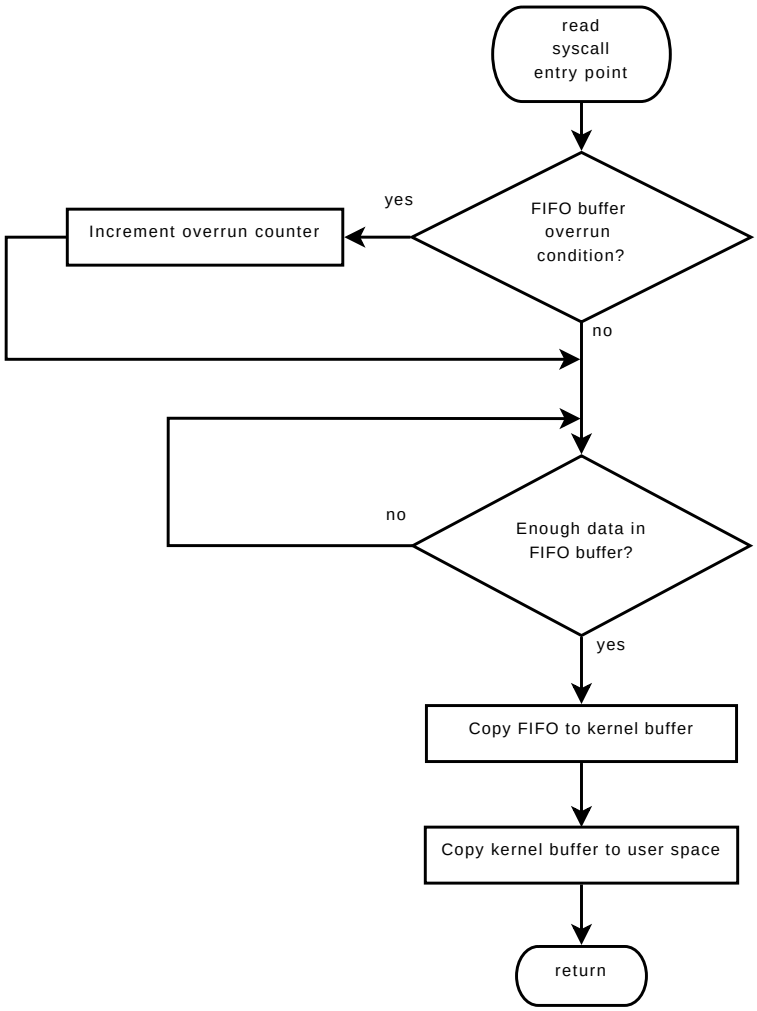


Figure 5.1: *Flowchart of peripheral sampling driver's read system call.*

Table 5.1: *Peripheral sampling driver's output word format.*

MSB (127) LSB (0)

Sample group							
128 bits (16 bytes)							
32 bits		32 bits		32 bits		32 bits	
14 bits	18 bits	14 bits	18 bits	14 bits	18 bits	14 bits	18 bits
Zeros	ADC 3	Zeros	ADC 2	Zeros	ADC 1	Zeros	ADC 0

The output format of the driver is a group of four 32 bit sample words, each containing a single 18 bit value, as read from the applicable ADC. The remaining bits in each sample word is padded with zeros. The structure is depicted in table 5.1.

In order to improve the efficiency of the driver, a block of sample groups is returned for each *read* system call. The number of sample groups returned is controlled by a constant in the driver code and has to be smaller than the length of the FIFO buffer. If the calling process is able to consume samples faster than the ADCs produce them, the size of the sample blocks will determine to average utilisation of the FIFO buffer. Choosing small values results in the FIFO operating at near empty and therefore leaves ample room in the FIFO buffer when the calling process is delayed. Small values also results in a higher system call to sample ratio, requiring the host system to do more work. Large values reduces the ratio of system calls to samples, but leaves less room in the FIFO buffers for program delays. Because the OBC is not required to do any work while acquiring samples for the experiments, a small value was used on the experimental payload.

Each of the two banks has a device node allocated to it. These are located at */dev/padc0* and */dev/padc1* on the experimental payload. A process selects which bank to sample by opening its corresponding device node. This is the same method used by the register access driver.

The driver implements the *open*, *read* and *release* system calls. When a process opens a device node the driver sets the FIFO's trigger to the same value as the size of the sample blocks and starts the ADCs.

The *read* system call (Fig. 5.1) is implemented as a blocking system call. This means that it will not return until it has data available. If the FIFO status register indicates that enough samples are available, the data is read from the FIFO and copied to user space; if not, the driver polls the FIFO status register in a busy-waiting loop until enough samples are available. A buffer overrun counter is also maintained, it is incremented each time the FIFO reaches the full mark to indicate that an overrun situation possible occurred.

The *release* system call stops the ADCs and prints a warning to the kernel message buffer if any overruns where detected.

5.3.2 IQ sampling and transmission

The IQ sampling and transmission drivers allow SDR processes to receive and transmit samples over the IQ sampling and transmission systems respectively. The interaction between these drivers and the SDR application is shown in Fig. 3.6.

Data is transmitted between user space and kernel space in sample groups. Each sample group consists of a number of IQ samples (see table 4.1). The number of samples in a group is determined by a constant in the driver code and is set at compile time. This was done to simplify the code. Future versions of the drivers may allow the number to be changed dynamically.

All the drivers implement blocking system calls. In addition to *close* and *release*, the transmission drivers implement the write system call, while the sampling driver implements *read*.

Two sets of drivers were implemented. The first set uses a simple busy-waiting algorithm to transfer samples between the SIC and user space. This requires the processes to process their sample groups in real-time. These drivers therefore have limited flexibility and was mainly used to debug the system.

The second set of drivers provide more flexibility by utilising a kernel space ring buffer to store the samples. The communication with the SIC is controlled by interrupt handlers and communication with userspace is through normal system calls. The buffer allows the two paths to run asynchronously, thereby providing the temporal decoupling discussed in section 2.1.4.

Busy-waiting drivers

The functionality that the IQ sampling driver provides is similar to that of the peripheral sampling driver's (Fig. 5.1). The important differences are the format of the sample data (see table 4.1) and the FPGA addresses accessed by the driver. The discussion of the peripheral driver therefore adequately describes the IQ sampling driver.

The IQ transmission driver allows processes to send data over the IQ DACs by writing data into the driver's device node.

As with the ADC drivers, the *open* system call sets the FIFO trigger. However, before the DACs are started, the FIFO is filled with zeros. This provides a grace period wherein the process may begin generating samples before an underrun occurs. An underrun counter is maintained so that warnings of underrun conditions may be generated The *write* system call (Fig. 5.2) copies the userspace data into a kernel buffer before it starts polling the FIFO status register. Carrying out this operation in advance reduces the average time a calling process is blocked by the system call. The underrun counter is incremented if the FIFO status register indicates that the FIFO is empty. The status register is polled until it indicates that enough space is available in the FIFO for the user space data. Finally the kernel buffer is copied into the FIFO.

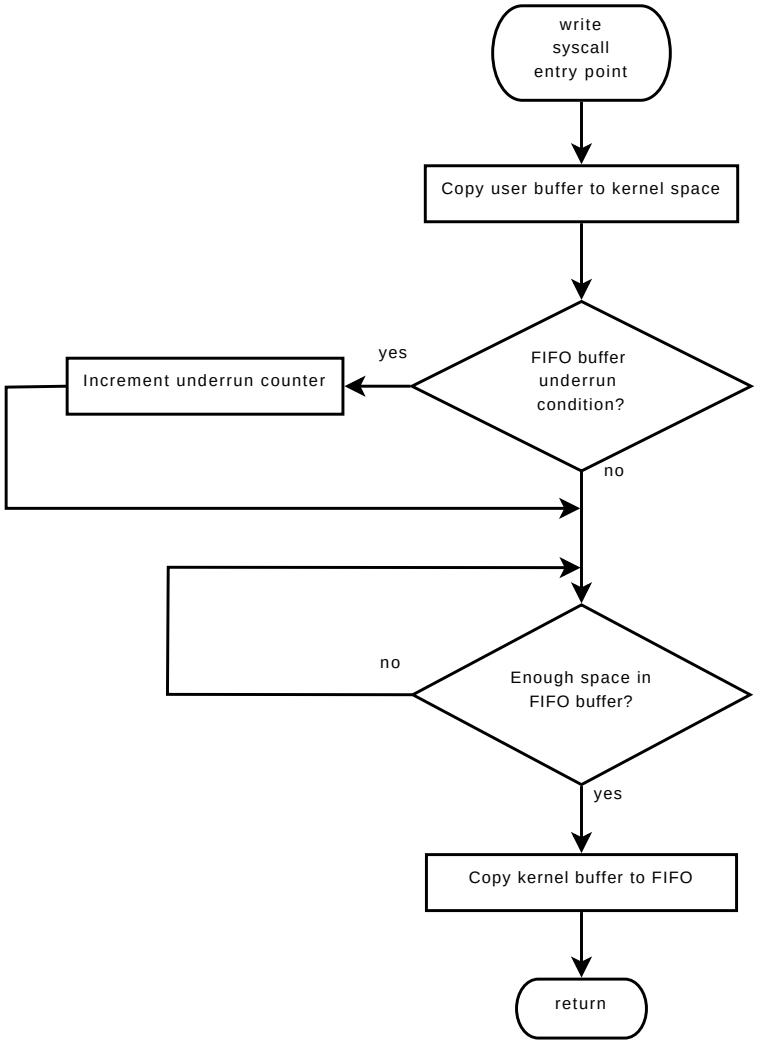


Figure 5.2: *Flowchart of busy-waiting IQ-transmission driver's write system call.*

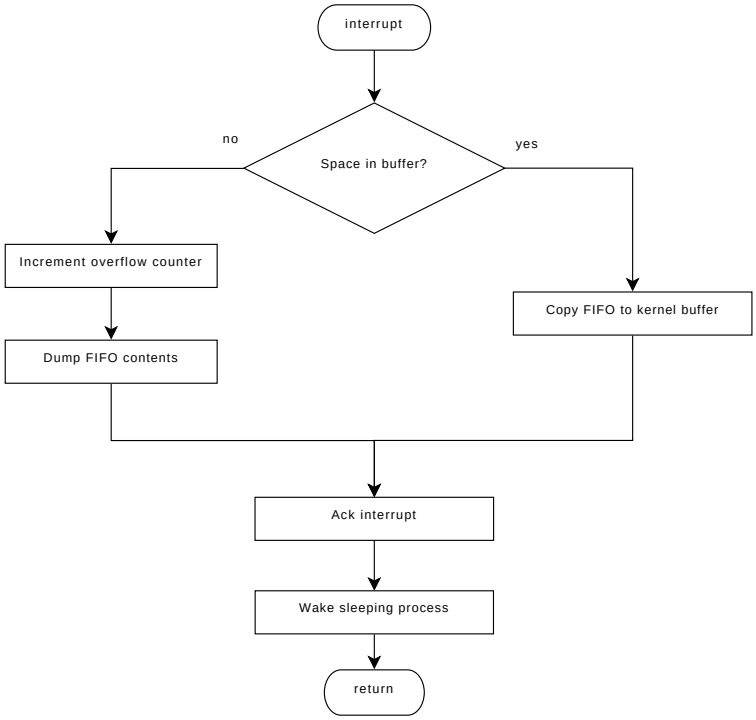


Figure 5.3: *Flowchart of interrupt driven IQ-sampling driver's interrupt handler.*

The release system call stops the DAC and writes an underrun warning in the kernel message log if possible underruns where detected.

Interrupt driven sampling driver

The purpose of the interrupt driven sampling driver is to allow calling processes to have more flexible execution times. Processing of single sample groups does not have to complete in real-time as long as the average processing speed is faster than real-time. During slow processing cycles incoming samples are stored in the device driver's buffer. These excess samples can then be consumed during faster processing cycles.

The *open* system call initialises the IQ sampling system. The FIFO's trigger is set to the size of the sample groups. This allows the interrupt handler to retrieve a complete sample group each time it is called.

The *close* system call turns interrupt generation off and stops the IQ sampling system.

The interrupt handler (Fig. 5.3) is called by the kernel when the SIC signals an interrupt condition. Samples are fetched from the FIFO buffer and stored in the device driver's buffer. If the buffer is full, the sample group is thrown away and the underrun counter is incremented. The calling process is, however, not terminated, but a warning will be generated in the system log upon releasing the device node. Before the interrupt handler returns, the SIC is signalled to indicate that the interrupt was served and the kernel is informed that new data is available in the buffer. This allows the kernel to wake the calling process if it was sleeping.

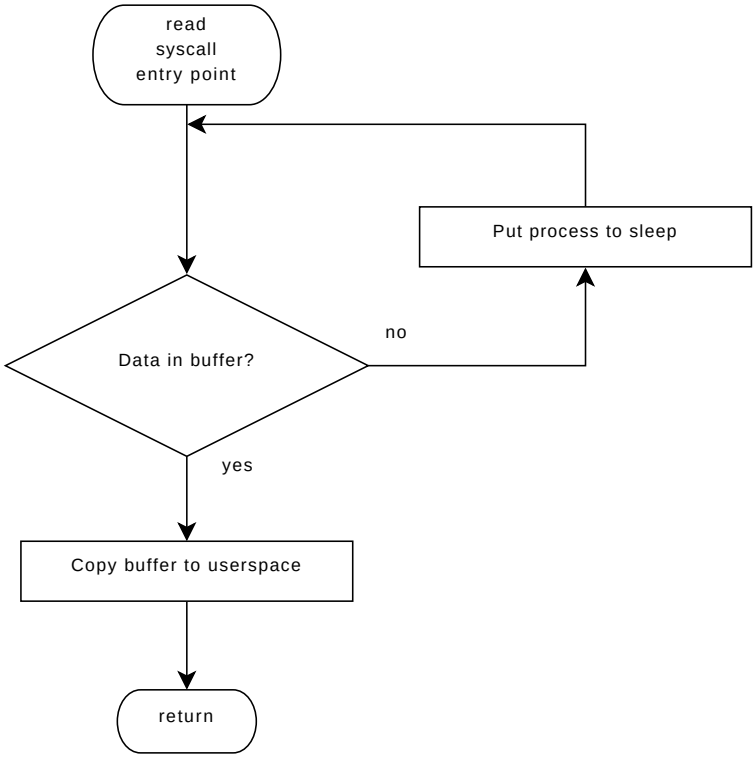


Figure 5.4: *Flowchart of interrupt driven IQ-sampling driver's read system call.*

The *read* system call (Fig. 5.4) retrieves samples from the driver's buffer and returns them to the calling process. If the buffer is empty the calling process is put to sleep until new data arrives.

Interrupt driven transmission driver

The interrupt driven transmission driver allows processes to generate single sample groups at a slower than real-time rate as long as the average rate is higher than real-time. Faster execution cycles will result in sample groups being cached in the device driver's buffer, the DACs will then be fed from this cache during slower execution cycles.

The *open* system call behaves similarly to that of the sampling driver. The FIFO trigger is set so that an interrupt is generated when enough space is available for a sample group. Before the transmission system is started, the driver's buffer is filled with zeros to allow the calling process some start-up latency.

The *release* system call waits for the driver's buffers to empty and then stops the transmission system.

The interrupt handler (Fig. 5.5) writes data from the driver's buffer into the FIFO. If the buffer is empty a sample group, consisting of all zeros, is written to the FIFO instead, and the underrun counter is incremented. The SIC is signalled that the interrupt has been served and the kernel is signalled that space is available in the driver's buffer.

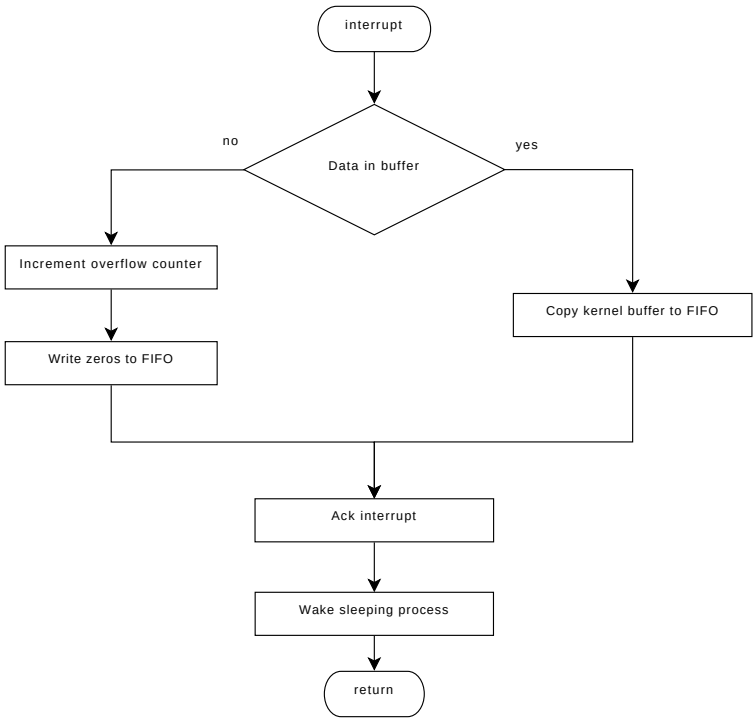


Figure 5.5: *Flowchart of interrupt driven IQ transmission driver's interrupt handler.*

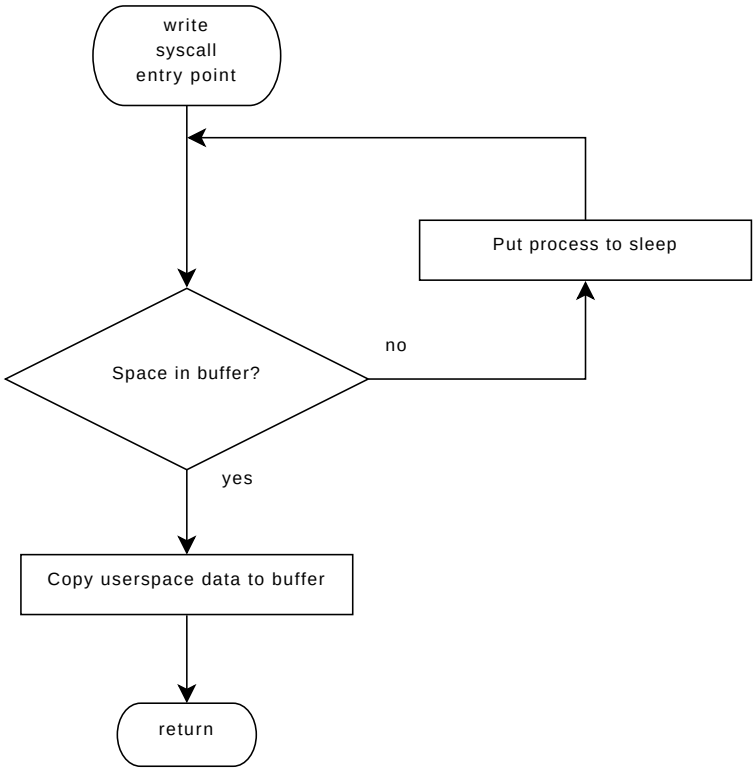


Figure 5.6: *Flowchart of interrupt driven IQ transmission driver's write system call.*

Table 5.2: *CAN Message identifier format.*

Frame identifier	Source address	Destination address
13 bits	8 bits	8 bits

The *read* system call (Fig. 5.6) writes the user space buffer into the driver's buffer. If the driver's buffer is full, the calling process is put to sleep until space becomes available.

5.3.3 DDS proof-of-concept driver

The purpose of this driver is to demonstrate how the NCO can be used for direct digital synthesis (DSS). In addition to *close* and *release* The driver only implements the *write* system call. The parameter update-rate (see section 4.3.3) is hard-coded to a value suitable for the demonstration modem in section 6.3.3.

The *write* system call accepts four 32-bit words—containing the frequency, amplitude, phase and DC-offset. These are loaded into the NCO control registers once every update-cycle. The *write* call blocks until the parameters are loaded and has to be called once for each output symbol. No provision is made for buffering parameters in the driver.

An interrupt handler is installed by the driver, which will be called once every update-cycle. The only functions of the handler are to acknowledge the interrupt to the SIC and to wake the sleeping *write* system call.

5.4 OBC Drivers

This section describes the drivers that allow access to the peripherals on the OBC. Section 3.4.2 gives an overview of their functions.

5.4.1 CAN Driver

Protocol

Physical and link layer is provided by the OBC. The hardware on the OBC does some filtering of messages so that only messages destined for the OBC board will be visible to the software running on the SH4 processor. SunSpace uses the extended frame format with a 29 bit message identifier (see table 5.2). The message identifier consists of a frame identifier (which also encodes the message priority for arbitration purposes in the event of a collision) and a source and destination address. Each frame can optionally also have a message payload of up to eight bytes.

The CAN bus acts as a low-bandwidth communication channel between the satellite components. It is primarily used for issuing telecommands and requesting telemetry information.

Table 5.3: *CAN Driver's packet format.*

Source	Destination	Frame identifier	Optional Payload Bytes
8 bits	8 bits	13 bits	0–8 bytes

Provision is also made for transmission of firmware upgrades to satellite components as well as tunnelling higher-level protocols such as TCP/IP.

Initialisation

The driver registers a character device for each CAN node. The SH4 OBC has two CAN nodes. On the experimental payload they are located at */dev/can0* and */dev/can1*. The driver is capable of handling an arbitrary number of nodes, but initialisation of each node is hard-coded in the start-up code of the driver. To extend the driver to more nodes, more calls to the node initialisation function would have to be hard-coded into the driver start-up code. A more elegant solution would be to add a hardware probe function to the start-up code. This was, however, decided to be too much work for a non-essential feature.

System calls

open Each device node implements an open and release system call. Only one process per device is allowed to open the device for reading at a time, this would usually be the process responsible for decoding the messages and responding to them. Multiple writers are allowed so that sub-processes can easily be created to respond to frames from the CAN bus.

release If the process that had the device node open for reading calls release on a device node, the read lock is removed. This enables future processes to gain read access to the device.

write The driver implements the write system call to enable transmission of CAN frames. Each call transmits a single CAN frame, the driver does not allow for partial frames to be sent from user space to kernel space. This limitation simplifies the driver since partial frames do not have to be reconstructed in kernel space code prior to transmission. All write operations behave in a blocking fashion. The low frequency at which frames would be sent did not require more advanced methods. The actions taken upon receiving a frame from a user space process are the following:

- Conversion of the hexadecimal string to binary form
- Locking the device to prevent race conditions with other writers
- Wait if CAN hardware is busy transmitting a frame

- Sending of frame to hardware
- Unlock device

read The driver allows for blocking and non-blocking read operations, by implementing the poll system call. In blocking mode the calling process will be put to sleep when no data is available for reading and woken up when new data becomes available. In non-blocking mode, the driver will return `EAGAIN` to indicate that no data is available. One of the polling system calls, such as `select` can be used to wake the process when new data becomes available.

Upon receiving a message over the CAN bus, the CAN controller on the OBC board generates an interrupt. An interrupt service routine (ISR) retrieves the frame from the CAN hardware and stores it in a buffer. Thereafter the waiting process, if any, is woken up to so that it may proccess the new frame.

5.4.2 NAND Flash Interface

In Linux terminology, devices such as RAM, ROM and flash chips are grouped together and called memory technology devices (MTD). The kernel includes a whole subsystem to provide a unified method of managing these devices. The MTD subsystem uses a layered approach to accommodate widely different technologies. At the lowest level, chip drivers address the peculiarities of different MTDs. This presents a unified layer to which MTD user modules[4] can connect. These user modules in turn provide a familiar interface to which the rest of the kernel may connect.

In order to use a device with the MTD subsystem, a mapping driver is required. The function of this driver is to tell the MTD subsystem where to find and how to connect to the device. Mapping drivers are required because there is no standard for connecting such devices to a system. A flash chip may for instance be directly connected the processor's bus or it may be connected through GPIO pins on a peripheral device.

Two file systems where considered for the NAND flash banks. The first one, Yaffs[5], was discarded due to inconsistencies in the way it connects to the MTD subsystem. At the time of writing, Yaffs made some assumptions about the MTD interface that were inconsistent with the way MTD was implemented. While these assumptions had merit, it excluded Yaffs from using the lower MTD layers. To correct the problem would have either required parts of Yaffs (or MTD) to be rewritten or the MTD subsystem to be completely bypassed. Bypassing the MTD layers would have required the functions of the NAND chip drivers to be reimplemented. It was decided that the time to implement would this was not merited by the gains of using Yaffs.

---

[4]This name is somewhat misleading as it neither refers to user space programs nor kernel modules.

[5]The name is derived from Yet Another Flash File system.

The file system selected was JFFS2. It has similar features to Yaffs, but is not as well suited to large chips as Yaffs. It does, however, integrate seamlessly into the MTD stack since, unlike Yaffs, it is part of the official MTD subsystem. At the time of writing the JFFS3 file system was still in early stages of development. If it had been sufficiently ready, it would have been the likely choice as it addresses JFFS2's limitations with large file systems.

In order to use the NAND banks on the OBC, a mapping driver has to be written. The mapping driver tells the MTD system how its chips should be partitioned, and defines the functions needed to access the chips. The OBC's mapping driver for the NAND banks has to define three functions in order to allow the rest of the MTD system to function. These functions are *hwcontrol*, which control the chip select and address latch inputs to the NAND chips and *read_byte* and *write_byte*, which writes single bytes of data or control information to the chips. The rest of the functions required to access the chips are derived from the MTD NAND chip driver, which uses the aforementioned functions to provide access to the MTD user modules.

The NAND chips on the OBC is connected so that they may be addressed as a single, larger, chip. According to SunSpace, the arrangement does not perform as expected. The chips, therefore, have to be addressed individually. Because they are electrically tied together, the chips also cannot be accessed simultaneously as two distinct flash devices. The solution adopted in the mapping driver is to only allow access one chip. If access to the other chip is required, the mapping driver for the first chip must be unloaded and the second chip's mapping driver loaded. If simultaneous access to both chips has to be implemented, it would require some of the inherited NAND-chip driver functions to be rewritten so that the state of the chips may be taken into account when accessing them.

5.4.3 LVDS Port Interface

The most efficient method for transferring large data blocks to the satellite's main data store is over the OBC's LVDS transceiver. Only the transmission port is of interest, the LVDS receiver is unused in the experimental payload.

The interface to the transmission port, as seen from the SH4's perspective, is quite simple. Handshaking is taken care of transparently by hardware on the OBC. Transmission of data is accomplished by writing a data word into one register and then polling a status register to determine when the next data word may be written. Transmission occurs in discrete 4 byte frames; if less data is to be transmitted, it has to be padded. The LVDS hardware has no capability for generating interrupts. The hardware has a DMA mode, but the operation of this mode is unproven and, at the time of writing, was not implemented in the SunSpace QNX drivers. This project has therefore not attempted to utilise this feature.

A simple driver was written that exposes a character device. The *open*, *write* and *release* file operations are implemented. The LVDS port is initialised within the *open* function, and shut down in the *release* function. The write function accepts four bytes during each invocation of the write system call. If less than 4 bytes are offered, it is padded with zeros.

An application wishing to transmit data over the LVDS port has to ensure, therefor, that at least four bytes of data are available at during each write cycle.

The driver was tested in the lab against hardware supplied by SunSpace that emulates the LVDS receiver in the satellite. Under lab testing conditions it was found that glitches would periodically appear in the received data. It was hypothesised that the hardware handshaking might be to blame. The problem was left to the integration team to solve, as the problem fell outside of the scope of this thesis.

5.5 User space

5.5.1 User space services

Programs

Most of the user space utilities are provided by the BusyBox package. BusyBox is a single, multi-call, binary file that contains most of functionality of the standard Unix utilities. It is targeted at embedded systems where footprint size is important. BusyBox's build system allows the included functionality to be carefully selected at compile time to create a binary file that is as small as possible.

Two other user space programs where cross compiled to be used in the user space environment. These are *pppd* and *rsync*. During development the two serial ports of the SH4 was used as a terminal, for shell access, and a point-to-point protocol (PPP) serial link. The PPP link enabled an IP connection to be established between the OBC and the development workstation. Files could subsequently be exchanged between the OBC and the workstation using the rsync protocol. This speeded up development as test files could be uploaded to a RAM drive on the OBC in stead of having to re-program the flash drives. The *pppd* and *rsync* programs have to be started manually after the OBC boots; this ensures that system resources are not waisted when these processes are not required.

Initialisation sequence

The final step of the kernel's start-up procedure is to execute the first user space program. The kernel executes the program located at */sbin/init*. On the OBC, this is a link to the BusyBox binary file, which results in BusyBox being invoked as the init process. In this mode, BusyBox reads the contents of */etc/inittab* to determine its behaviour. The */etc/inittab* file directs BusyBox to execute */etc/rc*[6]. This file is responsible for mounting the second file system (see section 5.2.4) and running its *rc* script. The second file system loads the drivers needed to perform the OBC functions and finally executes the process responsible for running the experiments.

[6]The letters *rc* are short for run command, which is usually an initialisation script.

5.5.2 Experimental Payload Interface

As depicted in figure 3.4, all actions performed by the experimental payload are scheduled by ground station control. Actions are controlled through commands sent through the satellites CAN bus to the OBC. The software design called for a single program to wait for commands from the CAN bus and to respond to those requests. A simple program, the CAN servicing program (see Fig. 3.7), was created to serve as an example of how to accomplish this goal. This program was used by the integration team for testing of the experimental payload in the satellite.

The main loop of the program reads the CAN node is in blocking mode. Upon reception of a CAN packet, its contents is examined and a appropriate response is initiated. If the packet is invalid, it is silently ignored and the read cycle begins again. This simple structure can easily be expanded on when new actions are required for the experimental payload.

Some requests are simple and can are handled directly be the CAN servicing program. A request to read a single register on the daughterboard, is for instance, handled directly. Other requests are handled by creating child processes to perform the requested actions. Returning the captured data from an experiment over the CAN bus, is for instance handled by a child processes.

To integrate the the CAN servicing program, some telemetry information and telecomands had to be specified for SunSpace's satellite control interfacing system, known as the Lua Command Interface (LCI). These specifications are used by LCI sessions to convert telecommands and telemetry into CAN frame identifiers.

5.5.3 IP over CAN

SunSpace's CAN protocol makes provision for internet protocol (IP) packets to be tunneled over the CAN bus. This could allow a payload to communicate with programs running on the ground station, using standard protocols. This is of special interest to the experimental payload because it can be used with the *telnet* program, to gain command-line access to the OBC without a serial cable. CAN packets can be transmitted over RF by the satellite, this will therefore give a powerful facility for fault finding once the satellite is launched.

A proof-of-concept IP-over-CAN program was implemented to test if it was feasible to connect the OBC to other hosts using tunneled IP connections. The SunSpace implementation of tunneled IP traffic works by splitting incoming IP frames into 8-byte long CAN packets [30]. These packets are reassembled on the receiving side to form the original IP frame.

The proof-of-concept implementation used the Linux *tun* virtual network interface driver. It allows IP or Ethernet frames to be sent to the Linux network stack through a character device node. Packets returned by the network stack can be read from this character device node. The program's function is therefore to split frames read from this character device into smaller packets which can be sent over the CAN bus. Incoming CAN packets have to

be reassembled into IP frames and sent to the character device.

Chapter 6

Evaluation

In this chapter the implemented systems are evaluated. The operation of the software environment is first verified. Simple communication between the software environment's drivers and the firmware is next established. Thereafter measurements are taken on the daughter board to verify the operation of the firmware modules. Finally an application is executed on the system to demonstrate the its use as a SDR platform.

6.1 Software Environment

6.1.1 Bootloader

The purpose of this test is to verify that the bootloader correctly loads the kernel image into memory. If the kernel image has been damaged or not completely transferred into memory, the operating system may become unstable and even crash.

A cyclic redundancy check (CRC) is calculated for the kernel image on the development host system, and a CRC subroutine is added to the bootloader's code. This subroutine is executed after the kernel image has been transferred from the boot flash (see section 3.2.4) to RAM. The resulting sum is printed out to the SH4's first serial port. A PC is connected to this serial port where a terminal emulator program reads the output. The bootloader's calculated CRC is then compared to the CRC calculated on the host to verify its correctness.

This test was successfully carried out during development.

A correct checksum indicates that the integrity of the kernel image was preserved during the transfer from the development host to the boot flash, as well as the transfer to the OBC's RAM. The latter is important because it also serves as a memory test and indicates that the SH4's bus state controller was correctly initialised and that the RAM can be accessed successfully.

6.1.2 Operating System

The purpose of this evaluation is the verify that the primary functions of the kernel (file access, process execution and, memory allocation) is working properly.

Evaluation of command-line environment

For this test the OBC is powered up and allowed to load the Linux kernel. The kernel was instructed (at compile time) to utilise the second serial port as a console device. A terminal emulator program (on a PC connected to the serial line) then provides command line access to the shell interpreter running on the OBC.

First, two files in the kernel's virtual *proc* file system is evaluated. The contents of */proc/cpuinfo* is examined by typing the following command in the command line:

```
cat /proc/cpuinfo
```

The output reported is the following:

```
machine         : SunSpace SH4 OBC board
processor       : 0
cpu family      : sh4
cpu type        : SH7750R
cpu flags       : fpu
cache type      : split (harvard)
icache size     : 16KiB
dcache size     : 32KiB
bogomips        : 191.69
cpu clock       : 192.00MHz
bus clock       : 64.00MHz
module clock    : 32.00MHz
```

Evaluating this information against the OBC manual [29] and SH4 datasheet [25] indicates that the kernel has correctly identified the important features of the OBC's architecture.

Next */proc/meminfo* is evaluated in a similar way. The first two lines of its output is reproduced below. The other lines give information about of the kernel's internal memory structures that are not important to this discussion.

```
MemTotal:        63852 kB
MemFree:         61800 kB
```

The total available memory is less than the 64 MB on the OBC. The missing memory (about 1.6 MB) is reserved by the kernel (in large part for the executable kernel code) and will never be available for reallocation. Free memory is memory that has not been allocated. The difference between total and free memory is mostly allocated to kernel data structures and running processes. Kernel modules loaded by the initialisation scripts also requires memory. Finally some of the allocated memory is allocated to a file system cache (for files that were decompressed from the CramFS images).

Table 6.1: *STREAM benchmark results on SH4 OBC.*

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	67.7418	0.4725	0.4724	0.4728
Scale	62.4309	0.5127	0.5126	0.5130
Add	68.5146	0.7007	0.7006	0.7008
Triad	61.1070	0.7858	0.7855	0.7866

Memory bandwidth benchmarking

The purpose of this test is to verify that the free memory reported by the kernel can be accessed by user space applications. Furthermore, an important aspect of the system's performance will be tested.

For this test both serial ports are connected to a PC. The SH4's first serial port is used as a PPP networking connection between the OBC and a PC. The *pppd* program is started on the PC's serial port. On the OBC the *pppd* and *rsync* programs are by executing */etc/serlink* (see section 5.5.1). The second serial port is again used as a console.

The test is carried out using the STREAM benchmark program. STREAM measures the sustained memory bandwidth by using long arrays that are much larger that the CPU's cache. It's code is also structured to prevent data-reuse [7]. The program is cross-compiled on the development host system and transferred to the a ramdisk on the OBC using the *rsync* protocol. The program is then executed on the OBC.

The results of the benchmark is shown in table 6.1. The different functions in the table represents different vector operations performed on its test data. The reader is referred to [7] for a detailed explanation. The import result of this test is that an indication of uncached memory access speed is given.

Conclusion

Evaluation of the command-line environment indicated that kernel is running and that sensible values are reported for the architecture parameters. By executing the benchmark program it was verified that external application programs can be executed in the software environment. It was also verified that large blocks of RAM can be accessed by programs. Finally an indication of the system's memory bandwidth was established.

6.1.3 NAND Flash

The purpose of this test is to verify that the mapping driver functions as expected.

The same serial port connections are used as for the operating system tests. The NAND driver as well as the drivers of the MTD stack are uploaded to the OBC using *rsync*. These drivers are then loaded with the following commands:

```
insmod mtdcore.ko
insmod mtdpart.ko
insmod nand_ids.ko
insmod nand_ecc.ko
insmod nand.ko
insmod mtd_blkdevs.ko
insmod mtdblock.ko
insmod crc32.ko
insmod jffs2.ko
insmod obc_nand.ko
```

After the mapping driver is initialised, it prints the following message to the kernel's message buffer:

```
Creating 1 MTD partitions on "NAND 256MiB 3,3V 8-bit":
```

The reported information agrees with the NAND chip's datasheet. This data is requested from the MTD stack by the initialisation code of the mapping driver; and requires the mapping functions to work. This, therefore, indicates that the MTD stack can successfully utilise the operations provided by the mapping driver. All further functionality is the responsibility of the MTD stack.

The JFFS2 filesystem is mounted with the following commands:

```
mknod n0 b 31 0
mount -t jffs2 n0 /mnt
```

Next the access speed is measured. Two 1 MB files are created on the development host. One is filled with binary zeros and the other is filled with random bits. These files were transferred to the ramdisk on the OBC. The following commands where used to create the files:

```
dd if=/dev/urandom bs=1k count=1k > randomnoise
dd if=/dev/zero bs=1k count=1k > allzero
```

Write speed was measured by copying these files to the */mnt* directory and measuring the transfer time. Reading speed was measured after rebooting the system to negate the effects of the kernel's file system cache. The same steps were followed to remount the partition, afterward the files was copied from */mnt* to the flash disk. The following is an example of the command used to copy the random data from the ramdisk to the NAND partition:

```
time cp /var/randomnoise /mnt
```

The integrity of the files recovered from the NAND partition was verified by computing their respective MD5 hashes and comparing it the that of the files on the development host.

Table 6.2: *Transfer speeds of NAND partition*

	Reading		Writing	
	Random bits	Zero bits	Random bits	Zero bits
Transfer time [s]	0.83	0.40	2.98	0.54
Approximate speed [MB/s]	1.2	2.5	0.34	1.9

The results of the transfer tests are displayed in table 6.2. It can be seen that, for the same data, reading is faster than writing. The difference in transfer speed between the two files are attributed to the compression algorithm that JFFS2 employs to compress stored and retrieve data. Using the same measurement technique, transfer rates of about 33 MB/s where observed when transferring data between files on the ramdisk. This indicates that ramdisk access time may have a small effect on the measurement.

6.1.4 CAN

The purpose of these tests are to verify that the CAN driver and userspace process function as expected.

The same serial port connections are used as for the operating system tests. The OBC's first CAN node is connected to a Ground Support Equipment Ethernet Interface (GSE-EI) unit. The GSE-EI functions as a bridge between the OBC's CAN bus and an ethernet network. A PC, running a session of SunSpace's Lua Command Interface (LCI), connects to the GSE-EI.

Verification of CAN node driver

The purpose of the first test is to establish that CAN packets can successfully be transmitted between the LCI software and the software environment on the OBC.

A arbitrary telecommand (TC) request packet is constructed and sent to the address of the LCI session. Sunspace's telecommand request FID consists of the hexadecimal value 06xx, with the lower 8 bits indicating the TC number. Packets are injected directly into the CAN drivers device node as follows:

```
echo 21 01 0621 00 11 22 33 44 55 66 77 > /dev/can0
```

In this example, a packet requesting telemetry channel 0x21 is sent from node 0x21 (Experimental OBC's address) to the LCI session (node number 1). An string of eight arbitrary payload bytes are added.

The activity log in the LCI graphical user interface (GUI) shows the structure of each packet that passes through it. It was used to verify that packets were correctly received. Arbitrary packets with different payload lengths were tested and confirmed to be correct.

Next packets were sent from the LCI session to the OBC. The received packets were directed to the OBC's serial console were they could be easily read and verified. The following command directs packets from the CAN driver to the standard output:

```
cat /dev/can0
```

Packets with different payload lengths were sent from the LCI GUI to the OBC. The received packets were verified to be correct.

This test was carried out on both of the OBC's CAN nodes. It shows that the CAN driver can successfully transmit and receive packets over the CAN bus.

Verification of CAN servicing process

To test if the CAN servicing process responds correctly, it is executed on the OBC and allowed to connect to the CAN device node. Telecommands were then sent from the LCI GUI instructing the process to change the state of registers on the SIC. The contents of the registers where read back to the LCI GUI using telemetry requests. To verify that the register contents were in fact altered, the procedure outlined in section 6.2.1 was followed.

This test was carried out during early development. Further testing was done as functions were added to the CAN process.

6.1.5 IP over CAN tunnelling

The purpose of this test is to verify that IP packets can successfully be passed between the experimental payload and the GSE-EI.

For this test the *tun* driver and proof-of-concept IP tunnelling program, *cantun*, is uploaded to the OBC. The *tun* driver is loaded and its device node created:

```
insmod tun.ko
mknod tun c 10 200
```

The CAN process on the OBC has to be stopped in order to allow another process to read from the CAN driver. The *cantun* process is then executed. The *cantun* process creates a virtual network interface, named *tun0*, using the *tun* driver. Next, the IP address of this network interface is assigned and the kernel routing table updated to use this interface as its default gateway:

```
ifconfig tun0 172.16.0.34 netmask 255.255.255.0
route add default gw 172.16.0.1
```

The addresses are assigned as follows : the 172.16.0.0/24 subnet is used by SunSpace for tunnelled connections. The *tun* interface's addres, 172.16.0.34, is one of the addresses assigned to the experimental payload. Finally, 172.16.0.1, is the address of the GSE-EI; which is the CAN network's gateway to the wider ethernet.

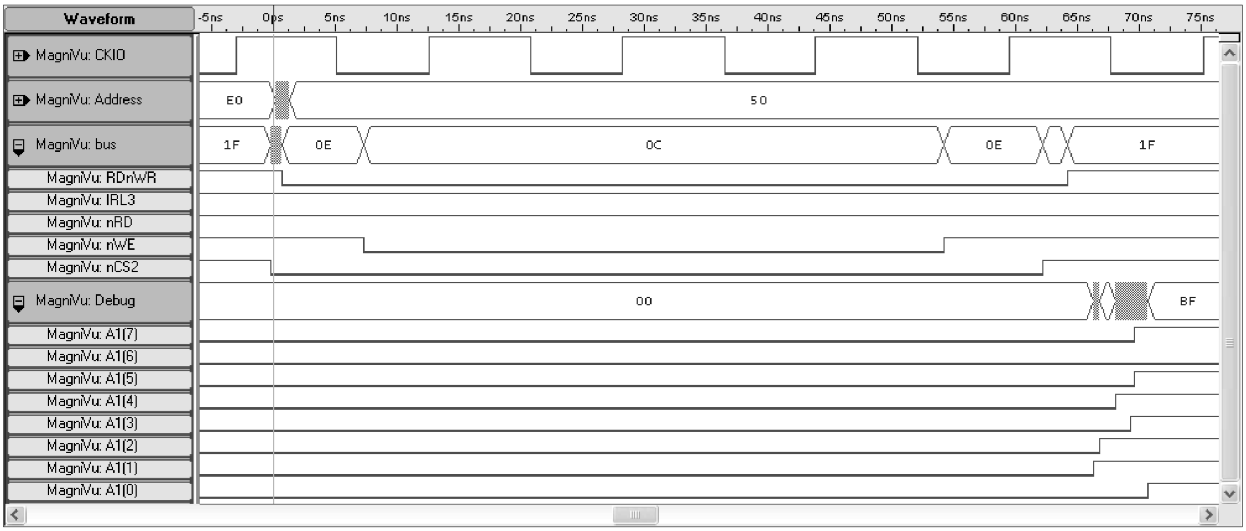


Figure 6.1: *Writing data into an FPGA register.*

The routing table of the PC running the LCI session is also updated to use the GSE-EI as gateway to the CAN network.

Ping packets could successfully be sent from the PC to the experimental payload. This indicates that IP packets can successfully be transmitted between them.

To measure the speed of the connection and test whether it remains stable when large transfers occur; a file was transferred from the PC to the OBC's ramdisk. A file with a much larger size than the CAN driver's buffer was used, the size of that file was 1,134,096 bytes. The *rsync* program was used to the transfer the file. The operation was successfully complete with a reported transfer rate of 26 kb/s.

These tests indicate that IP packets can successfully be tunnelled over the CAN bus to the experimental payload.

6.2 Firmware

The same serial port setup as in section 6.1.2 is used for the following tests. Additionally a Tektronix TLA 3013B logic analyzer was used to measure signals on the daughterboard's debug port and the SH4's bus. A Tektronix 5202 oscilloscope was used to measure the analogue signals.

Unless otherwise noted, the IQ subsystems were clocked at the originally specified rate of 1.2288 MHz.

6.2.1 Register Input and Output

The purpose of this test is to verify that the OBC can write data into a register on the FPGA and successfully read back the contents.

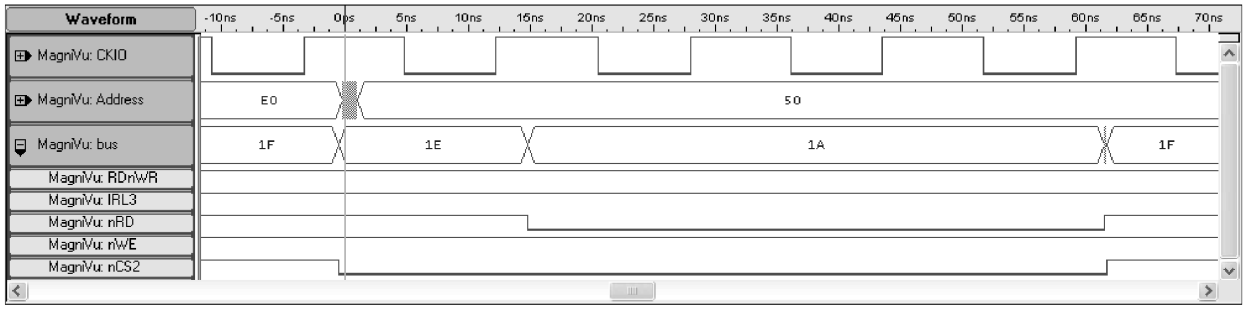


Figure 6.2: *Reading from an FPGA register.*

A control register is instantiated in the FPGA and its outputs are routed back to a status register (see section 4.2.3). The lowest eight bits are also routed to the debug pins on the daughterboard. The address of both the control and status registers, in hexadecimal, is 50. Its register number, in decimal notation, is therefore 20 (see section 5.3.1).

Values are written to the register using the low-level register access driver. The result is viewed on the logic analyzer. The following example was used to create figure 6.1:

```
echo bf > /dev/x20
```

Next, the contents of the register is read back:

```
dd if=/dev/x20 count=1
```

This operation was captured on the logic analyzer and the important bus signals reproduced in figure 6.2. These tests were performed with various values to verify the operation off all 32 bits.

Figure 6.1 shows how the bus reacts to the write operation, as well as the transition of the output on the debug pins. The value in the register before the operation was all zeros. The register is latched on the rising edge of the *nWE* signal and it can be seen that the output transition finishes about one cycle, of the bus clock *CKIO*, later. The different arrival times of the output signals are attributed to routing induced delays in the FPGA—specifically in the clock inputs to the register blocks and their outputs.

These tests verifies that data may be written to, and read from, registers on the FPGA. Additionally it verifies that the low-level register access driver functions correctly by showing that the correct signals are observed on the SH4's bus.

6.2.2 Peripheral ADCs

The purpose of this test is to verify that data can be transferred from the peripheral ADCs to the FPGA and the SH4.

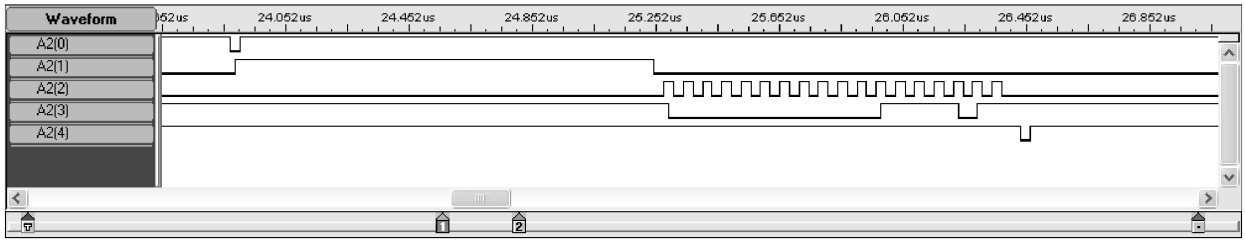


Figure 6.3: *Peripheral ADC signalling.*

Table 6.3: *Signal descriptions for logic analyzer output.*

Signal	Description
A0	Conversion-start pulse. Driven low by the FPGA to signal the ADC to start a conversion cycle.
A1	Busy signal. Driven high by the ADC to indicate conversion in progress.
A2	Serial data clock. Driven by the FPGA to shift data over serial data line. Rising edge indicates next bit should be sent.
A3	Serial data output. Driven by the ADC.
A4	Output ready. Internal FPGA signal—strobed low to indicate that a new sample is ready.

For this test, the pertinent data lines between the FPGA and the peripheral ADCs are routed, in the firmware, to the debug pins. The debug pins are sampled with the logic analyzer while samples are acquired from the ADCs.

The conversion process was initiated using the low-level register access driver and the command line interface on the serial console. The FIFO buffer was emptied beforehand by asserting, and then clearing, the reset bit in the peripheral control register *x1*. The following command was used :

```
echo 10f > /dev/x1
echo f > /dev/x1
```

The captured signals are reproduced in figure 6.3. For clarity, only the serial data input from a single ADC is reproduced, all four inputs are read simultaneously using four separate FPGA input. Table 6.3 maps out the relationships between the designations in the figure and their signal functions.

A complete sampling cycle can be observed in figure 6.3: The conversion-start pulse initiates the cycle, shortly afterward the ADC assert the busy signal. At the end of conversion, the busy signal is lowered and data is written out. The FPGA generates 18 pulses, one for each of the 18 bits, and the corresponding bit is returned by the ADC over the serial data

line. After all the bits have been read into the internal shift register, an output-ready pulse is generated to clock the sample into the FIFO buffer.

Next, the sampled values were read into the SH4, and displayed in hexadecimal on the console terminal. The values were compaired to the bit sequence observed on the logic analyser and their correctness verified. The commands used to read the samples from the four respective ADCs' FIFO outputs (register numbers 32–35), are as follows:

```
dd if=/dev/x32 count=1 | hexdump
dd if=/dev/x32 count=1 | hexdump
dd if=/dev/x32 count=1 | hexdump
dd if=/dev/x32 count=1 | hexdump
```

The above operation was carried out twice. The second set of samples were also compaired to the bit-sequences of the second sampling cycle (captured on the logic analyzer). This was done that to verify that the FIFO successfully buffered the incoming samples.

The conversion-start pulse's period was measured with the logic analyzer (not shown) and confirmed to be 10μs. This verifies that the 100 kHz sample rate is correctly generated. Finally, the test was also carried out with different subsets of the ADCs enabled to verify that selected ADCs may be disabled during sampling (see section 4.3.1).

This test established that samples are correctly retrieved form the peripheral ADCs, that the samples are correctly stored in the FIFO buffer and that the stored samples can be read back to the SH4.

6.2.3 Peripheral Sampling system

The purpose of this test is to verify that a continuous stream of samples can be acquired from the peripheral sampling system. Most of the firmware functions has been verified in section 6.2.2, the only addition function that has to be verified is the FIFO buffer's trigger.

A signal generator was used to provide a known waveform. On the prototype board, the analogue inputs to the peripheral banks were accessible through a connector on the daughterboard. The signal generator waveform was applied to one of these inputs at a time; and the peripheral sampling driver was used to acquire samples. Sampling was initiated on the command line, in the serial console, as follows:

```
dd if=/dev/padc1 count=10000 > /var/outdata
```

The above command requests 10,000 sample groups, which amounts to 1 second of sampled data. The output is written to the file */var/outdata*. After the command finishes, the output data file is transferred, using *rsync*, to a workstation where the data is plotted using a Python script. The test was executed during development and each of the inputs on both banks were verified. Figure 6.4 shows the result of one of these measurements using a 1 kHz test tone.

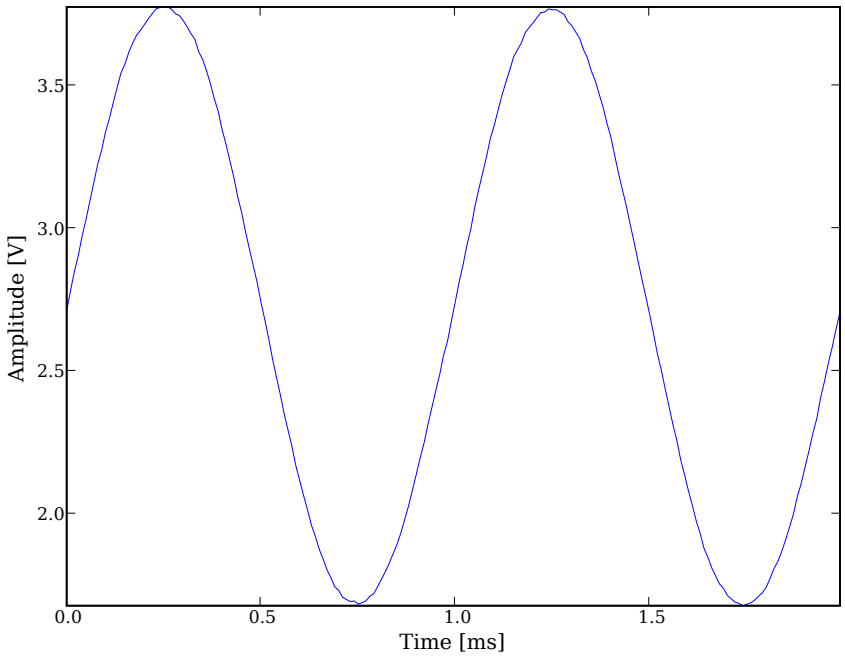


Figure 6.4: *Sampling a 1 kHz sinusoid on a peripheral ADC.*

During integration testing at SunSpace, the peripheral banks were also tested using the actual experiments as inputs. Figure 6.5 shows result of sampling the Magnetosphere experiment's x, y and z inputs[1]. The coils were excited using a fourth coil connected to a signal generator. The integration tests were conducted using the CAN bus to issue the capture requests and to return data.

This test showed that continuous sample stream can be extracted using the peripheral driver. The functioning of the driver depends on the FIFO trigger. This test therefore verifies that the peripheral sampling system functions correctly.

6.2.4 IQ DACs

The purpose of this test is to verify that the IQ DAC outputs can be controlled from the SH4.

A sample value will be written into the FIFO buffer, thereafter the DAC will be switched on and allowed to consume the sample. Before each value is written into the FIFO, the buffer will first be cleared. The output voltage at the buffer amplifier will be measured with an oscilloscope. Using the low-level register access driver, the following commands where

[1]The E-probe's amplifier was too sensitive to be tested on the workbench, it was subsequently grounded and did not produce a waveform.

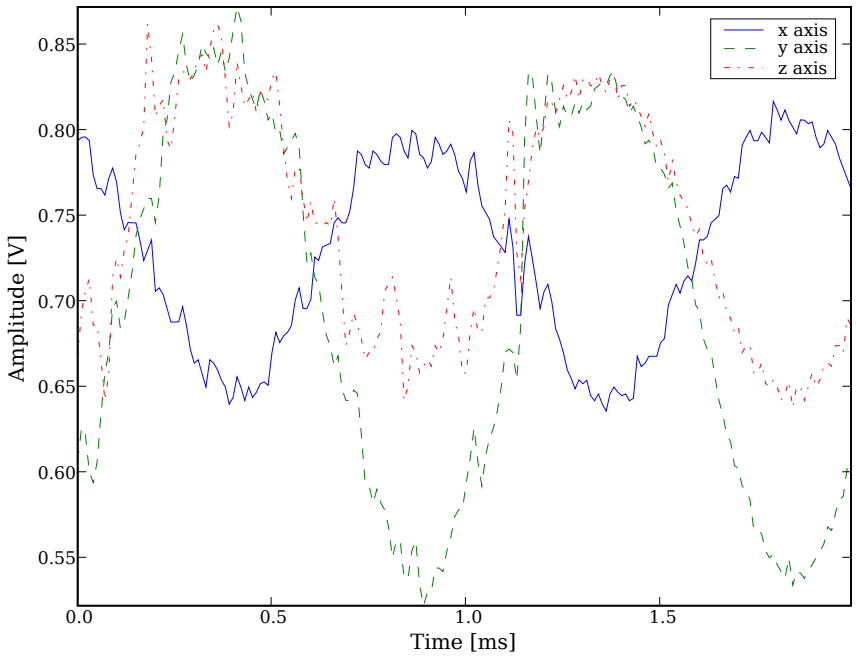


Figure 6.5: *Sampling the Magnetosphere experiment's magnetic sensors.*

executed for each sample value:

```
echo 0 > /dev/x8
echo 2 > /dev/x8
echo 7fff > /dev/x10
echo 3 > /dev/x8
```

The first command resets the FIFO buffer and the second enables the buffer by setting the appropriate bits in the IQ DAC control register. In the third line a value of 3FFF, in hexadecimal notation, is written into the FIFO buffer. The last command turns the IQ DACs on, the DACs will continuously output the same sample value.

The above procedure was carried out with input values covering the full scale output of the DACs. In the example only the I channel is given a value, since the Q channel is implicitly given a zero value (see table 4.1). Both channels were, however, verified.

Table 6.4 shows the result of the measurement. The input code to the DAC is calculated by performing the inverse of the operation described in section 4.3.3, in other words, the most significant bit is inverted and the result right shifted two places. The DAC's expected output voltages was calculated according the formula given in [22]. That result had to be scaled by a factor of two, to compensate for the amplification of the buffers on the SIC. The expected voltage can therefore be expressed as : $v_{out} = \frac{5n}{16384}$

The results clearly show non-linear behaviour at voltages above 4V. It is believed that

Table 6.4: *Measured DAC output voltage.*

Driver input value	DAC input code	Measured [V]	Calculated [V]
0x7FFF	16383	4.22	5.000
0x6FFF	15359	4.22	4.687
0x5FFF	14335	4.21	4.375
0x4FFF	13311	4.04	4.062
0x3FFF	12287	3.74	3.750
0x1FFF	10239	3.12	3.125
0x0	8192	2.52	2.500
0xEFFF	7167	2.23	2.187
0xCFFF	5119	1.62	1.562
0xAFFF	3071	1.01	0.937
0x8FFF	1023	0.4	0.312
0x8000	0	0.1	0.000

this is caused by the output buffers being operated too close to their supply voltage. The rest of the measured data conforms to expectations and therefore confidently suggests that the firmware behaves as expected.

6.2.5 NCO

The purpose of this test is to verify that the NCO can generate sinusoidal outputs at the DACs; and that the parameters of those signals can be adjusted.

The low-level register access driver is used to control set the NCO's input parameters. An oscilloscope will be connected to the DAC output buffers and the captured signals will be examined.

The following commands were used:

```
echo 31 > /dev/x8
echo 400 > /dev/x14
echo 4000 > /dev/x15
echo 13301330 > /dev/x18
echo 0 > /dev/x19
```

The first command enables the NCO and DACs; and sets up the DACs to receive samples from the NCO. The second command sets the output frequency to 1.2 kHz, the third sets the phase difference between the I and Q channel to $\frac{\pi}{2}$ radians. Next the output amplitude of both channels are set, and finally the DC offset is set to zero. The result of these commands on the output are shown if Figure 6.6.

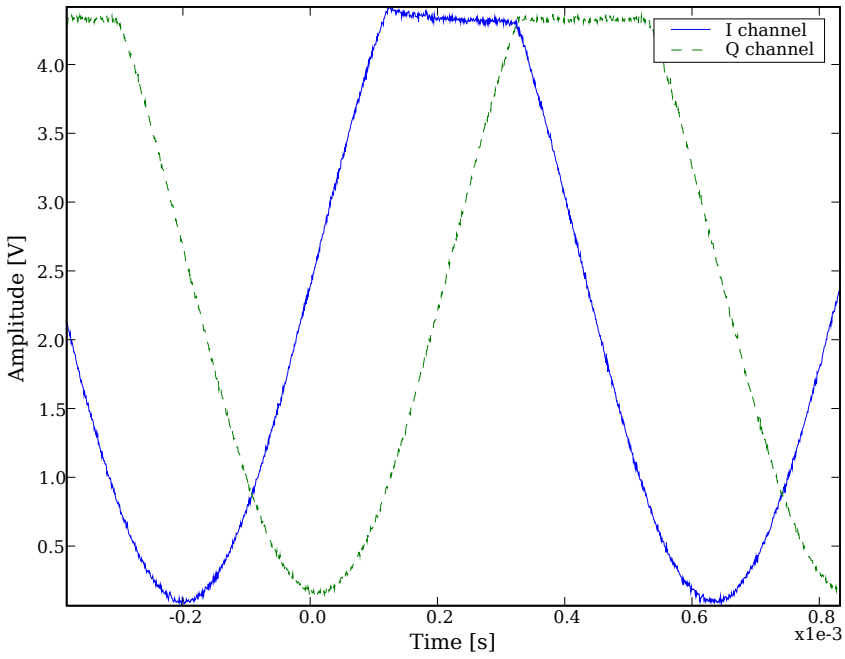


Figure 6.6: *Quadrature output of the NCO.*

The waveform in figure 6.6 clearly shows the clipping observed in section 6.2.4. Figure 6.7 shows the result of the test with the parameters adjusted as follows:

```
echo 755 > /dev/x14
echo 8000 > /dev/x15
echo 1c300630 > /dev/x18
echo 2b00 > /dev/x19
```

The increased frequency (2.2 kHz) was verified using the oscilloscope. The new phase difference and DC-offset is also clearly visible in the figure. Finally it can be observed that the I channels amplitude is smaller that that of the Q channel.

In this test it was verified that the NCO can generate sinusoidal signals at the DAC, It was further verified that the parameters of the generated signals are correctly adjustable.

6.2.6 IQ sampling

The purpose of this test it to verify that a continuous stream of samples can be acquired from the IQ ADCs.

A quadrature signal will be applied to the inputs of the ADC. The busy-waiting IQ sampling driver will then be used to acquire samples from the ADCs. The SDR switching network (see section 3.2.2) was used to route the output of the NCO to the input of the ADCs. This provides a known waveform at the input without the need for addition equipment.

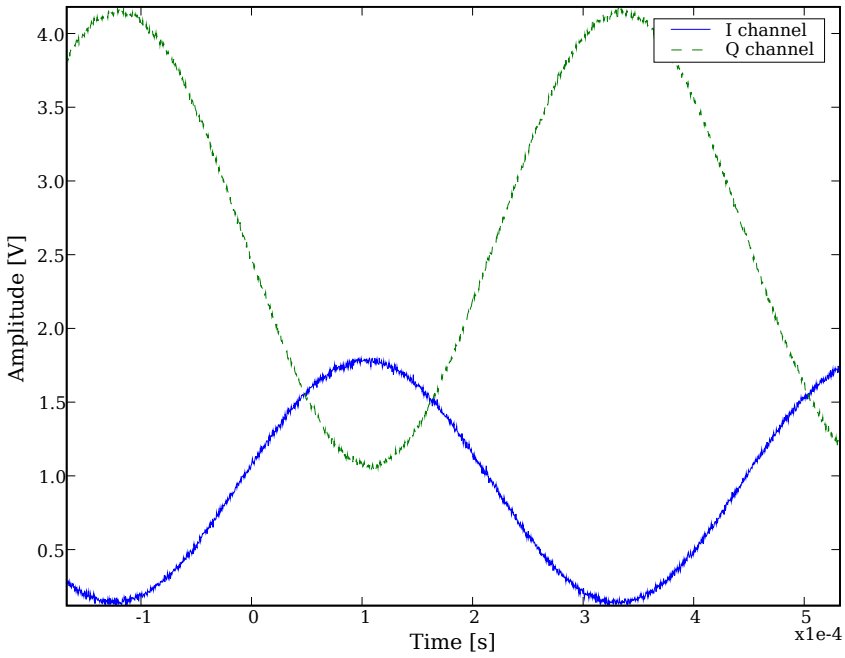


Figure 6.7: *NCO output with altered, phase, frequency and amplitude and DC-offset.*

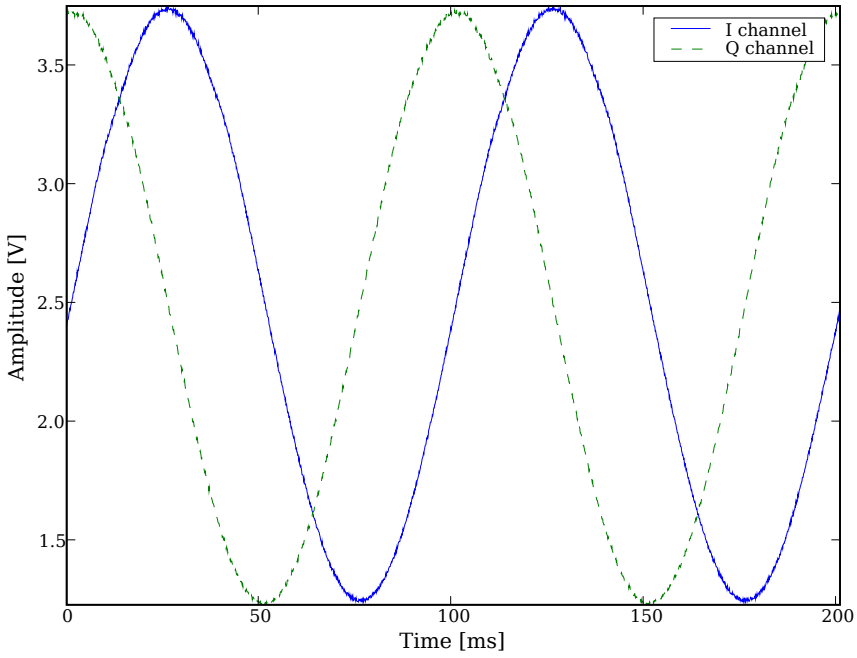


Figure 6.8: *Quadrature signal sampled on the IQ ADCs.*

The procedure used to sample the signals is the same as that used in section 6.2.3. In stead of the peripheral driver, the IQ sampling drivers were used. These drivers were still under development, the tests therefore required manual loading of the drivers and creating of their device nodes on the ramdisk:

```
insmod simpleadc.ko
mknod adc c 254 0
dd if=adc count=1000 > adcout
```

Two quadrature signals with a frequency of 10 Hz was applied to the ADCs. The amplitude of the signals was adjusted to prevent clipping. The result was transferred to a workstation using *rsync* and is presented in figure 6.8.

Sampling with the busy-waiting driver produces correct results. Using the interrupt driven driver generated overflow warnings. It is possible that the added overhead of waking up the blocked process might be responsible. This, however, indicates that it is unlikely that the OBC will be able to process samples at the originally specified sample rate (1.2288 MHz)

This test shows that sample streams can successfully be acquired on the IQ ADCs, but that processing those samples might not be feasible at the originally specified sample rate.

6.2.7 IQ transmission

The purpose of this test is to verify that a stream of samples can be transmitted over the IQ DACs.

For this test, waveforms will be generated in the OBC and transferred to the IQ DACs using the IQ transmission drivers. An oscilloscope will be used to capture the output of the DACs. The drivers for this test were manually loaded, as was done in section 6.2.6. Two simple programs were written. The first generated samples for a ramp waveform, the second generated sinusoidal samples. Only one set of samples are generated, which is then copied the inputs of both DACs.

Neither of the test programs were able to successfully generate an output at the 1.2288 MHz sample rate, regardless of which driver was used. Buffer undderruns where reported by both the busy-waiting, as well as the interrupt driven driver.

In stead of spending time attempting to optimise the programs, it was decided to lower the sampling rate for the following tests. An sample rate of 9.6 kHz was selected (see section 6.3).

At the 9.6 kHz samplerate both programs successfully generated signals at the DACs. The results are reproduced in figures 6.9 and 6.10.

This test showed that the IQ transmission system functions correctly, at a reduced speed. The firmware components has already been verified by previous tests, at full speed. The bottleneck in performance is therefore attributed to software on the OBC.

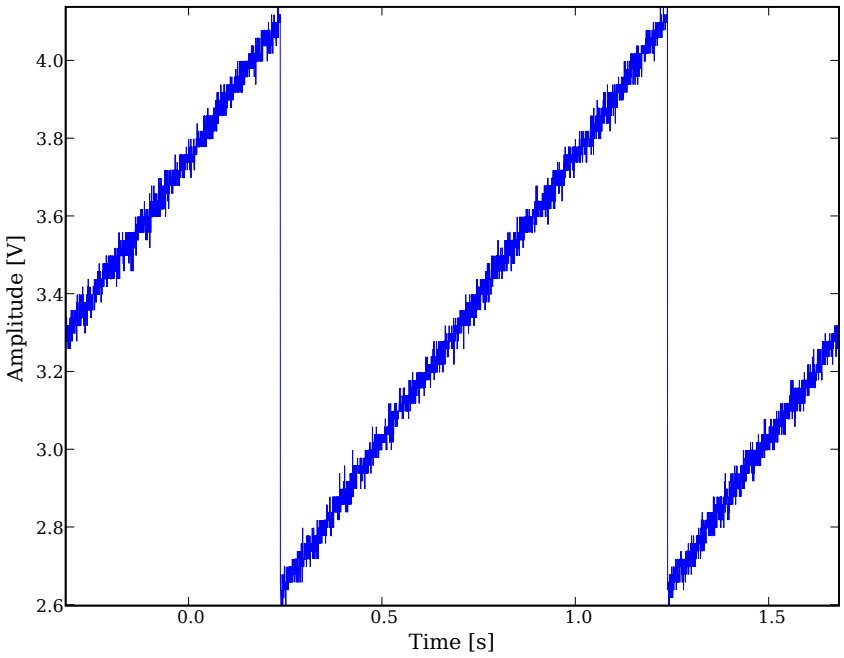


Figure 6.9: *Ramp waveform output, generated on the OBC.*

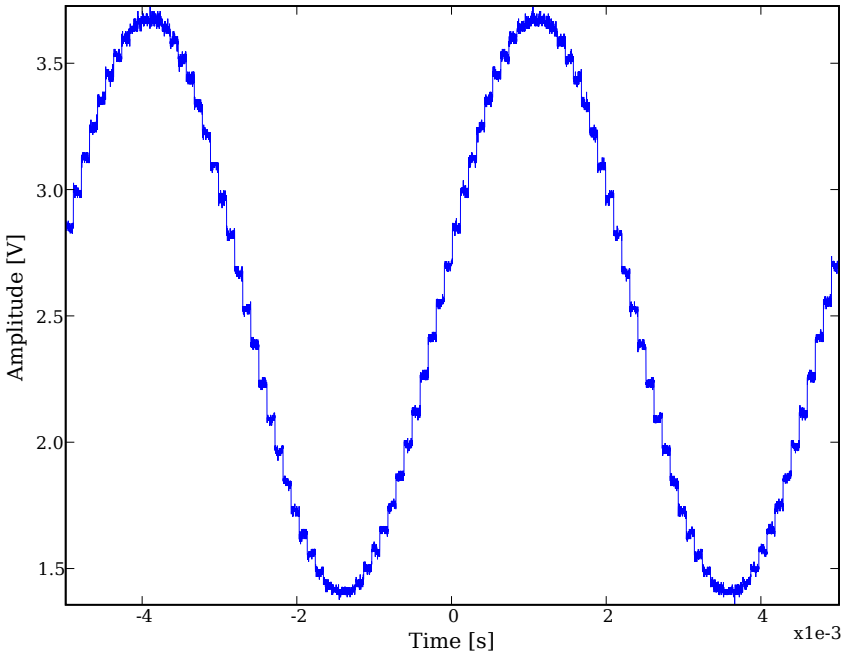


Figure 6.10: *Sinewave output, generated on the OBC.*

6.3 SDR Proof-of-concept application

In this section the use of the platform for SDR will be demonstrated. To save time, a well tested modulation scheme, with already implemented software, was selected. The *afsk* (which stands for audio frequency shift keying) modem from the *soundmodem* program, written by Thomas Sailor, was selected. The modulation scheme is well known among radio-amateurs since is can be used for packet radio over an ordinary audio channel, without needing modifications to the radio equipment.

This modem was chosen as it is the simplest modem provided by the package. It has low processing speed requirements from the host system[2] and it can operate at the low sample rate of 9.6 kHz when running at 1200 baud.

Furthermore, the modulation scheme implemented by this modem is simple binary frequency shift keying (FSK), using frequencies of 1.2 kHz and 2.2 kHz to represent its two states. This can easily be generated by the NCO and the use of the NCO as a modulator can therefore also be demonstrated.

The full functionality of *soundmodem* was not required for this test. Because *soundmodem* is written in a modular way, it was possible to extract the modulator and demodulator code and wrap it in small programs. The program named *mod* accepts a stream of binary data on its standard output and returns a stream of modulated samples on its standard output; the *dem* program does the inverse. An option was also added to these programs to use the IQ device nodes on the OBC in stead of the standard output and input. Finally an Open Sound System (OSS) version of the programs was created to allow the applications run on workstation PCs, these were called *modoss* and *demoss*.

The Higher-Level Data Link Control (HDLC) protocol's framing code was also extracted from *soundmodem* and wrapped into programs name *frame* and *unframe*. The *frame* program, uses the unique sequence, 7E in hexadecimal notation, to frame packets incoming packets. Bit stuffing is applied to prevent the framing sequence from occurring inside the packets [11, p. 8]. The *unframe* program does the inverse of the frame program. A Cyclic Redundancy Code (CRC) is used to identify and reject damaged packets.

6.3.1 OBC to PC using the IQ transmission path.

In this test data will be transmitted from the OBC, using one of the IQ DACs, to a workstation PC's sound card. The DAC's output was low-pass-filtered using a 4^{th} order Butterworth filter with a cut-off frequency of 3.2 kHz. The filter was built using two LF351 op-amps in a Sallen-Key configuration. The output of the filter, with the modulated singal applied as input, is shown in figure 6.11. The filter's output is applied at the line-input of the PC's sound card.

The same debug pins as in section 6.2.1 were instantiated for this test. To measure the

[2]On a 2 GHz AMD Athlon workstation, the demodulator requires less then than 1% of the CPU's time.

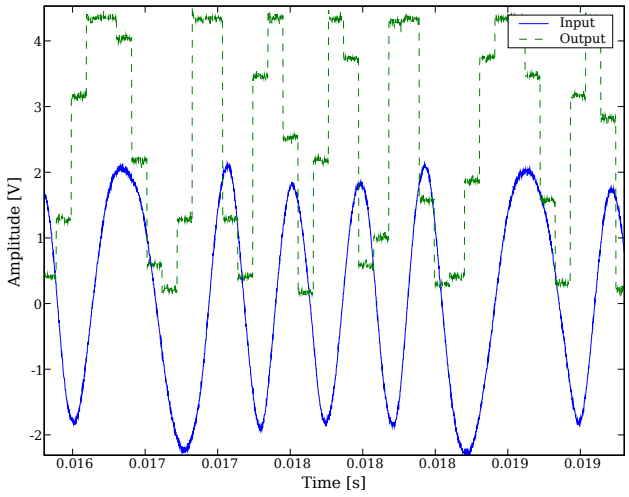


Figure 6.11: *Input and output of the reconstruction filter.*

Table 6.5: *Debug signal descriptions for modem tests.*

Signal	Description
Busy	Busy-waiting cycle in progress.
User	Transfer of data between userspace and kernel space in progress.
Fifo	Transfer of data between FIFO buffer and kernel space in progress.
Read	Userspace process suspended in in *read* system call.
Write	Userspace process suspended in in *write* system call.
Irq	Interrupt handler executing in progress.

IQ driver's performance, functions were added to the driver code which enabled toggling of these pins. The important stages in the drivers (see figures. 5.2, 5.5 and 5.6) where given pins to toggle upon entering and exiting.

This test was carried out with both the busy-waiting driver as well as the interrupt driven driver. The state of the debug pins were sampled for both cases using the logic analyzer. Additionally the interrupt request (*IRL3*) and chip select (*nCS2*) lines for the SIC were also captured. The states represented by the debug pins are as described in table 6.5

A large file, containing human-readable text, was copied to the OBC and the modulator subsequently started using the following commands:

```
cat longtextfile | ./frame | ./mod dac
```

The following command was then issued in a terminal on the PC:

```
./demoss | ./unframe
```

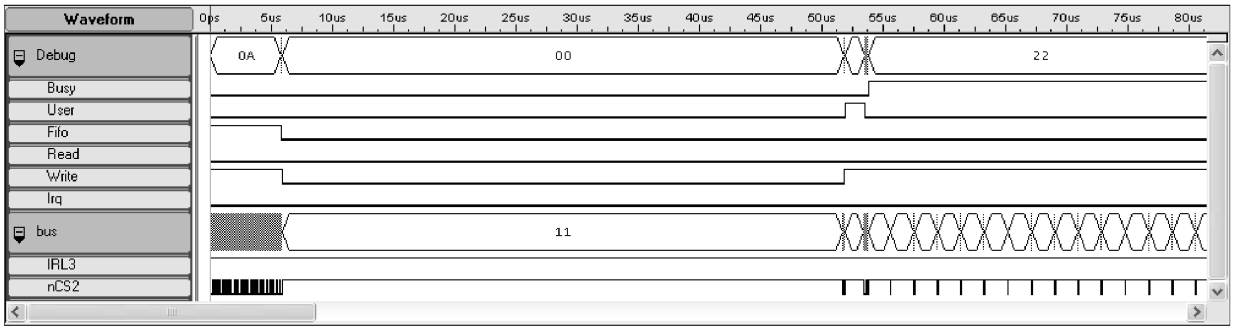


Figure 6.12: *Busy-waiting IQ transmission driver states.*

Readable text was streamed into the terminal on the PC, verifying that the data was modulated and demodulated successfully.

Figure 6.12 shows the result of the test using the busy-waiting driver The logic analyzer was triggered on the falling edge of the busy signal, the graph, therefore starts at the point when new space becomes available in the FIFO buffer. The time it took to copy new samples into the FIFO was measured at about 5.8 μs, the size of the buffers was 256 bytes. From this information the transfer rate of data to the SIC was calculated at about 45 MB/s. This means that, according to the results in section 6.1.2, access to the SIC is slower than access to main memory. After the write-bit is lowered, control is returned to the user space process (modem), The write-line is asserted moments later as the process attempts to transfer its next batch of samples. The following copy operation, from user space to kernel space, happens much faster; this is because that operation is cached in the CPU's data cache. After the copy operation, the driver enters its busy-waiting loop. The periodic polling of the FIFO status register can be observed by examining the chip-select line.

In figure 6.13, the result of the test using the interrupt driven driver is shown, here the logic analyzer was triggered on the falling edge of the interrupt-request line. The copy operations follow a similar pattern as for the busy-waiting driver, in that the cached operation is much faster. It can be seen that, after the FIFO buffer is filled, the interrupt is canceled in the SIC. A fairly long delay is observed before the interrupt handler exits. This delay is believed to be related to waking the sleeping *write* system call. It will, therefore, stay constant when longer buffers are used. The moment the sleeping system call (for the next samples) is revived, the user buffer copy operation is observed; afterwards the system call finishes. When the next system call arrives, no polling action is observed because the calling thread is put to sleep.

6.3.2 PC to OBC

In this test a modulated signal will be generated at the PC's soundcard output and fed into the input of the IQ ADC's. No low-pass filter was required for this test as the output

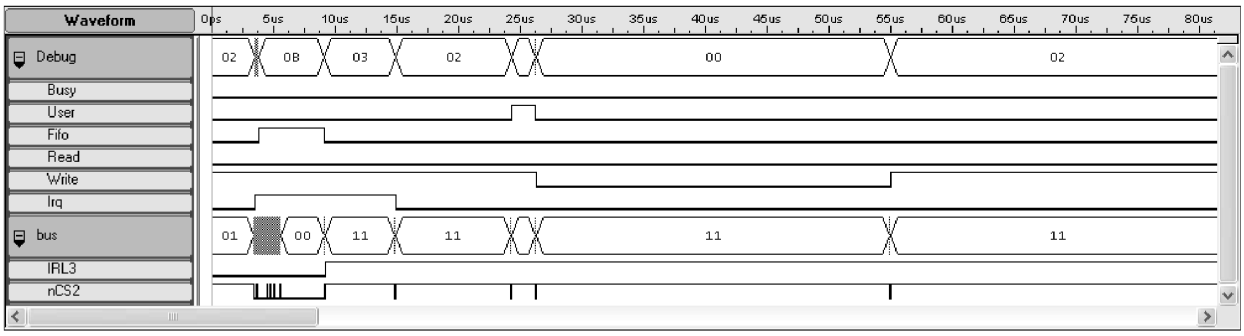


Figure 6.13: *Interrupt driven IQ transmission driver states.*

from the soundcard is sufficiently low-pass-filtered already. The same debug pin setup as in section 6.3.1 was used. Figure 6.14 shows the signal sampled on the SIC's ADC.

Another large, human-readable, text file was used in this test. The modulator was started on the PC with the following command:

```
cat longtextfile | ./frame | ./modoss
```

The following command was then issued on the OBC using the serial terminal:

```
./dem adc | ./unframe
```

Readable text was streamed into the serial terminal. This confirmed that signals, received over the ADC, could be demodulated on the OBC. This test was successfully carried out with both the busy-waiting and the interrupt-driven drivers.

Figure 6.15 shows the result of the test using the busy-waiting driver, the logic analyzer was triggered on the falling edge of the busy. The FIFO buffer is copied to the kernel buffer, and immediately copied to user space thereafter. The transfer times for the copy operations are similar to that of the IQ transmission case. The rest of the driver's behaviour is similar to the transmission driver's.

In figure 6.16, the result of the test using the interrupt driven driver is displayed. The logic analyzer was triggered on the falling edge of the interrupt-request line. The behaviour of is similar that of the transmission driver. The main difference being that the samples copied from the FIFO are the same samples being copied to user space afterwards.

6.3.3 OBC to PC using the NCO

The purpose of this test was to verify that the NCO can be used to transmit modulated data.

The same low-pass filter as in section 6.3.1 was used. A simple program, called *ncomod*, was written to replace the *soundmodem* modulator. This program uses the proof-of-concept

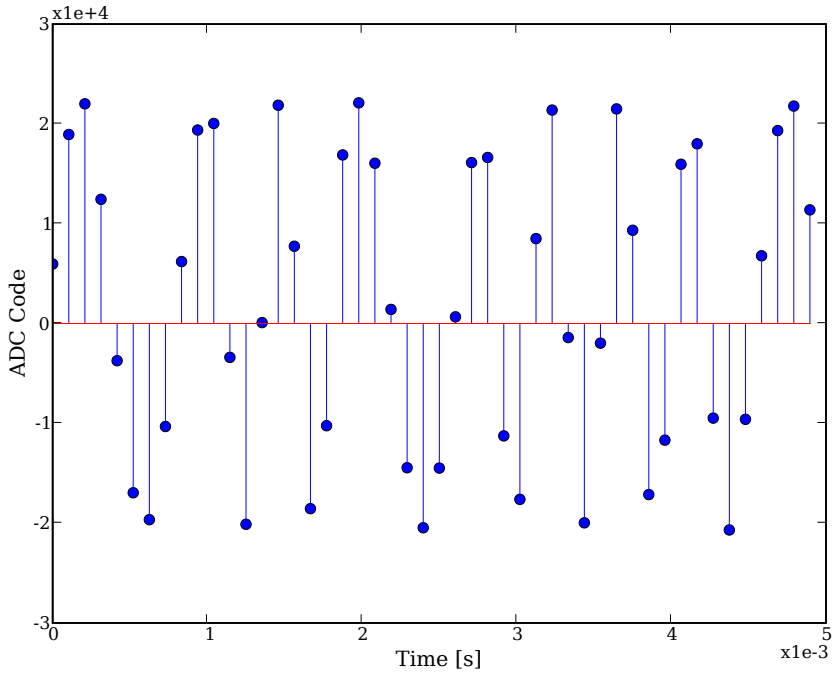


Figure 6.14: *Stem plot of FSK signal sampled on the daughterboard.*

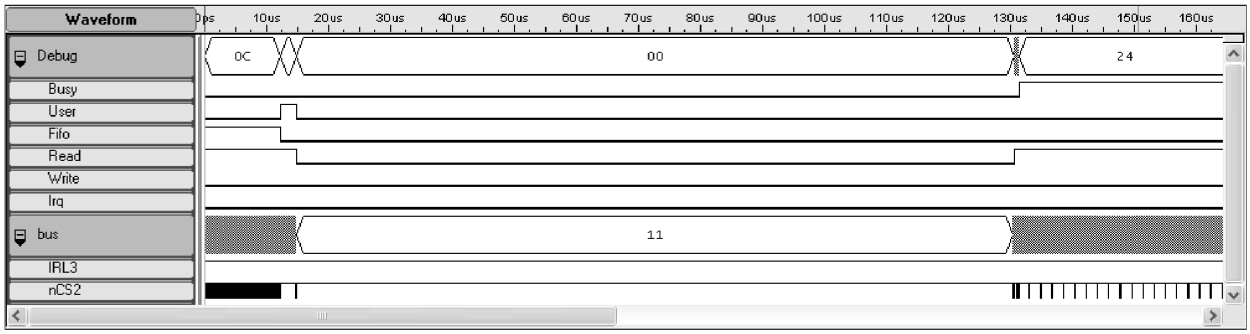


Figure 6.15: *Busy-waiting IQ sampling driver states.*

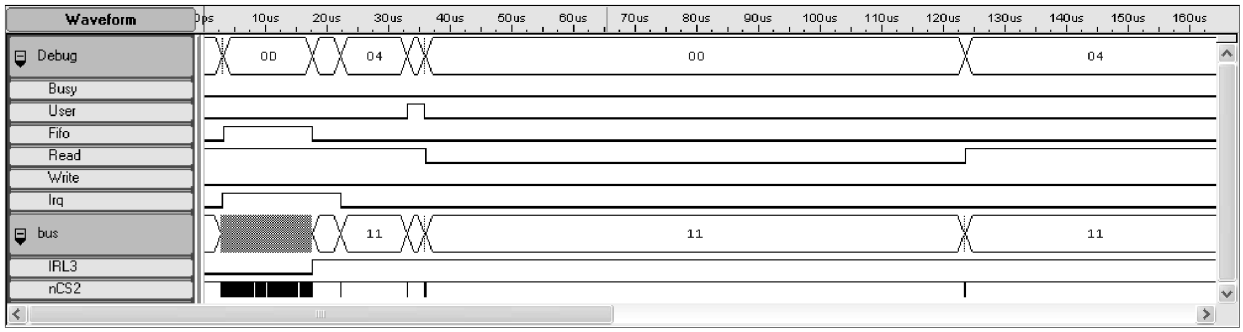


Figure 6.16: *Interrupt driven IQ sampling driver states.*

NCO driver discussed in section 5.3.3. The program accepts input data at the standard input. One bit of this input data is shifted out to the NCO driver per *write* cycle.

The NCO driver was loaded on the OBC and a device node, *nco*, was created. The transmission of data was then started on the OBC with the following command:

```
cat longtextfile | ./frame | ./ncomod nco
```

The signal was demodulated in the same way as for the IQ transmission version of this test. Data was successfully transmitted to the PC.

6.3.4 Result

Data was modulated on the OBC and transferred to a workstation PC's sound card. The signal was successfully demodulated on the PC. This test was carried out using the IQ transmission path, as well as the NCO path on the daughterboard. Signals generated on the PC were also successfully demodulated on the OBC using the IQ sampling system. These tests therefore illustrated how the software platform can be used for SDR applications.

Chapter 7

Conclusion

This thesis described the design and implementation of the experimental payload software on board a micro-satellite. This chapter gives an overview of the conducted work. The strengths and weaknesses of the implemented systems will also be discussed and possible future enhancements will be highlighted.

7.1 Thesis Results

The principles of SDR and the software framework for the SP Group's research was examined in Chapter 2. It was determined that SU SDR require a POSIX-compliant operating system. Linux was selected as operating system kernel for the OBC. A software environment, based on the Linux kernel, was then implemented on the OBC and its functionality verified—achieving the first objective of this project.

The software requirements of the experimental hardware were considered in the first part of Chapter 3. A firmware architecture was devised to meet those requirements. The necessary firmware components for the FPGA were designed, and their implementation discussed in Chapter 4. Device drivers needed to access those components were designed in Chapter 5, thereby achieving the second and third objectives. These systems were tested and verified to function correctly. A performance bottleneck was identified in the IQ sampling and transmission subsystems. In order to conduct further tests, the sampling rate of those systems where reduced.

Use cases for the experiments were examined in the last part of Chapter 3. The components of the experimental payload were partitioned into functional blocks. Actions carried out by the satellite operators on those blocks were defined. The device drivers necessary to perform those actions were implemented and an example program was created to integrate all the systems. This program, which provides access to the payload using a CAN bus, was used by the integration team. The fourth objective was thereby reached.

In the last part of Chapter 6, a proof-of-concept SDR application was constructed. It was successful at transmitting data using the IQ transmission system and receiving over the IQ sampling system. The final objective was therefore reached.

7.2 Future Improvements

The system was implemented according the objectives laid out in Chapter 1. Several areas were, however, identified during the course of this project where improvements could be made. The following is a summary of those:

- The register module was found to be unreliable when many of them were instantiated on the FPGA (see section 4.2.1). The method used to drive the latches was found to be flawed; the bus timings were adjusted to compensate. Alternative options for address decoding and triggering of the latches should be examined, and the register module subsequently improved or rewritten.

- Accessing the daughterboard's memory segment was found to be slower than accessing the SDRAM. Options for increasing access speed can be investigated. Analisis of the timing requirements for the firmware components that connects to the bus will be necessary. The bus speed can be increased by using faster settings in the SH4's bus state controller.

- The current IQ drivers has to copy samples twice before the SDR applications can access them; once from the FIFO to the kernel buffer, and once from the kernel buffer to user space. More advanced techniques can be implemented to reduce this number to zero. Using the *mmap* system call would allow user space programs to have direct access to the kernel buffers without first requiring the driver to copy them. Implementing direct memory access (DMA) for the SIC would allow the samples to be transferred to the kernel buffers without requiring the CPU to explicitly copy them.

- The proof-of-concept NCO driver can be improved by allowing the update-rate to be dynamically adjusted. This can be done with the use of an *ioctl* system call.

- The drivers for the NCO and IQ transmission operates on the same subsystem within the SIC. This requires that one is first removed before the other can be loaded. They can be integrated into a single driver which would remove the need for driver swapping.

- Registers on the experimental payload were assigned addresses, roughly, as they were implemented. Addresses belonging to the same subsystem can be grouped into memory regions. This will allow the drivers to claim memory regions and thereby prevent other, conflicting drivers, from claiming their registers.

- The proof-of-concept IP-over-CAN program can be integrated into the CAN service program and the kernel can be recompiled with pseudo terminals enabled. This will allow remote TELNET sessions over the CAN bus.

7.3 Final remarks

A software environment was created on the OBC which can be used to run the proposed experiments. An example program was written that allows the system to integrate with the rest of the satellite. Firmware and drivers were implemented that allow the platform to perform SDR functions. Finally, it was demonstrated that the software platform can be used to run SDR applications.

Bibliography

[1] "GNU Radio - The GNU Software Radio." http://www.gnu.org/software/gnuradio/. July 2007.

[2] "Linux kernel API." http://www.gnugeneration.com/books/linux/2.6.20/kernel-api/. July 2007.

[3] "LinuxSH." http://sourceforge.net/projects/linuxsh/. September 2007.

[4] "OpenCores." http://www.opencores.org. July 2007.

[5] "SDR Forum." http://www.sdrforum.org/. July 2007.

[6] "SOUTHERN AFRICAN AMATEUR RADIO SATELLITE ASSOCIATION." http://www.amsatsa.org.za/SZASAT.htm. July 2007.

[7] "STREAM Benchmark Reference Information." http://www.cs.virginia.edu/stream/ref.html. July 2007.

[8] ANDRAKA, R., "A survey of CORDIC algorithms for FPGAs." *Proceedings of the ACM/SIGDA sixth international symposium on field programmable gate arrays*, 1998.

[9] BRADY, R., "A Cross Platform Framework for Software Defined Radio." Master's thesis, University of Stellenbosch, 2006.

[10] CHAPIN, J., "Software Engineering for Software Radios: Experiences at MIT and Vanu, Inc." in *Software Defined Radio: Enabling Technologies* (TUTTLEBEE, W. (Ed.)), John Wiley & Sons, Ltd, 2002.

[11] COOKE, A., "Rural E-mail System for the Sumbandila Satellite." Master's thesis, University of Stellenbosch, 2007.

[12] CORBET, J. *et al.*, *Linux Device Drivers*. Third edition. O'Reilly Media, February 2005.

[13] CRONJE, J. J., "Software Architecture Design of a Software Defined Radio System." Master's thesis, University of Stellenbosch, 2004.

[14] CUMMINGS, M., "Radio Frequency Front End Implementations for Multimode SDRs." in *Software Defined Radio: Enabling Technologies* (TUTTLEBEE, W. (Ed.)), John Wiley & Sons, Ltd, 2002.

[15] Department of Science and Technology, "Announcement of Opportunity, Onboard Experiment on DST Micro Satellite." 2006.

[16] GATLIFF, B., "Embedding with GNU: The GNU Compiler and Linker." `http://www.embedded.com/2000/0002/0002feat2.htm`. July 2007.

[17] KEYES, R. W., "The Impact of Moore's Law." *IEEE solid-state circuits society newsletter*, September 2006.

[18] KOEKEMOER, J., "DST satellite experiment Interface Control Document." SunSpace, October 2005.

[19] LINEAR TECHNOLOGY. *LTC1403A Serial 12-Bit/14-Bit, 2.8Msps Sampling ADCs with Shutdown*, 2007.

[20] LUND, D., "Baseband Processing for SDR." in *Software Defined Radio: Enabling Technologies* (TUTTLEBEE, W. (Ed.)), John Wiley & Sons, Ltd, 2002.

[21] MARSH, D., "Software-defined radio tunes in." *Electronics Design, Strategy, News*, March 2005.

[22] MAXIM. *MAX5143 Serial-Input, Voltage-Output, 14-Bit DACs*, 2007.

[23] RAYMOND, E. S., *The Art of UNIX Programming*. First edition. Addison-Wesley Professional, September 2003.

[24] REED, J. H., *Software Radio: A Modern Approach to Radio Engineering*. First edition. New Jersey: Prentice Hall, May 2002.

[25] RENESAS. *Hitachi SuperH RISC engine SH7750 Series*, 2002.

[26] RUSHTON, A., *VHDL for Logic Synthesis*. Second edition. New York: John Wiley & Sons, 1998.

[27] STEVENS, W. R. and RAGO, S. A., *Advanced Programming in the UNIX Environment*. Second edition. Addison Wesley Professional, June 2005.

[28] STORMYRBAKKEN, C., "Automatic Compensation for Inaccuracies in Quadrature Mixers." Master's thesis, University of Stellenbosch, 2005.

[29] SUN SPACE AND INFORMATION SYSTEMS. *User Manual: SH4 Unit*, 2005.

[30] SUN SPACE AND INFORMATION SYSTEMS. *Generic Specification: CAN-bus Protocol*, 2006.

[31] VAN ROOYEN, G. J., "Research Proposal: ZaSAT1 "Pathfinder" SDR Experiment." University of Stellenbosch, 2006.

[32] VAN ROOYEN, G. J., "ZaSat-1 Experimental Payload, Interface Specification and Configuration of Experiments." University of Stellenbosch, February 2006. Preliminary Specification.

[33] YAGHMOUR, K., *Building Embedded Linux Systems*. First edition. O'Reilly Media, April 2003.

Appendix A

SIC register descriptions

Addresses depends on specific instantiation, was changed often as specification changed. Refer to the constant declarations in the top-level VHDL file for the latest addresses. Unshown bits are unused as are bits marked with the "–" symbol. These bits are ignored in the firmware (when written by host) and are their values are undefined when read by the host.

A.1 Peripheral Sampling Subsystem

Peripheral sampling control register

Bit:	8	7	6	5	4	3	2	1	0
Function:	$\overline{\text{RST}}$	INT	–	BS	$\overline{\text{TC}}$	CE3	CE2	CE1	CE0
Initial value :	0	0	–	0	0	1	1	1	1

Name	Definition
$\overline{\text{RST}}$	Peripheral subsystem is disabled and FIFO contents erased when set to 0. When set to 1 system is active.
INT	Interrupts are generated when set to 1.
BS	Bank Select. 0 - Bank 0 selected, 1 - Bank 1 selected.
$\overline{\text{TC}}$	When set to 1 output is formatted in straight binary, when set to 0 output is formatted as two's compliment.
CE3-0	Chip Enables for ADC3-0, Respective ADC is enabled when corresponding bit is set to 1.

Peripheral sampling FIFO status register

Bit:	3	2	1	0
Function:	GEQ	EQ	EMPTY	FULL

Name	Definition
GEQ	Set when FIFO is at or above trigger level.
EQ	Set when FIFO is at trigger level.
EMPTY	Set when FIFO is empty.
FULL	Set when FIFO is full.

Peripheral sampling FIFO level register

This is an 8-bit writable register that controls the point at which the FIFO trigger activates. The register is initialised to zero on reset. Bits 31–8 are ignored by the firmware when written to.

Output data format

All four output data registers have the same format.

Bit:	31–18	17–0
Contents:	Zeros	ADC Data

A.2 Quadrature Sampling Subsystem

Quadrature sampling control register

Bit:	7	6	5	4	3	2	1	0
Function:	FINT	–	–	–	–	–	$\overline{\text{FRST}}$	$\overline{\text{ARST}}$
Initial value :	0	0	0	0	0	0	0	0

Name	Definition
FINT	FIFO trigger generates interrupts when set.
$\overline{\text{FRST}}$	FIFO is reset when cleared and active when set.
$\overline{\text{ARST}}$	ADC are disabled when cleared and enabled when set.

Quadrature sampling FIFO status register

This register is identical to the peripheral sampling FIFO status register.

Quadrature sampling FIFO level register

This register is identical to the peripheral sampling FIFO level register.

Output data format

Bit:	31–16	15–0
Contents:	Q Data	I Data

A.3 Quadrature Transmission Subsystem

Quadrature transmission control register

Bit:	7	6	5	4	3	2	1	0
Function:	FINT	NINT	$\overline{\text{NRST}}$	SRC	–	–	$\overline{\text{FRST}}$	$\overline{\text{DRST}}$
Initial value :	0	0	0	0	0	0	0	0

Name	Definition
FINT	FIFO trigger generates interrupts when set.
NINT	NCO generates interrupts when set.
$\overline{\text{NRST}}$	NCO disabled when cleared (set to 0). NCO active when set.
SRC	Samples for DAC are sourced from FIFO when cleared and from NCO when set.
$\overline{\text{FRST}}$	FIFO is reset when cleared and active when set.
$\overline{\text{DRST}}$	DAC are disabled when cleared and enabled when set.

Quadrature transmission FIFO status register

This register is identical to the peripheral sampling FIFO status register.

Quadrature transmission FIFO level register

This register is identical to the peripheral sampling FIFO level register.

Input data format

Bit:	31–16	15–0
Contents:	Q Data	I Data

Appendix B

Source Code

The source code for the project is included on the accompanying disk. The directory structure on the disk is listed in table B.1

Table B.1: *Contents of the accompanying disk.*

Location	Source Code Description
afsk	Modem derived from soundmodem's afsk modulator.
base	Files created for SunSpace's LCI sessions.
boot	Bootloader.
can/app	CAN service program and files for child processes.
can/driver	CAN node driver.
can/tun	proof-of-concept IP over CAN tunnelling program.
sic/exp	Drivers for the experimental payload. (Low-level register access and peripheral ADCs).
sic/iq	IQ sampling and transmission drivers.
sic/nco	NCO proof-of-concept driver.
lvds	LVDS driver.
nand	nand flash mapping driver.
rootfs/root	primary root filesystem.
rootfs/ext	secondary filesystem.
hdl	VHDL firmware.