University of Technology, Sydney
Faculty of Engineering

Interface Electronics for the Australia Telescope Compact Array

by

Suzanne Amanda Jackson

Student Number: 97051597
Major: Telecommunications Engineering

Supervisor: A/Prof Sam Reisenfeld
Industry Co-Supervisor: Graham Moorey (CSIRO ATNF)

A 12 Credit Point Project submitted in partial fulfilment of the requirement for the degree of Bachelor of Engineering

February 2003

Synopsis

I have been employed by the CSIRO Australia Telescope National Facility (ATNF) for much of the duration of my studies towards the Bachelor of Engineering. Early in my career with the ATNF, it became evident that the AT dataset hardware which we had made great use of over the years would have to be re-thought, especially given the increasing complexity of upcoming receivers for the Major National Research Facilities (MNRF) upgrade.

The germ for this project was planted late one night at Parkes observatory, whilst faultfinding an interface unit for the Parkes conversion system, a module which provides extra functionality for an AT dataset. Much of the interface hardware was realised in programmable logic, and it occurred to us that it would be possible, and indeed useful, to re-engineer the core of the dataset functionality using programmable logic, so that it was no longer tied to a specific processor, or indeed set of hardware.

This thesis seeks to document much of the progress to date towards the goal of a simple, re-useable dataset engine, as well as the implementation, using this engine, of an interfacing system for the MNRF millimetre receivers on the Australia Telescope Compact Array. It covers some background information on the AT dataset protocol, the development of a VHDL model describing the protocol, and several instances of hardware making use of the protocol.

Of particular interest are the efforts at reducing RFI emanations from the receiver interface, as well as the design of an ultra low noise ADC subsystem for the water vapour radiometer.

Finally, the thesis documents some of the test software used along the way to prove system operation, as well as some of the systems integration and project management hurdles experienced throughout the course of the project.

Statement of Originality

This thesis is the result of work undertaken between 2000 and 2003 in the Department of Telecommunications Engineering at the University of Technology, Sydney, and the CSIRO Australia Telescope National Facility.

Work on the Australia Telescope dataset protocol is based upon work carried out at the Australia Telescope National Facility between 1985 and 1990, principally by Richard Ferris. Further developments on this work were carried out in consultation with Mr Ferris, and George Graves, of the Australia Telescope Receiver Group.

Mechanical fabrication of components for this thesis was carried out by technical staff of the Australia Telescope, both at the Marsfield Radiophysics Laboratory, and the Narrabri Compact Array Observatory. Similarly, assembly of production versions of printed circuit boards for this thesis was carried out by technicians at the Radiophysics Laboratory.

Overall top level design of the interfacing system, including the decision to use optical fibre to connect the millimetre receivers back to the antenna control computers, was done in conjunction with Graham Moorey, head, Australia Telescope Receiver Group.

Dataset protocol 'C' libraries used within the Australia Telescope were incorporated in the test software written for this project. The author of these libraries is unknown.

LaTeX 2ε code written by Dr Shaun Amy of the Australia Telescope National Facility was utilised as a template for this thesis.

This thesis contains no material which has been presented for another degree at this or any other university and, to the best of my knowledge and belief, contains no copy or paraphrase of work published by another person, except where duly acknowledged in the text.

Suzanne A. Jackson
February 2003

Acknowledgements

I feel a debt of gratitude towards a large number of people, both for assistance over the course of this project, and also over the course of my studies. First and foremost must come my supervisors, Graham Moorey and A/Prof Sam Reisenfeld, for their unfailing support over the last eight months.

My colleagues within the Receiver Group also bear special mention, both as a rich source of ideas and inspiration, and also for helping in every step of the way in taking these ideas and creating practical hardware from them. In particular, I must thank Henry Kanoniuk, George Graves, Mark Bowen, Eliane Hakvoort, Les Reilly, Alex Dunning, and Jennifer Lie.

Numerous people in the ATNF workshops at Marsfield and Narrabri Observatory have also been of great assistance, machining metal and installing equipment where required, and on occasion showing great patience when not everything worked as expected.

Dr Dave McConnell, Dave Brodrick, Dr Mike Kesteven, and Simone Magri have been extremely helpful in debugging software aspects of this project, and have written large amounts of code to make these interface units work with AT online computing systems.

Graeme Carrâd analysed huge amounts of data from the water vapour radiometer acquisition system, whilst Dr Peter Hall, and Dr Robert Sault offered help in setting design goals for the system.

Dr Shaun Amy provided invaluable assistance in the typesetting and layout of this document, as well as kind and considerate advice, and lending an ear where warranted.

Finally I must thank my partner, Perry Armstrong, who has stood by me throughout the trials, tribulations, and occasional successes that have made up this degree. Without his unfailing emotional support, none of this would have been possible.

The presentation of this thesis was made possible through the use of LaTeX 2ε.

Abstract

The Australia Telescope National Facility runs a synthesis array near Narrabri comprising six 22m dishes. New receivers are being built for these antennas to cover 16-26GHz and 85-115GHz. As part of this upgrade, interface modules for the receivers must be designed and built, and these interfaces must be connected back to the antenna control computers.

These interface units will allow numerous analogue and digital variables to be interrogated via a high speed fibre optic interface. Close proximity to the receiving feedhorns dictate that particular attention must be paid to radio frequency emissions. The interfaces will make use of programmable gate arrays, for which firmware will be developed using schematic and VHDL.

This thesis covers the design and implementation of the interfacing hardware and software for these new receivers.

Contents

List of Tables xiii

List of Figures xv

1 Introduction 1
1.1 Overview 1
1.2 Australia Telescope Receiver Group 1
1.3 Receiver Interfacing History and Background 3
1.4 Precis of Work 4

2 Literature Review 5
2.1 Introduction 5
2.2 VHDL Literature 5
2.3 Compact Array Documentation 5
2.4 Xilinx Design Documentation 6
2.5 RFI Mitigation 6
2.6 Labwindows/CVI Coding 6

3 Dataset Engine 7
3.1 Introduction 7
3.2 AT Dataset Protocol 8
3.2.1 Request Format 9
3.2.2 Response Format 10
3.2.3 Packet Padding and Response Latencies 12
3.2.4 Function Address Partitioning 13
3.2.5 Parallel Buss 14
3.3 VHDL Coding 14
3.3.1 UART Design 15
3.3.2 Protocol Engine Design 16
3.3.3 Simulation and Testing 18
3.4 Schematic Instantiation 19
3.5 Summary 20

4 Receiver Interfaces **21**
4.1 General Interfacing Requirements 21
4.2 F83 Interface 22
4.2.1 RFI Mitigation 22
4.2.2 PCB Design 24
4.2.3 FPGA Design 26
4.3 Conversion Interface 27
4.3.1 PCB Design 27
4.4 Local Oscillator Interface 28
4.4.1 PCB Design and Packaging Issues 29
4.5 Water Vapour Radiometer 29
4.5.1 PCB Design 30
4.5.2 FPGA Design 32
4.6 Testing 33
4.7 Summary 37

5 Test Software **39**
5.1 Overall Requirements 39
5.2 Dataset Libraries 40
5.3 Dataset Test Code 40
5.4 Receiver Monitor Panel 41
5.5 Water Vapour Radiometer Code 42
5.5.1 Server 42
5.5.2 Client 43
5.6 Summary 45

6 Systems Integration and Project Management **47**
6.1 Overview 47
6.2 Fibre Cabling 47
6.2.1 Fibre Mux 48
6.2.2 Fibre Modems 49
6.3 Project Management Issues 49
6.4 Summary 49

7 Conclusions and Future Work **51**
7.1 Conclusions 51
7.2 Future Work 51

Bibliography **53**

A AT Dataset Engine VHDL Source **55**
A.1 UART Source 55
A.2 Dataset Source 60
A.3 F83 ADC Sequencer Source 71

A.4 F83 Bus Controller Source 74
A.5 WVR ADC Sequencer Source 77

B F83 Interface Schematics 83

C WVR Interface Schematics 97

D Conversion Interface Schematics 105

E LO Interface Schematics 111

F Fibre Mux Schematics 115

G Labwindows/CVI Source Code 117
G.1 Dataset Test Panel 117
G.1.1 dataset serv.c 117
G.2 Receiver Monitor Panel 127
G.2.1 mm rx.c 127
G.3 Water Vapour Radiometer Server 133
G.3.1 wvr.c 133
G.4 Water Vapour Radiometer Client 146
G.4.1 wvr client.c 146

List of Tables

3.1 Dataset request packet composition 10
3.2 Dataset request special characters 10
3.3 Dataset monitor reply packet composition (no errors or warnings) . . . 11
3.4 Dataset command reply packet composition (no errors or warnings) . . 11
3.5 Dataset error register bit allocations 11
3.6 Dataset response special characters 12
3.7 Control function address partitioning (D3) 13
3.8 Monitor function address partitioning (D3) 13
3.9 Dataset serial format 15
3.10 Common baud rate dividers 16

4.1 HFE4074 fibre transmitter specifications summary 26
4.2 HFD3023 fibre receiver specifications summary 26
4.3 Compact Array receiver interface fibre link budget 26

List of Figures

1.1 ATNF Compact Array antennas, in compact configuration with solid panels 2
1.2 Compact Array millimetre receiver prototypes 2

3.1 AT dataset buss structure 7
3.2 A "D2" dataset module 8
3.3 VHDL dataset parallel buss monitor timing 14
3.4 VHDL dataset parallel buss control timing 15
3.5 VHDL dataset simulation run 19
3.6 Dataset schematic symbol 19

4.1 F83 interface card 22
4.2 F83 ADC schematic 24
4.3 Conversion interface card 28
4.4 Local oscillator interface card 29
4.5 WVR interface card 30
4.6 WVR data acquisition schematic 31
4.7 WVR ADC reference histogram 32
4.8 F83 interface test board 34
4.9 F83 interface RFI test setup 35
4.10 F83 interface RFI results 36

5.1 Dataset test panel screendump 40
5.2 Receiver monitor panel 41
5.3 Water vapour radiometer server panel 43
5.4 Water vapour radiometer client panel 44

6.1 Fibre mux board 48

B.1 F83 interface project sheet 83
B.2 F83 interface ADC schematic 84
B.3 F83 interface digital I/O 1 schematic 85
B.4 F83 interface digital I/O 2 schematic 86
B.5 F83 interface RFI filtering schematic 87
B.6 F83 interface Xilinx support schematic 88
B.7 F83 interface power supply schematic 89

B.8 F83 interface Xilinx project sheet 90
B.9 F83 interface Xilinx comms schematic 91
B.10 F83 interface Xilinx address decode schematic 92
B.11 F83 interface Xilinx address buss schematic 93
B.12 F83 interface Xilinx local ports schematic 94
B.13 F83 interface Xilinx ADC sequencing schematic 95
B.14 F83 interface Xilinx transceiver direction schematic 96

C.1 WVR interface project sheet 97
C.2 WVR interface ADC schematic 98
C.3 WVR interface RFI filtering schematic 99
C.4 WVR interface digital I/O schematic 100
C.5 WVR interface DAC schematic 101
C.6 WVR interface Xilinx support schematic 102
C.7 WVR interface power supply schematic 103
C.8 WVR interface Xilinx schematic 104

D.1 MM conversion interface project sheet 105
D.2 MM conversion interface ADC schematic 106
D.3 MM conversion interface connector schematic 107
D.4 MM conversion interface digital I/O schematic 108
D.5 MM conversion interface Xilinx schematic 109
D.6 MM conversion interface power supply schematic 110

E.1 MM local oscillator interface schematic 112
E.2 MM local oscillator interface Xilinx schematic 113

F.1 MM fibre mux schematic 116

Chapter 1

Introduction

The engineer's first problem in any design situation is to discover what the problem really is.
Unknown

1.1 Overview

The Australia Telescope National Facility (ATNF) is a division of the Commonwealth Scientific and Industrial Research Organisation (CSIRO) charged generally with the administration and development of radio astronomy in Australia, and more specifically with the operation of telescopes for use by Australian and international researchers. The ATNF operates three installations; the 64m Parkes radio telescope, the 22m Mopra radio telescope, and an array of six 22m radio telescopes at Narrabri, termed the Australia Telescope Compact Array (ATCA).

As a result of the Working Nation report the ATNF was awarded an $11m grant in 1995 in order to upgrade the Narrabri Compact Array telescope to operate at millimetre wavelengths (DIST 1995). In addition to upgrades to the antenna surfaces and provision of extra stations to support short baseline interferometry, a substantial component of this upgrade is the design and manufacture of six new millimetre wave receivers, covering frequency bands of 16-26GHz and 85-115GHz (Moorey et al. 2002).

In order to facilitate control and monitoring of these receivers from the antenna control computers, a new programmable logic based “dataset” was developed. This design serves as the core enabling technology for an interfacing subsystem to allow computer control of the receivers and associated electronics via optical fibre.

This thesis covers the design and development of the dataset based interfacing hardware used to control these receivers, as well as software written to test the hardware and issues encountered in the implementation of the system.

1.2 Australia Telescope Receiver Group

The Australia Telescope (AT) receiver group has its origins in the birth of the science of radio astronomy, immediately following World War 2. In the past fifty years, the AT

Figure 1.1: An aerial photo showing five ATNF Compact Array antennas, in a compact millimetre-wave configuration, with new solid panels in place. The North/South spur can be seen extending from the main line towards the left of the photograph.

Figure 1.2: Two Compact Array millimetre receiver prototypes, undergoing cool down tests in the Narrabri receiver workshop. Much of the ancillary equipment, such as the down conversion modules and Water Vapour Radiometer, are missing in this picture.

receiver group (originally the radiophysics receiver group) has designed and constructed numerous receivers for telescope installations in Australia and overseas.

Some of the more noteworthy examples of late are the Galileo receiver; an ultra low noise, narrow band deep space communications receiver designed specifically to support the NASA Galileo mission, following the near disastrous mishap which prevented the spacecrafts' main high gain dish from unfurling. Utilising a synthesis array composed of the 64m dish at Parkes, and the 70m dish at Tidbinbilla, NASA scientists were able to download some 70% of the expected Galileo mission data.

The Parkes telescope has also been given a new lease of life, thanks to the commissioning of a 13 beam 21cm Hydrogen "multibeam" receiver, comprising a massive dewar supporting 26 independent low noise amplifiers and associated electronics.

The receiver group consists of some sixteen engineers and technicians, covering the fields of cryogenics, RF and MMIC design and assembly, and digital electronics design.

1.3 Receiver Interfacing History and Background

The development of the Australia Telescope Compact Array, in the 1980s, posed numerous control and monitoring challenges for AT engineers, principally in terms of how to control a large, highly distributed electromechanical system with the utmost in reliability, at low cost, whilst producing a minimum of unwanted Radio Frequency Interference (RFI) that might be picked up by the sensitive receivers.

Prior to the development of the Compact Array, AT receiver systems had no remote monitoring. Telescopes typically used only a few simple receivers, consisting of a basic RF signal chain and cryogenics. Should something go wrong, the only way to tell was that the signal disappeared. Configuration changes, such as changing frequency, were done manually.

The "AT dataset" with its accompanying protocol, was developed in the late 1980s to fulfil this need. The protocol is a simple asynchronous serial command/response string, whereby registers within different equipment may be interrogated or changed by providing an address and data string.

Previous AT receivers connected to a dataset via a single ended parallel buss. The dataset then connected back to the Antenna Control Computer (ACC) via a serial line. In order to mitigate the effects of RFI from the datasets, they were physically located separately from the receiver feedhorns.

As part of the MNRF receiver development, a new interfacing scheme is also being developed for use with the new receivers. This interfacing scheme must maintain a high level of software compatibility with previous incarnations, whilst improving on the original dataset design by being smaller, cheaper, faster, and most importantly quieter (in terms of electromagnetic interference).

1.4 Precis of Work

This capstone project details a subset of the development work for the MNRF receiver interfacing. This is a continuing project, starting in approximately 1999, and is expected to continue in some form well past the deadline for thesis submission. As such, this thesis can only hope to capture a portion of this development and implementation effort.

Chapter 2 details the process of literature review that was undertaken as part of this project.

Chapter 3 goes into the development of the "dataset engine", a simple VHDL model describing the operation of the AT dataset protocol, so that the protocol may be implemented in programmable logic.

Chapter 4 covers the design of most of the physical modules that make up the receiver interfacing subsystem, and details some of the design challenges that were overcome in order to realise the project. Of particular interest are the RFI challenges posed by these units, as well as the ADC subsystem of the water vapour radiometer interface.

Chapter 5 discusses some aspects of the various test programs that were used in the commissioning of the prototype receivers.

Chapter 6 details aspects of the systems integration effort, required to patch all these systems together and update the Compact Array antennas to use fibre optic communications. This chapter also covers the project management issues encountered over the course of the project.

In Chapter 7, the project as a whole is reflected upon, detailing what has been achieved, and what is still to be done.

The appendices contain documentation which is of interest as reference material, such as system schematics and code.

Chapter 2

Literature Review

Study the past if you would define the future.
Confucius

2.1 Introduction

As this thesis is part of an ongoing project, the process is not easily broken up into literature review, design, implementation etc. Instead, each subsystem of the project has been treated in this manner, so the literature review, like other aspects of the development process, is a necessary adjunct to other work, and is revisited where necessary.

In order to make some sense of the chaos, this literature review has been broken down into subsections, which whilst they do not reflect a chronological progression, certainly do represent a logical ordering of ideas for the project.

2.2 VHDL Literature

The seed for this project was germinated whilst studying Advanced Digital Systems, in which the Very high speed logic Hardware Description Language (VHDL) was a major topic. VHDL is a key enabling technology for this project. The text for this subject (Yalamanchili 1998) was initially useful, but proved to be focused more on simulatable rather than synthesisable VHDL code. My primary reference whilst designing the VHDL subsections used in this design was Synopsis (1998). This reference proved an invaluable resource. In addition, I came across a useful compendium of VHDL code in Smith (1996), which provided numerous examples which were of use in this project.

2.3 Compact Array Documentation

The Australia Telescope Compact Array is a large, complex instrument. Much useful information about the online computer systems and monitoring systems is available in The Australia Telescope Compact Array Users' Guide (1999). A succinct coverage of

the monitoring schema was found in Hall et al. (1992). Descriptions of the AT Dataset protocol were obtained from Ferris (1991). With such a large instrument, much remains unwritten. A lot of useful information and ideas for dataset use were thus gleaned from discussions with Dick Ferris, who designed the original datasets, and Simone Magri, who has written Unix device drivers for them.

Details of the previous receiver interface were found in Sinclair et al. (1992) and Reilly (1997). Further information was provided by Les Reilly, Henry Kanoniuk, Graham Moorey, and George Graves, in the form of design documentation (schematics etc) and advice.

2.4 Xilinx Design Documentation

Xilinx publishes a series of application notes on their website dealing with a myriad of design issues with Xilinx devices, including configuration schemes and design tips for the different families of FPGA. The Xilinx libraries guide (Xilinx 1993) was an invaluable published resource, as was the Xilinx databook (Xilinx 2002). Rather than use Xilinx configuration PROMs, I chose to use Atmel In System Programmable (ISP) parts. The Atmel (Atmel 2002) application note was of great use here. In addition were the notes on the Atmel in system programmer (Atmel 2002) which I built rather than buy.

2.5 RFI Mitigation

Good references on RFI mitigation and design practices, with practical advice and a minimum of "sacred cows" proved hard to find. One useful book was White and Mardiguian (1985), which contains a lot of good practical advice on coupling, grounding, and shielding, with plenty of graphs providing good "rule of thumb" values for different situations. This was an area where the experience of the senior engineers within the receiver group proved invaluable.

2.6 Labwindows/CVI Coding

No work involving the C language is complete without a reference to the seminal work on the subject, the "white bible" (Kernighan and Ritchie 1988). Whilst deceptively thin, this volume serves as a definitive reference for C syntax and semantics. Stevens (1994) provides a similarly useful introduction to TCP/IP, whilst Wright and Stevens (1995) gives lots of details of code and algorithms for connecting computers via TCP. The Labwindows Libraries Guide (National Instruments 1996) and Labwindows getting started guide (National Instruments 1996) were also useful compiler references.

Chapter 3

Dataset Engine

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

Douglas Adams

3.1 Introduction

In order to control and monitor low bandwidth systems on the Compact Array, equipment, such as receivers, conversion modules, power supplies etc. are connected to an "AT Dataset" (Hall et al. 1992). The dataset partitions different control and monitor points into registers, which are addressed using a request over an (typically RS485 twisted pair) asynchronous serial buss, as shown in Figure 3.1 (Ferris 1991).

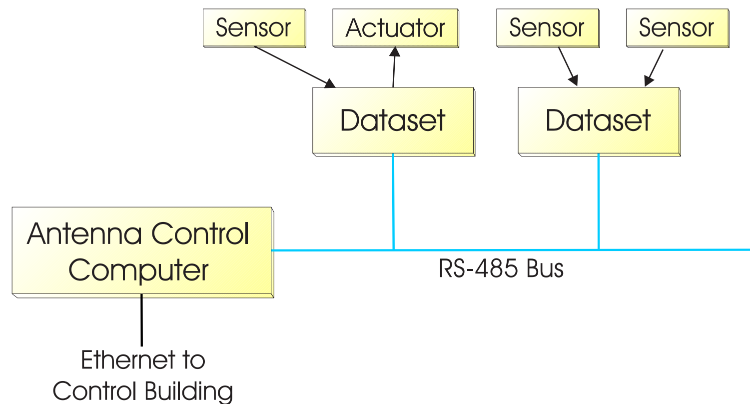

Figure 3.1: A diagram showing a typical dataset buss, with equipment connected via the datasets to a serial buss, and thence to the antenna control computer. In many implementations, a second module, called an interface, connects to a parallel buss on each dataset in order to add extra control and monitor points to standard dataset.

Until now, these devices have been comprised of a double width AT module, approximately 480 x 220 x 70mm, containing an 8031 microcontroller and associated

electronics, including a 12 bit Analogue to Digital Converter (ADC). The D3 version also included a multiplexer for the ADC. These devices are limited to a reasonably modest number of control and monitor points, and are also limited to 38.4kbps.

Figure 3.2: A photo of a "D2" dataset module. These units have provided sterling service to the Compact Array since their design in the 1980s. Parts for these modules are becoming increasingly scarce as they reach the end of their design life.

3.2 AT Dataset Protocol

In order to replace the standard dataset hardware, it is necessary to replicate at least a subset of the dataset protocol; the syntax and semantics for formulating dataset requests and responses.

A common thread in discussions about the dataset protocol revolves around the suitability of the protocol for controlling CA antenna equipment. Before blindly duplicating the protocol, it is perhaps useful to analyse the requirements of the connection between equipment and ACC, lest a different scheme; for example Ethernet, is more suited to the task.

The information transferred from the equipment is generally of a telemetry and diagnostic nature (Hall et al. 1992), for example:

1. Dewar temperatures, vacuums, and helium pressures,
2. LNA bias voltages,
3. System configuration commands, such as attenuator and switch settings,
4. Conversion chain power levels, used for setting gain.

Across an entire antenna, there are of the order of 5,000 different control and monitor points. Many of these are entirely static, so need only be accessed infrequently. A subset (perhaps a few percent) are more important and are usually accessed every

conversion cycle (approximately every ten seconds). A small number of points relate to rapidly varying values (for example Water Vapour Radiometer levels) and are read tens of times per second.

The aggregate data rate is of the order of 1000 points per second. Given a 16 bit data word (two bytes) and protocol overhead of circa 50%, this results in around 30kbps data rate.

So we may thus summarise the requirements for telemetry control and monitoring on Compact Array antennas:

1. Highly spatially dispersed equipment, typically with moderate to low complexity.
2. Moderate data rates, of tens of kbps.
3. High data integrity.
4. Extremely low RFI.
5. Low cost.
6. Serviceability.

The RS485 buss is a useful standard on which to base the design, as it is simple and uses inexpensive twisted pair cable. Numerous robust industrial transceivers and modems are available which may drive an RS485 buss, at the modest data rates required by the application.

Alternatives such as coax or twisted pair Ethernet prove unsuitable for the task, as they tend towards higher data rates (with concomitant higher levels of RFI) and Ethernet equipment tends to be more delicate and complex, being designed for data center use rather than industrial environments. However it should be mentioned that Ethernet is a clear choice in terms of implementation cost, being ubiquitous in the computer network field.

3.2.1 Request Format

Details of the inner workings of the dataset protocol are not widely published. In order to clarify the issue somewhat, this document summarises some of the information in (Ferris 1991), as well as points learnt from discussions with Mr Ferris.

In order to communicate with a dataset, the buss master (typically the ACC) must compose a request packet, and transmit this packet over the buss.

The request is sent bytewise in an asynchronous serial fashion. Information carried in the request packet is shown in Table 3.1.

The Sync byte is used to signal the start of the dataset request packet. On reception of this byte (0x16) all datasets on the buss prepare to receive a dataset address byte.

The Command/Monitor bit signifies the direction of data transfer, with a ‘1’ signifying a command (data transfer from the ACC to the dataset).

The Test bit is not normally used, and should be set to ‘1’.

Code	Description	Num. Bits	Value
SYN	Sync Byte (designates start of request packet)	8	0x16
CMD	Command/Monitor Bit	1	0 - Monitor
			1 - Control
Spare	Spare Bit (not used)	1	1
DSA	Dataset Address	5	0..31
FN	Function Address	9	0..511
DATA	Transferred data (0x0000 for monitor)	16	0..65535

Table 3.1: Dataset request packet composition

The Dataset Address (DSA) is a five bit value denoting which dataset on the buss is being addressed. The length of this field sets an upper number of 32 datasets on any given buss.

The Function Address (FN) is a nine bit value denoting the register that is being addressed within the dataset. The length of this field sets an upper limit of 512 registers within each dataset.

The final two bytes of the dataset request packet contain either data in the case of a command, or else zeros in the case of a monitor request.

Special Characters

Request packets are sent as asynchronous serial data, with an eight bit byte, one stop bit, and odd parity.

The Sync pattern (0x16) is treated as a special case. This pattern normally indicates the start of a packet. It is possible that this bit pattern will appear in normal data or address fields.

To account for this special case, a system of escape codes is used to replace control characters in data and address bytes. in essence, an ESC character is sent, followed by a code describing the bit pattern of the substituted data. The details for the request packet are shown in Table 3.2.

Code	Replaced Byte
ASCII ‘0’ (0x30)	ESC (0x1b)
ASCII ‘1’ (0x31)	SYN (0x16)

Table 3.2: Dataset request special characters

3.2.2 Response Format

When a dataset is validly addressed, in that it receives a valid Sync, and then a DSA matching its own, it replies immediately with a response packet.

Depending on whether the request is a command or monitor request, the dataset will respond differently. For a monitor request, the dataset replies with the contents of the addressed register, as shown in Table 3.3.

Code	Description	Num. Bits
ACK	Acknowledge Byte (designates start of response packet)	8
DATA	Transferred data	16

Table 3.3: Dataset monitor reply packet composition (no errors or warnings)

The dataset also replies to a command request, as verification that the dataset has received the data. In this case, the packet is of the form shown in Table 3.4.

Code	Description	Num. Bits
ACK	Acknowledge Byte (designates start of response packet)	8
ERR	Contents of dataset error register	8
WARN	Contents of dataset warning register	8

Table 3.4: Dataset command reply packet composition (no errors or warnings)

Error Response

In the case where a packet is received with an error, the dataset may reply with the ACK code substituted by an NAK. In this case the response is as for a command, and the contents of the error register are sent along with the NAK.

The VHDL implementation makes use of only a small subset of possible error conditions, as many sources of errors, such as a watchdog timer timing out on the microcontroller version of the dataset, are not meaningful for a hardware based device.

Table 3.5 shows all possible errors returned by the hardware dataset implementation:

Error bit	Description
0	Not used at present (always returned as '0')
1	Parity or Framing error in function address or data
2	Sync byte received when data expected
3	Invalid escape sequence received in function address or data
4	Not used at present (always returned as '0')
5	Not used at present (always returned as '0')
6	Not used at present (always returned as '0')
7	Not used at present (always returned as '0')

Table 3.5: Dataset error register bit allocations

Note that the dataset only replies with an error if it is correctly addressed. Should the dataset receive a malformed Sync or DSA byte, it will simply flash an indicator on its front panel (if it has a front panel) and wait for a valid Sync.

Thus the most common response to a problem, be it baud rate mismatch, data inversion, or other problem, is that the dataset does not reply to a request. In this case, the ACC must implement some form of time-out, so that it can report problems.

Warning Response

The original dataset implementations had a number of conditions, such as a low back-up battery, that would result in the dataset sending back a warning. In this case the dataset functions as normal, except it substitutes ACK bytes in each response with BEL bytes. Also the warning register (returned after each command request) will be set to some value depending on the nature of the problem.

The VHDL implementation, being a simple hardware based device, has no conceivable state where it could respond with a warning. Hence the dataset will never issue a BEL, and the warning register will always be returned as ‘0x00’.

Special Characters

As for the request packet, several byte patterns are special, and are used as control characters. Table 3.6 shows each of these. Note that although warnings are never issued by the dataset, the BEL character is replaced by an escape sequence to maintain overall protocol compatibility with earlier units.

Code	Replaced Byte
ASCII ‘0’ (0x30)	ESC (0x1b)
ASCII ‘2’ (0x32)	ACK (0x06)
ASCII ‘3’ (0x33)	BEL (0x07)
ASCII ‘4’ (0x34)	NAK (0x15)

Table 3.6: Dataset response special characters

3.2.3 Packet Padding and Response Latencies

Due to the possibility of escape sequences in the request and reply packets, these packets are conceivably of variable length. If this were allowed, the possibility exists of a minimum length request packet being shorter than a maximum length response, leading to buss congestion and mangled response packets on heavily loaded busses.

In order to avoid this problem, request packets are always padded with null (0x00) bytes, so that they have the same (maximum) length.

The maximum request packet length would occur when the function address byte is an escape sequence, as well as both data bytes. This means that the ACC transmits an ACK, a DSA, and then a further six bytes, for a total of eight. In cases where this doesn’t happen, the ACC must pad to at least eight bytes.

The response time of the dataset is also an important figure. If datasets wait for variable times before they respond, then the possibility exists for a slow responding

dataset to still be using the buss when a fast dataset starts responding to a subsequent request. To avoid this possibility, the dataset must respond as soon as it is validly addressed. On the VHDL implementation, the dataset sends an ACK (or NAK) sequence as soon as it recognises its own DSA.

3.2.4 Function Address Partitioning

Original datasets partition the control and monitor function address space (a total of 512 16 bit words) into a series of subsections, as indicated in Table 3.7 and Table 3.8 (Ferris 1997).

Function Type	Base	Range	Index
Undefined	0		
Single Bit Data	64	32	0..31
Addressed 8 Bit	96	64	0..31
Addressed 16 Bit	160	64	0..63
Decoded Addressed 8 Bit	224	4	0..3
Decoded Addresses 16 Bit	228	4	0..3
Reserved	232		
Undefined	236		
Word	256	4	0..3
Undefined	260		

Table 3.7: Control function address partitioning (D3)

Function Type	Base	Range	Index
Balanced Analog	0	8	0..7
Unbalanced Analog	8	56	0..55
Single Bit Data	64	32	0..31
Addressed 8 Bit	96	64	0..31
Addressed 16 Bit	160	64	0..63
Decoded Addressed 8 Bit	224	4	0..3
Decoded Addresses 16 Bit	228	4	0..3
Reserved	232		
Undefined	236		
Word	256	4	0..3
Undefined	260		
Multiplexed Analog	288	64	0..63
Undefined	352		

Table 3.8: Monitor function address partitioning (D3)

The millimetre receiver contains some 512 analog monitor points. Unfortunately this does not fit within the prescribed multiplexed analog portion of the dataset function address space. Rather than have the millimetre receiver occupy eight dataset

addresses (of a maximum of 32 on each buss) the decision was made to disregard the function address partitioning and treat all addresses within the function space as addressed 16 bit control and monitor points.

Thus, for all function addresses, the dataset block generates a 9 bit address, and either transmits 16 bits of data, or receives 16 bits of data on its data buss.

Should an application require the function addresses to be partitioned, this is easily accomplished by adding external logic to the dataset block.

3.2.5 Parallel Buss

The original dataset communicates with addressed devices via an eight bit buss. Sixteen bit transfers are made in two bytes, with a high/low signal to signify which half of the word is being controlled.

In order to simplify the buss, the VHDL dataset engine uses simple sixteen bit transfers, and dispenses with the high/low signal. Should eight bit transfers be required, an additional state machine is available that interfaces an eight bit data buss to the 16 bit VHDL dataset buss.

The timing relationships for the VHDL dataset are shown in Figure 3.3 and 3.4.

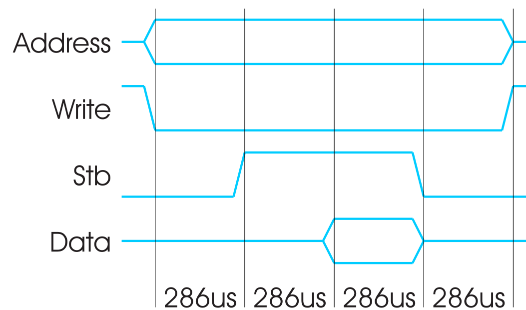

Figure 3.3: VHDL dataset parallel buss monitor read timing. Note that each of the time periods are dependant on the serial data rate. The example shown is for a 38.4kbps connection. Timings for other rates may be deduced by simply scaling the times shown.

3.3 VHDL Coding

The new dataset engine was developed using Very high speed Hardware Description Language (VHDL). This language allows the description of complex digital hardware using a high level of abstraction and a relatively straightforward text based language. The language supports decision constructs, such as if..then, as well as case decision trees, and allows complex synchronous state machines to be designed simply, and with a high level of reliability and guaranteed metastability.

In order to develop the dataset engine, the problem was split into two parts. Firstly, a simple Universal Asynchronous Receiver and Transmitter (UART) was designed, to

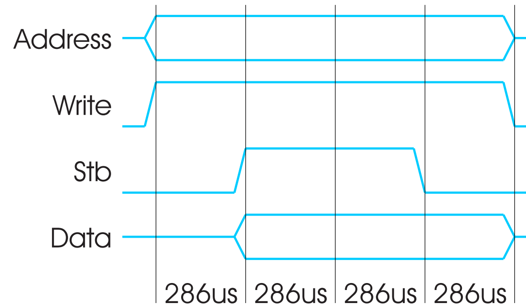

Figure 3.4: VHDL dataset parallel buss control write timing. Note that each of the time periods are dependant on the serial data rate. The example shown is for a 38.4kbps connection. Timings for other rates may be deduced by simply scaling the times shown.

implement the eight bit, parity asynchronous communications layer with a minimum clock division overhead.

Next, a layer was built on top containing the dataset protocol state machine, which decodes data from the UART according to the AT dataset protocol.

Finally, a series of VHDL "test harnesses" were written in order to exercise the dataset engine under different conditions, and to test its response under a wide range of conditions, such as buss congestion and corrupted packets.

3.3.1 UART Design

Current datasets operate with either 38,400 bps or 4800 bps asynchronous serial data, with the parameters shown in table 3.9.

Start Bits	1
Data Bits	8
Parity	Odd
Stop Bits	1

Table 3.9: Dataset serial format

The VHDL dataset supports these standard rates, and also allows for higher data rates, dependant on clock rate.

In order to minimise clock frequency, and thus keep RFI down, the UART design was carefully considered with a view to reducing the necessary number of clock cycles per bit.

Start Detection

To allow for errors in clock frequency between transmitter and receiver, it's important to start sampling data as close to the center of the start bit as possible. This is done by sampling the serial line eight times each bit. Whilst the receiver is waiting for a start

bit, a running buffer is kept of the last four samples. The rest state for the serial input is a '1', so a start condition is defined as "0b1000" in the running buffer. When this pattern is recognised, the receiver then samples data every eight clocks thereafter. The samples are thus taken near the center of each bit time. The requirement for sensing three subsequent zeros also provides a degree of noise immunity to the receiver, as a short noise spike is unlikely to start it.

Clock Divider

The dataset is designed to operate with clock frequencies of nominally 1.2288MHz to 4.9152MHz. In order to translate this frequency to eight times the desired baudrate, a simple programmable divider is supplied, which will prescale the input clock by any integer from 1 to 16.

A number of clock frequencies in this range divide neatly to standard baud rates, as shown in Table 3.10:

Prescale	1.2288MHz	1.8432MHz	3.6864MHz	4.9152MHz
0000	153.6Kbps	230.4Kbps	460.8Kbps	614.4Kbps
0001	76.8Kbps	115.2Kbps	230.4Kbps	307.2Kbps
0010	51.2Kbps	76.8Kbps	153.6Kbps	204.8Kbps
0011	38.4Kbps	57.6Kbps	115.2Kbps	153.6Kbps
0101	25.6Kbps	38.4Kbps	76.8Kbps	102.4Kbps
0111	19.2Kbps	28.8Kbps	57.6Kbps	76.8Kbps
1011	12.8Kbps	19.2Kbps	38.4Kbps	51.2Kbps
1111	9.6Kbps	14.4Kbps	28.8Kbps	38.4Kbps

Table 3.10: Common baud rate dividers

For other clock frequencies and prescale values, the baudrate may be calculated using Equation 3.1.

$$Baudrate = \frac{f_{osc}}{8(Prescale + 1)} \tag{3.1}$$

The standard clock frequency is 3.6864MHz. This is a commonly available oscillator, and allows the same dataset to operate at 38.4Kbps (for compatibility on busses shared with D1 etc datasets) and also to run at up to 460Kbps, to allow for higher transfer rates.

In order to minimise RFI, it is also possible to clock the dataset at just 307.2KHz, and have it operate on a 38.4Kbps buss.

3.3.2 Protocol Engine Design

In order to participate on a dataset buss, we need to be able to understand the protocol, from the point of view of the dataset.

Like with the UART, the protocol engine is effectively split into a receive state machine and a transmit state machine. The receive state machine communicates with the transmit machine using the tx_req variable, which triggers the transmit state machine to send either a monitor reply packet or an error reply packet.

The receive engine makes use of the following states:

1. Idle: In this state the engine is waiting for the reception of an ACK from the UART. it should ignore all other data.

2. Receive DSA: Reception of an ACK while idle invariably leads to the receive DSA state. In this state we wait for the next byte from the UART, which should contain the dataset address, and command/monitor bit. Whether we pay attention to the rest of the packet depends on whether the dataset address received matches our own. Also note that the most significant bit of the function address is transmitted with this byte. This must be saved for future reference.

3. Receive Function: After the DSA byte, the next byte transmitted by the ACC is the function address. This address is simply copied out onto the address lines.

4. Receive Function after ESC: Should the receive function state receive an ESC, this state is triggered. The receiver simply reads the next byte, and decodes it accordingly.

5. Receive High Data: After receiving the function address, the request packet contains the high data byte for a command, or else nulls for a receive.

6. Receive High Data after ESC: Should the receive high data state receive an ESC, this state is triggered. The receiver simply reads the next byte, and decodes it accordingly.

7. Receive Low Data: The packet finishes with the low data byte, or else nulls for a monitor request.

8. Receive Low Data after ESC: Should the receive low data state receive an ESC, this state is triggered. The receiver simply reads the next byte, and decodes it accordingly.

The protocol engine makes decisions in each state as to how to proceed with the transfer, and drives the transmit state machine accordingly. If a framing or parity error is received in the first two bytes, the engine ignores them and goes back to the idle state, waiting for the next SYN byte. Errors in subsequent bytes result in an error packet being requested, with the error flags set accordingly.

The transmitter state machine makes use of the following states in order to send the reply packet:

1. Idle: The transmitter monitors the tx_req variable. If tx_req is send_ack then the transmitter goes to the send_ack state. If tx_req is send_err then the transmitter goes to the send_err state.

2. Send ACK: In this state the transmitter loads the UART with the ACK code, and then waits for the UART to signal that it is free before going to the send_high state (for a monitor request) or the cmd_err state (for a command request).

3. Send High: This state latches the high byte on the data buss and sends it. If the byte to be sent is a reserved code, it instead sends an ESC sequence and goes to the send_high2 state, where the byte is sent, appropriately coded.

4. Send Low: This state latches the low byte on the data buss and sends it. If the byte to be sent is a reserved code, it instead sends an ESC sequence and goes to the send_low2 state, where the byte is sent, appropriately coded.

5. Send NAK: The Send NAK state is triggered when a request is received with an error in the function address or data bytes. The transmitter must send an NAK code, as well as the err and warn registers, which are loaded according to the error.

6. Unload: Each of the error or ACK paths terminate with this state, which simply waits for the UART to unload before returning to the idle state. This ensures that packets do not become corrupted when, for example, a valid packet is received in the middle of an error reply.

3.3.3 Simulation and Testing

In order to faultfind operation of the dataset engine, a series of test harnesses were written for it, again in VHDL. These harnesses included simple commands to sequence through operations, such as supplying a valid request packet to the engine, then deliberately malformed packets, etc. while also simulating the operation of external devices connected to the dataset.

When compiled and run, the simulation yielded a simple waveform trace, much like that found on a logic analyser, which could then be traced through in order to determine correct operation, or in the case of errors, used to track down the source of problems.

Once the dataset engine was tested with a range of simulated data, it was then programmed into an interface card, and tested using a PC, with a simple dataset test program written in Labwindows/CVI.

As implementations were developed using the engine, these too were tested, generally on test hardware, using a PC running the test software. Of interest here was the uncovering of a bug in the Labwindows serial communications routines, in that Labwindows (and perhaps Windows generally) ignores parity errors in received data, even when explicitly asked to report such errors via the serial drivers.

This problem was uncovered because the dataset UART transmitter neglected to initialise its parity variable before each byte, so the parity of each byte was essentially random. Months of testing under Windows showed no problems whatsoever, but when

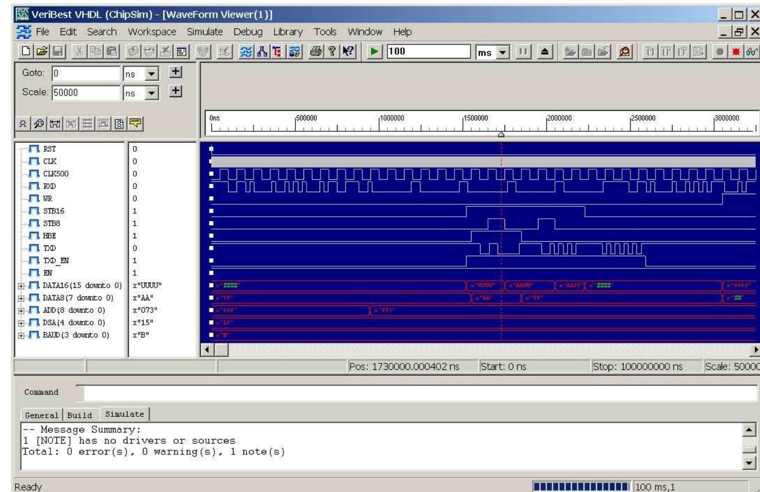

Figure 3.5: A screendump of the waveform output from a simulation of the VHDL dataset. In this case the dataset is being tested along with additional code to transfer eight bit items.

the interface cards were run under a Linux test system, parity errors abounded. Careful inspection of logic analyser output illuminated the problem.

3.4 Schematic Instantiation

In order to make use of the dataset engine within schematic based designs, it was necessary to build a Protel schematic component representing the dataset block.

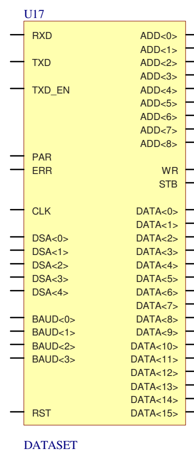

Figure 3.6: The dataset schematic symbol. Pins on this symbol are related to the VHDL dataset entity description.

The VHDL entity description provides information about how the component connects to the outside world, such as the naming and type of all signals. It is thus a

simple task to create a Protel schematic component with identical pins, and ensure that the Xilinx compiler maps the correct netlist file when reading the design.

3.5 Summary

The ATNF has no previous experience with VHDL models, either for use in simulation only or as synthesisable cores. The dataset engine synthesisable core, whilst useful in its own right as a control structure for embedded equipment, has also provided a thorough grounding in the use of VHDL both in a stand-alone sense, and combined with a schematic entry process, such as is typically used at the ATNF.

The same development process used for creating the dataset engine has since been applied to a number of other digital projects, such as ADC sequencers, cross-point switch units, and has more recently been proposed for use in digital correlator designs.

Chapter 4

Receiver Interfaces

> This is the rock-solid principle on which the whole of the Corporations [IBMs] Galaxy-wide success is founded...their fundamental design flaws are completely hidden by their superficial design flaws.
>
> **T.H. Nelson**

4.1 General Interfacing Requirements

The millimetre receiver package is actually three receivers sharing a common dewar. As such, a large number of control and monitor points are required.

In order to reduce the likelihood of interference between different receiver subsystems, and also to reduce the size of wiring bundles on the receiver package, the interfacing task is subdivided into a number of different parts, namely:

- Electronics cage, providing control and monitor of dewar parameters (vacuum, cryogenic temperatures, helium pressure, etc.), LNA bias voltages and currents, power supplies, and translator control and monitoring,
- Water Vapour Radiometer, providing monitoring of the four water vapour channel power levels, as well as total received power,
- Conversion system, providing monitoring of RF levels through the conversion chain, as well as configuration of various microwave switches and attenuators, to set the IF bandwidth and levels,
- Local oscillator system, providing monitoring of the 160MHz local oscillator reference, the optical reference, and the local oscillator power, as well as control the YIG oscillator control word.

These interfaces communicate with one another via an RS485 buss, carried on the electronics cage backplane, and via cables to the conversion enclosure and water vapour radiometer. The main electronics cage interface, designated the "F83", in addition to performing its own interface functions, also acts as a fibre modem, allowing all electronics on the receiver package to be controlled and monitored via a simple fibre pair.

4.2 F83 Interface

The F83 interface is used for control and monitor of power supplies, LNA bias supplies, and dewar cryogenic data, including cryogenic temperatures, helium supply and return pressures, and dewar vacuum, as well as providing control for the translator, which drives different feeds onto the telescope axis.

Previous receivers made use of an "F33 Dataset Interface" card, which contained a number of simple latches and buffers, allowing a D2 dataset, connected via a parallel address and data buss, to control various functions within the receiver electronics cage. A number of other cards are standard on AT receivers to do such things as multiplex bias monitor points and condition the vacuum sensor signals, etc.

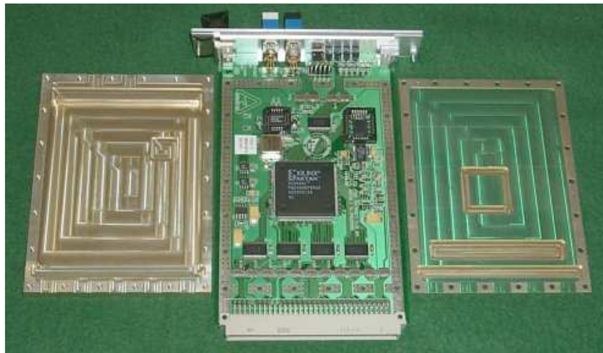

Figure 4.1: A picture showing the F83 interface card, along with its RFI covers. The fibre interfaces are toward the top of the picture, whilst the DIN91216 connector is toward the bottom. In operation, the RFI covers are bolted either side of the PCB, forming a sandwich.

In order to allow retrofitting of older receiver designs, the F83 is designed to have a modicum of backward compatibility with the previous F33 + D2 dataset combination that drives most AT receivers.

4.2.1 RFI Mitigation

Radio astronomy receivers are carefully designed to maximise receiver sensitivity within their operating band. Whilst these receivers are coupled to large feedhorns, in order to couple energy efficiently from the antenna system, sidelobes mean that the receiver is susceptible to radiated fields within the vertex room. In order to minimise pickup of Radio Frequency Interference (RFI) from electronic equipment, it is customary to isolate all clocked components, such as datasets, samplers, and the like on a separate floor, with shielding in between. Further, digital signals controlling the receivers are usually static during the active part of a conversion cycle.

The F83 card introduces a 3.6864MHz oscillator into the same room as the microwave feeds. This oscillator is necessary for recovering data from the dataset asynchronous serial stream. It is imperative to ensure that RFI from this oscillator is at a low enough level that it does not interfere with observing.

The worst case scenario for RFI on the Compact Array involves the L-band receiver. This receiver operates down to 1.2GHz. The system temperature of the receiver system (at 1.5GHz, and neglecting contributions from the cosmic background, the atmosphere, and the antenna structure) is approximately 21K (Sinclair et al. 1992). Equation 4.1 (Christiansen and Högbom 1969) provides a means to convert this value to a detection threshold:

$$(\Delta T) = QM\frac{T_{sys}}{\sqrt{\Delta v t}} \tag{4.1}$$

Substituting an arbitrary value of 1 for Q (information factor) and 1 for M (ideal receiver), and using a typical channel bandwidth (Δv) of 64KHz (64MHz IF, 1000 channels) over a 12 hour integration, gives a sensitivity threshold of some 3.2 $\times 10^{-4} K$.

This is expressed as a power using the following relationship:

$$P(dBm) = 10log(1000kBT) \tag{4.2}$$

Where k is Boltzmans constant (1.38×10^{-23}), B is bandwidth (64KHz) and T is temperature. This comes out to some -185dBm. Such a figure should be taken with a grain of salt, as it does not take into account the complex losses between the interference source and the feed (instead assuming perfect coupling), and also neglects the correlation isolation for such signals afforded by the phased array. However it still provides some feel for the signal levels which may be detected by the receiver.

In order to keep radiated signals from the interface equipment to these low levels, a number of strategies were employed to reduce emissions originating from the oscillator. These were:

- Using as low a clock rate as practicable. This is facilitated by the VHDL UART design, which has a $\times 8$ clock, rather than the more usual $\times 16$ clock of microcontrollers.
- Ensuring clock trace lengths were kept to a bare minimum, to minimise radiation.
- Using lossy ferrite "T" filters on clock and oscillator power.
- Pouring solid ground planes on all board layers, to minimise RFI coupling between traces.
- Manufacturing a bolt on RFI shield for the PCB.
- Passing all I/O and power traces through lossy ferrite filters at the point where they cross the RFI shield.

4.2.2 PCB Design

The Printed Circuit Boards (PCBs) for the F83 interface were designed using Protel Schematic and Protel PCB. These packages communicate with one another via netlists. This software is part of the standard design suite used at the AT.

In order to reduce overall cost, a two layer design was used, with 7 thou (0.18mm) minimum track width and separation, and 20 thou (0.5mm) minimum hole diameter. These design rules allow the use of inexpensive local manufacturing services, and do not generally require the use of bare board testing.

In order to make the most use of the reconfigurability of the Xilinx FPGA, the schematic was modified to make the PCB layout as clean as possible.

The PCB conforms to DIN "Eurocard" standards. All I/O is either via a 128 way DIN91216 connector to the receiver backplane, or via the fibre interface on the front of the card. The PCB is shown in Figure 4.1

Analog to Digital Conversion

The Analog to Digital Converter (ADC) posed some interesting design challenges, inasmuch as the original F33 card, which the F83 was supposed to provide backwards compatibility with, had only a +5V supply, yet afforded an analog signal range of ±5V.

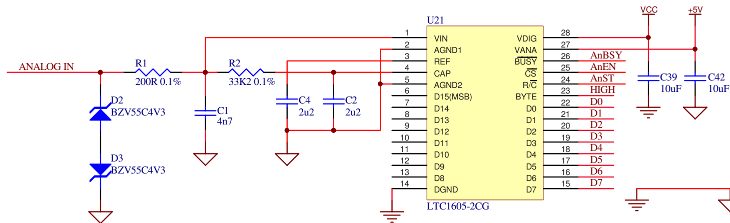

Figure 4.2: The F83 ADC schematic. The ADC chip is able to digitise a ±4.096V range, despite being powered by +5V.

A design using a DC-DC converter was considered, but rejected due to concerns that the supply would generate excessive RFI. Instead, a 16 bit single supply ADC (Linear Technologies 1999) was utilised, which uses a novel switched capacitor input stage to enable it to handle a ±4.096V input.

In order to ensure that the ADC can keep up with the dataset engine, without introducing delays that may cause conflict between buss devices, it's necessary to look at the time available in the protocol for returning data.

On reception of a valid request Dataset and Function Address, the dataset engine then replies immediately with an ACK byte, and then the upper and lower eight bits of the monitor point. The upper eight bits of the ADC result must be valid at the end of the ACK byte, ten bit times after the function address is valid.

At 38.4Kbps (the upper speed used by the AT standard datasets) this corresponds to 260μs. The ADC used is self clocking, and performs a conversion in just 10μsec

(100Ksps). This implies an upper dataset buss limit of approximately 1Mbps, which is ample for current and future requirements.

Careful attention to detail was observed with the ADC board layout, to ensure that digital signals would not degrade the noise performance of the ADC. The ADC power supply was bypassed with a 100μH inductor, and a series of 10μF ceramic capacitors. These ceramic capacitors were chosen for their extremely low ESR and small size when compared with similar value tantalum or electrolytic capacitors.

Digital I/O

The F83 card provides some 108 individual TTL level digital I/O lines. Each line is individually driven by a Xilinx pin, for maximum flexibility in design.

The digital lines from the interface travel on a receiver backplane, with a distance between transmitter and receiver of as much as a metre. Whilst high speed operation is not important, low RFI contribution is a serious concern, as is current drive for expected loads of up to ten TTL gates. A high level of noise immunity is also desirable.

Digital I/O from the interface is buffered using 74ACTHQ series buss transceivers. These devices are well suited to the application, as they have slew rate limited drivers, and 25Ω series resistors to minimise ground bounce and RFI. In addition, they make use of a “keeper circuit” which holds receiver inputs at their current level with a weak current, to improve the noise immunity. In this way, it’s possible to build the card without pullups on the digital inputs, which would otherwise be difficult and tedious to fit on the board.

Xilinx Configuration

The program for Xilinx FPGAs is stored off-chip in a programmable memory device, which must be programmed before the interface is used.

In order to simplify the development process, an Atmel EEPROM device was used, which is simply reprogrammed in circuit using a short cable from between the programming board and a header on the interface card. The prototype interface located the programming header outside the RFI shield, but the RFI filters played havoc with configuration signals, so it was decided to relocate the header under the shield, necessitating the removal of the shield to reprogram the card.

Fibre and Twisted Pair I/O

The F83 interface is the boundary point between the external (fibre optic) dataset buss, and the internal (twisted pair) dataset buss.

In order to facilitate the F83s role as a fibre modem, the interface card has both RS485 twisted pair drivers and optical fibre transceivers.

Signals for each are derived from the FPGA, for increased flexibility in use. This means that, for example, F83 cards may be programmed as slaves on the twisted pair buss, rather than driving it. This functionality was used in the Taiwanese AMiBA (AMiBA 2002) prototype receiver, where extra I/O was desired.

The twisted pair interfaces make use of Maxim MAX487 RS485 transceivers. These devices use low input current receivers (enabling up to 128 devices on a buss) and slew rate limited transmitters, which operate with reduced RFI at up to 250Kbps.

The fibre interfaces are based on a Diamond E2000 duplex fibre pair, which utilise Honeywell HFE4047 fibre emitters and HFD3023 schmidt trigger PIN detectors.

The detector specifications are summarised in Table 4.2, whilst those of the transmitter are in Table 4.1. Link budget calculations for the link are shown in Table 4.3, using the transmitted power, receiver sensitivity, and nominal values for loss of various interconnections.

Coupled Power	-17dBm (into 50/125μm fibre
Drive Current	50mA
Peak Wavelength	850nm
Bandwidth	100MHz

Table 4.1: HFE4074 fibre transmitter specifications summary

Sensitivity	-27dBm (from 50/125μm fibre
Min. Dynamic Range	15dB
Peak Wavelength	850nm
Max. Data Rate	5Mbps

Table 4.2: HFD3023 fibre receiver specifications summary

Transmit power	-17dBm
ACC to pedestal bulkhead fibre (0.01km @ 2dB/km)	0.02dB
Pedestal bulkhead connector	0.30dB
Pedestal fibre splice	0.10dB
Fibre through az. and el. wraps (0.2km @ 2dB/km)	0.40dB
Sampler rack fibre splice	0.10dB
Sampler rack bulkhead connector	0.30dB
Sampler rack to receiver fibre (0.01km @ 2dB/km)	0.02
Power at receiver	-18.24dBm
Noise margin	8.76dB

Table 4.3: Compact Array receiver interface fibre link budget

4.2.3 FPGA Design

The F83 interface card is mainly a programmable logic design. In order to communicate with the antenna control computer, it includes a dataset engine, which controls other logic within the FPGA via a simple parallel buss.

Addresses from the dataset engine are decoded, and a variety of registers are accessed to control receiver systems.

One of the primary functions of the F83 is as a monitoring point for LNA bias and dewar environmental parameters, such as vacuum, temperature, etc. These variables are read in analog form, and are multiplexed via a series of bias mux and analog mux cards in the receiver electronics cage.

In order to perform a conversion when needed, the F83 contains a simple ADC sequencer, which sets up the analog multiplexer addressed, and then initiates an ADC conversion after a settling period. The sequencer then monitors the ADC ready/busy line, waiting for the conversion to finish, and finally transfers the data in two bytes from the ADC to the dataset parallel buss.

This logic, being based on a simple state machine, is written in VHDL. A similar model was also written to read the ADC, and then scale and offset the result to a 12 bit number, as per the D2 specification. This design was used along with a simplified buss driver to "emulate" a D2 dataset + F33 interface combination, as used in the multibeam receiver, as well as other previous AT receivers. This allowed RFI problems caused by the proximity of the multibeam to its D2 dataset to be alleviated.

4.3 Conversion Interface

The conversion interface is used to control switches and attenuators within the down-conversion system (Graves et al. 2002), and to monitor signal and local oscillator levels within the conversion system.

This interface was originally intended to also control the local oscillator module, but in order to minimise connections to the local oscillator module, the local oscillator instead has its own dedicated interface, which is detailed later.

Due to fears of local oscillator leakage from the conversion system, the entire system is sealed up within an RFI shield, and is controlled with just two twisted pairs, plus an event pair.

As with the F83 interface, the conversion interface controls switches and attenuators etc. via a number of TTL level digital lines. Unlike the F83, all analog multiplexer functions for monitoring signal levels must be done by the interface.

4.3.1 PCB Design

Many of the same design constraints that applied to the F83 interface are also pertinent with the conversion interface. Some elements of the design, for example RFI mitigation, were somewhat relaxed however, as the entire conversion module is enclosed in an RFI shield, with filtered feedthroughs used for all signal and power connectons.

Space constraints on the conversion module led to the use of a four layer board, with 7 thou design rules for the interface. Much of the PCB real estate is occupied by connectors.

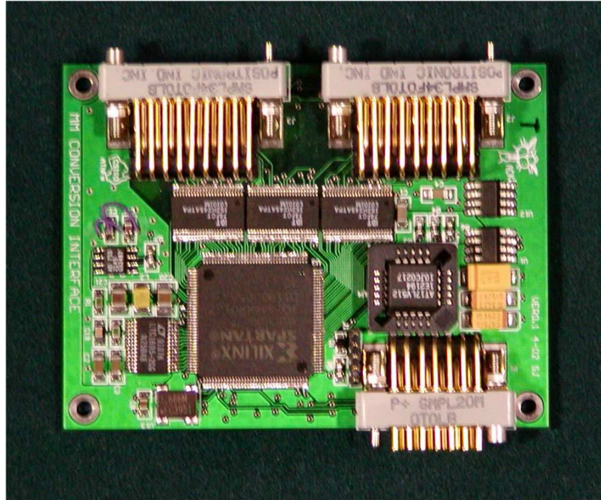

Figure 4.3: A picture showing the millimetre conversion interface card. The PCB size is dictated largely by the connectors used to get signals and power on and off the board. Note that this card does not have the strict RFI requirements of the F83, and thus does away with the RFI covers and filters.

Whilst there was no need to add RFI shielding to the interface, attention was still paid to minimising possibility for interfering fields being coupled off the board. As with the F83 interface, this centered around the oscillator, and its connection to the FPGA. This net was kept short, and solid ground planes were provided around and under the oscillator and FPGA so that fields would preferentially couple to the planes.

Analog to Digital Conversion

The detector signal conditioning circuits used for the conversion system use differential signals. In order to accommodate these signals, the analog multiplexer was doubled up, and a differential to single ended converter was used before the ADC. This improves the noise immunity of the sampler, because coupled common mode noise is rejected by the differential amplifier. Otherwise, the ADC setup is the same as that used on the F83 interface.

4.4 Local Oscillator Interface

The local oscillator interface is responsible for control and monitoring of the L86 local oscillator module. This module was originally intended to be controlled by the conversion interface, but concerns over possible intermediate local oscillator frequencies leaking from the enclosure led to the design of a separate local oscillator interface, in order to minimise wiring to this module.

The local oscillator module is comprised of a number of parts, including a fibre receiver, with a power detector, and a PLL driven YIG oscillator, which is locked to

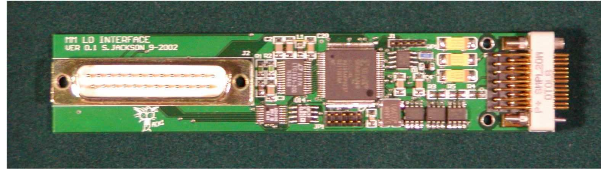

Figure 4.4: A picture showing the local oscillator interface card. This card fits alongside the local oscillator phase locked loop module, hence the odd shape. As with the conversion interface, RFI is not a great concern, so the card has no covers and no RFI filtering.

the fibre reference. The YIG frequency depends on a 12 bit digital control word, which is supplied by the interface. In addition, the YIG coil current is monitored, so that a simple under/over comparison may be made to determine if the oscillator is likely to lose lock.

4.4.1 PCB Design and Packaging Issues

The design of the local oscillator interface follows closely that of the F83 and conversion interfaces. RFI shielding is not as important for this module, as it is enclosed within the L86 module, which is heavily shielded in order to keep local oscillator signals from escaping. However, the same design techniques as used on the F83 board, such as locating the clock chip close to the FPGA, and minimising clock trace length, have also been employed on this board.

The primary challenge in the design of the local oscillator interface was one of packaging it so that it would add a minimum of size to the L86 module. A four layer board was designed to accommodate the Xilinx chip, config PROM, ADC, some muxes, and power supply electronics in a scant 15cm^2. Indeed the majority of space on the PCB is occupied by two connectors, via which it communicates with the L86 and the outside world.

4.5 Water Vapour Radiometer

The Water Vapour Radiometer (WVR) (Abbott and Hall 1999) is an experimental system intended to increase the upper frequency limit of the Compact Array under less than ideal seeing conditions.

The placement of the Compact Array, in Northwestern NSW, whilst being close to optimum for centimetre wave observing, is too low in altitude and high in humidity for good year round millimetre observing.

Water vapour has a higher dielectric constant than dry air, having the effect of delaying microwave and millimetre wave signals (as well as attenuating them except in certain bands). In areas of relatively high humidity, the density of water vapour in a given column of atmosphere varies markedly from place to place. The result with a synthesis telescope is decorrelation, and subsequent loss of signal, at higher frequencies.

By accurately measuring the density of water vapour in the beam for each antenna, it is hoped that the phase shift on the astronomical signal introduced by the water vapour can be cancelled out by the correlator, thus improving the system tolerance to atmospheric water vapour.

The method used in the millimetre receiver to measure water vapour consists of a simple offset feed, with a low noise amplifier, followed by four simple TRF receivers, centered on different bands around 26GHz.

The signal level in these bands approximate the density of water vapour within the received beam. The WVR interface thus digitizes the output of each of the detectors, along with a number of other parameters that may affect the system, such as LNA temperature and feed temperature.

4.5.1 PCB Design

The PCB design methodology adopted for this interface was as for the F83. As with the F83, great care was exercised to ensure that RFI, predominantly from the oscillator, was not carried outside the enclosure. To these ends, all signals in and out of the interface are filtered with RFI suppression filters, and a tight fitting RFI shield was designed and fabricated.

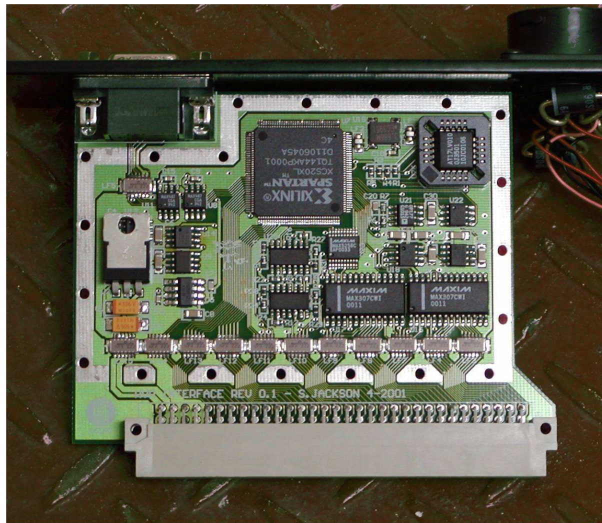

Figure 4.5: The interface card for the Water Vapour Radiometer.

The WVR interface connects to the receiver via a simple 9 pin D connector. Power (±20V) and all other signals are carried on a 2 row DIN connector.

The WVR assembly was originally designed with the intention that the analog signals (±10V differential) would be carried on a cable to the main receiver backplane. However, tests showed that the 16 bit ADC on the receiver electronics package wasn't up to the task. In order to avoid a costly redesign of the WVR package, the interface board was designed to fit within the enclosure, above the WVR power supply board.

Analogue to Digital Conversion

Initial system tests using a National Instruments 16 bit data acquisition card indicated that the quantization noise in this system was a major contributor to total system noise. Thus, in order to ensure that quantization errors did not adversely effect performance, the digitizer was specified with a minimum SNR of 90dB, or 15 bits.

This specification, when teamed up with the requirement to sample all channels in under 100 milliseconds and the need for high levels of long term system stability, proved a daunting challenge.

A survey of available ADCs revealed that most 16 bit chips only guaranteed typically 14.5 effective bits SNR. In order to guarantee 15 bits SNR, a 24 bit delta-sigma ADC chip was chosen as the core of the design. This device, an ADS1252 (Texas Instruments 2000), guarantees 18.6 bits effective resolution over temperature, at a sample rate of some 40Ksps.

It was important when designing the ADC signal conditioning circuitry (shown in Figure 4.6) that the SNR specifications not be compromised by noisy input switches and op-amps. Similarly, variations with temperature were to be minimised, in order to reduce the effect of the periodic air-conditioner cycle on the system.

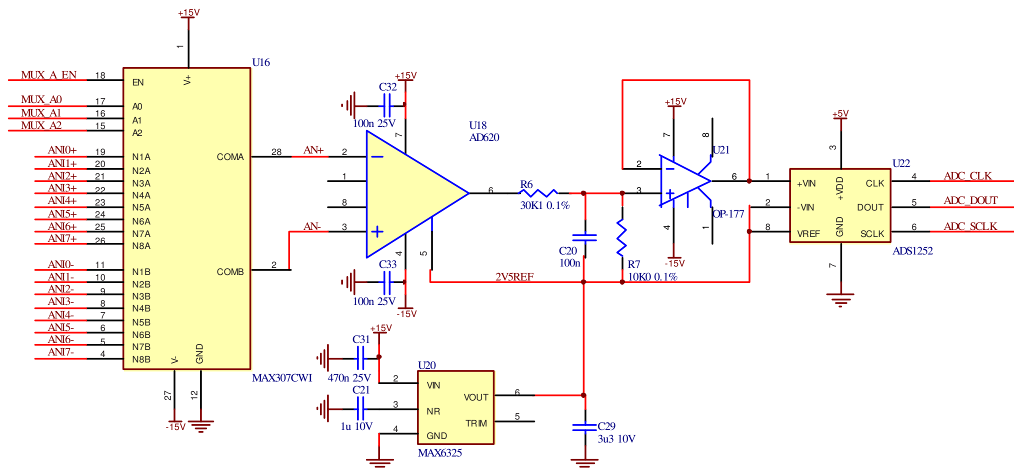

Figure 4.6: A schematic diagram showing the signal conditioning circuitry, ADC, and associated reference for the WVR. Note that this figure is slightly inaccurate, as it omits the second multiplexer, but is adequate to illustrate the design principles used.

The analog inputs are thus multiplexed with a pair of MAX307 differential 8 to 1 multiplexers. These use low on-resistance switches, of just 60Ω, minimising additional noise contribution by series resistors in the input stage.

The selected signal then passes through an AD620 differential amplifier, to convert to a single ended signal centred around +2.5V. The AD620 has a low input noise of $70\text{nv}/\sqrt{\text{Hz}}$ (Gain = 1), as well as an input bias current of 1nA. When combined

with the MAX307 input resistance, the noise contribution is approx 3μV pk-pk (0.1 to 10Hz). This equates to around 22 bits, or 8 LSB.

The signal is next divided by four by a simple voltage divider, in order to scale it to the correct ADC range, and buffered by an OP177 precision low noise op-amp, before being input to the ADC. The OP177 maintains the low noise levels of the differential amp.

The voltage reference is based around a MAX6325 precision reference, with 3ppm accuracy over temperature. The MAX6325 offers access to the reference element, so that noise may be reduced by the use of a large value capacitor (in this case a 1μF ceramic).

Of course, it is necessary to follow strict layout guidelines in order to preserve the specs of these components, paying particular attention to the separation of analogue and digital ground planes on the board, and ensuring that no traces cross the planes to couple digital noise into the analogue subsystem.

The result, as shown in Figure 4.7 is an impressive quantisation SNR of 110dB, or 18.5 effective bits.

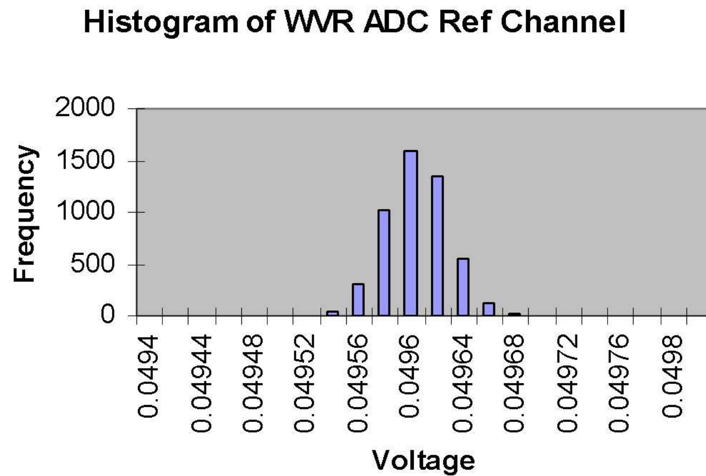

Figure 4.7: A histogram showing the distribution of samples from the 24 bit WVR ADC, when recording data from a precision 50mV reference. Each bin is 80 μV or 67 ADC counts. The standard deviation of this data is 24 μV (110dB SNR)

4.5.2 FPGA Design

The main additional component required in this interface was a sequencer for the analog to digital converter. A serious issue in the design of the sequencer is that the ADC is far too slow to perform a conversion during a normal dataset request-response interval. In addition, the 24 bit data returned by the ADC will not fit into a 16 bit dataset register. Further, the ADC has no “conversion start” command.

The solution to these problems lay in continually reading all analog inputs one after the other, and storing the results in temporary registers, which may be read as desired.

This method has the drawback of introducing a measure of indeterminacy in the timing of the sample, as well as consuming a considerable amount of Xilinx resources for the registers necessary.

The timing jitter turns out not to be a serious concern, as it is masked to some extent by software timing jitter in any case, and by significant filtering on the sampled data, both before the interface, and in software post-processing.

The sequencer is designed in VHDL, and is a simple state machine, composed of a counter which drives the multiplexer to select one of the eight inputs, and a clock generator for the ADC, which clocks the converter at 1/6th of the crystal frequency (614.4KHz).

For each input, eight successive conversions are performed. All but the last are discarded. This is necessary because the ADC contains digital FIR filters, which are six conversions long. When a step input is applied, as with switching from one voltage to another, six conversions are necessary before the ADC settles down.

The result is that the sequencer updates all eight registers once every 20msec. This rate was not chosen by accident, but rather gives a high degree of 50Hz mains frequency rejection to the system.

The final part of the design is a simple shift register to reconstruct the serial output of the ADC into a parallel form.

When reading a 24 bit register, a scheme was developed to allow 24 bit reads while ensuring that the high and low order data was always from the same conversion. The high order 16 bits are read first by a dataset monitor request. Concurrently with the read, the low eight bits from the relevant register are placed in a temporary register, which may then be read at a different address. In this way, the converter may be treated as an accurate 16 bit converter in one read, or as a 24 bit converter in two.

4.6 Testing

In order to simplify testing of these various interfaces, they were tested in a number of different ways at various points in their development to ensure that they worked as expected.

As a rule, the FPGA design paralleled that of the physical interface boards. Elements of the FPGA designs, such as the dataset engines and various ADC sequencers, were simulated stand alone using a VHDL synthesis package, before any hardware was built.

Whilst designing the PCBs, design checks were made both during schematic development, and during board design. In the case of the schematic design rule check, the test flagged such items as:

- Multiple outputs driving a common net.
- Unconnected inputs, or nets with no source.

- Unconnected power nets.
- Duplicated designators etc.

Whilst these tests are by no means exhaustive, they give some idea that the schematic is a true representation of the design.

Next, as a final stage in PCB design, a second design rule check was run on the PCB design. This check tested the PCB against the loaded netlist, and flagged errors such as:

- Footprint mismatches between PCB and schematic.
- Unrouted or partially routed nets.
- Clearance violations and shorts between nets.
- Track widths not within design rules.

Of course the creation of a reasonable PCB layout depends heavily on generating workable, sensible design rules for features, such as signal and power net track widths, clearance constraints, via size, etc.

Finally, once the prototype boards were constructed, they were powered up and programmed, and then test software was loaded into the FPGAs to exercise their various functions.

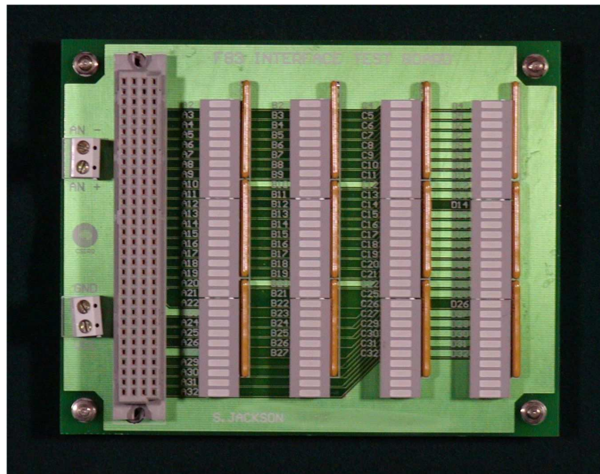

Figure 4.8: The F83 interface test board, which proved an invaluable tool for chasing PCB soldering problems amongst the thousand of soldered connections on each F83 interface.

One area that proved troublesome was the RFI filters on the F83 cards, which were difficult to solder, resulting in a large number of dry joints or bridged pins and subsequent frustration. In order to effectively and quickly test the digital I/O lines on the F83 cards, a test board was built, which the F83 plugged into. This board

contained an LED for each digital I/O line. Test software was loaded into the F83 card which simply displayed a moving bar on the LEDs, making shorts and opens very obvious because of disruptions in the LED pattern.

As RFI emanations were identified in the design specifications as a major concern, the F83 card was also tested extensively to ensure that the RFI levels were within acceptable limits. The RFI performance was measured on a prototype F83 card, with RFI covers in place, and using a 1.2288MHz oscillator as a timing reference. The emitted field was sampled at a distance of 2cm from the backplane connector, using a broadband Rhode & Schwartz E-Field probe, and wideband LNA (100KHz-2GHz, 30dB gain, 3.5dB noise figure). The test setup (with an unshielded card) is shown in figure 4.9.

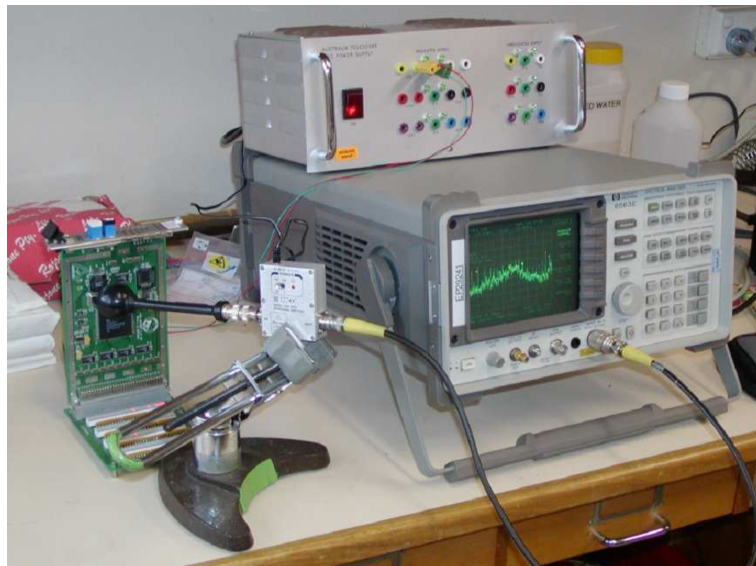

Figure 4.9: The F83 interface, being tested for RFI emissions, using an E field probe, LNA, and spectrum analyser.

The results of the RFI testing were extremely encouraging. Whilst it was possible to find evidence of the oscillator harmonics, they were down in the spectrum analyser noise, and required considerable work to display, with long sweep times. The results are shown in Figure 4.10 for three frequencies; 480MHz, 960MHz, and 1.44GHz. Emissions are around -125dBm below 1GHz, and drop to around -140dBm at 1.4GHz. Whilst this is considerably more than the minimum detectable signal at 1.4GHz, this is unlikely to cause problems unless the interface is placed within the primary beam of the telescope.

Of course, the final proof of operation of any device is in actual use. The final stage of testing was to install the interfaces on prototype receivers and then to test the receiver interfacing as a whole. To facilitate this effort, a number of programs were written, as detailed in Chapter 5.

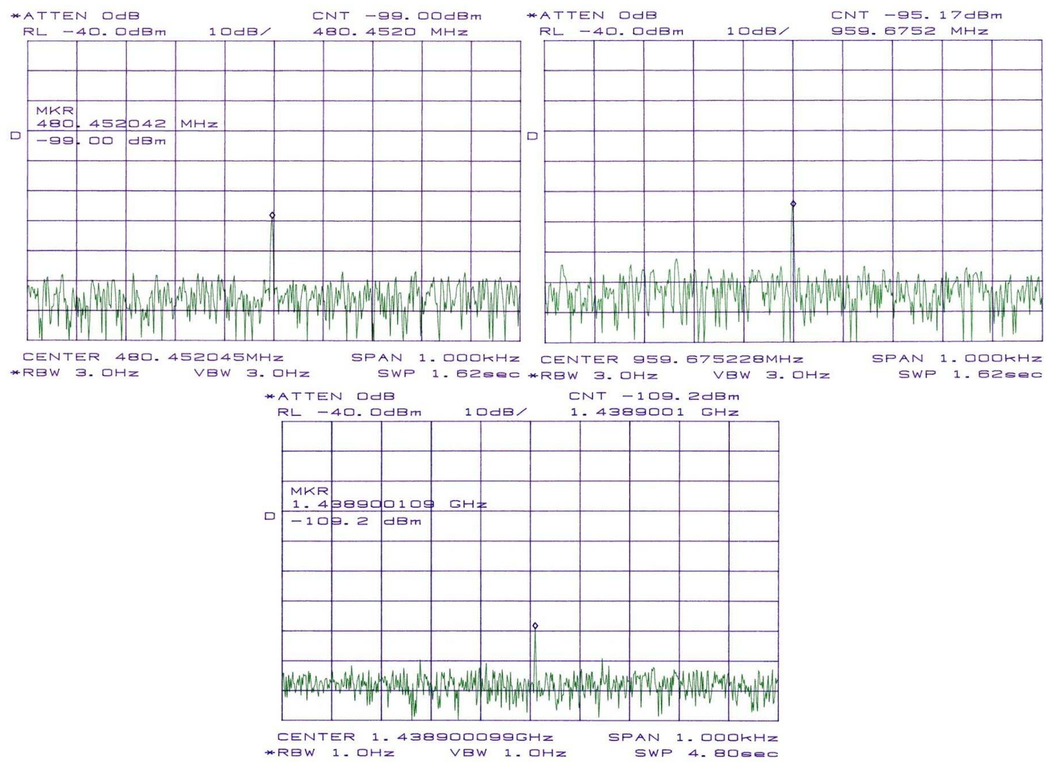

Figure 4.10: Test plots from the F83 interface, with a 1.2288MHz oscillator and RFI covers fitted.

4.7 Summary

This chapter describes the design and construction of a number of different implementations of the dataset engine in hardware form. Each interface unit makes use of programmable logic to house the engine, as well as other logic, and has peripherals attached to add additional functionality.

In designing the F83 interface, careful attention was paid to minimising radiated fields, so that it does not interfere with observing processes. It was also designed in such a way that it provided an upgrade to existing receivers.

The WVR interface data acquisition circuit proved challenging to implement, in order to minimise noise on the analog inputs. The result of this careful design was an acquisition system with some 18.5 bits of effective resolution, more than enough to ensure that quantisation noise was an insignificant contributor to overall WVR system noise.

The local oscillator and conversion interfaces show just how small the interface hardware can be made using multi-layer circuit boards and fine pitch SMT parts, and provide a glimpse of the future directions the interfacing effort will take in the AT; one of reducing the number of control wires in systems by locating small, cheap interfaces close to the sources of data.

Each of these implementations follow a similar design process, whereby the system is partitioned into manageable modules, and then the I/O requirements of each module are identified, and an interface designed to suit. Testing is done in each case first on a simulator, for VHDL components, then with specially constructed test jigs for prototype hardware, and finally on the actual receiver system.

Chapter 5

Test Software

> C is often described, with a mixture of fondness and disdain varying according to the speaker, as "a language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language".
>
> **MIT Jargon Dictionary**

5.1 Overall Requirements

The code developed as part of this project was done so more for diagnostic and testing processes than for final use. A large software suite already exists to control observations with the Compact Array, and in routine use, it is anticipated that this software will also encompass the hardware described in Chapter 4.

That being said, however, two important points led to the need to develop some semblance of control and monitoring software which is independent of the main online code. These are speed of development, and the need for an independent "arbiter" of system functionality.

This last point is brought about by the concurrent upgrade of antanna control computers, from LSI-11 based systems to industrial PC machines running the pSOS real time operating system. It was felt that some rudimentary independent driving software for the receivers would be helpful in diagnosing both receiver hardware and ACC software problems.

The language used for development of this code was National Instruments Labwindows/CVI (National Instruments 1996). This language was chosen for the following reasons:

1. We already owned a copy, and I was proficient in its use,

2. The languages' C base meant we could use pre-existing dataset drivers,

3. The language includes a range of useful instrumentation GUI "widgets" which would simplify development of control panels, etc.

4. Unlike its close cousin, Labview, Labwindows/CVI code is compiled, and runs at reasonable speed on modest hardware.

5.2 Dataset Libraries

As stated previously, existing dataset libraries were used as the basis for much of the test software. These libraries have been developed over the last fifteen years or so in order to allow control of equipment connected to the usual "D1", "D2" etc datasets. Whilst the VHDL "dataset engine" was designed as a software compatible replacement of the older datasets, due to differences in implementations, some thought was still needed in the use of the libraries.

The best example is the way original dataset functional address space was partitioned into ranges, for example "16 bit addressed" range etc. The libraries are based on this partitioning, and contain functions of the following form:

```
int Addressed_8_Bit_Out(int ds_address, int control_point,int data_out)
```

which allow access to the relevant subset of the functional address, and perform meaningful bit masking etc. on data transferred. While the possibility exists of building additional logic around the dataset engine to implement these ranges (as indeed was done in the multibeam dataset replacement), by and large the ranges are dispensed with.

This means that the most useful functions in the dataset libraries are the slightly lower level "`Initialise_Dataset`" (used to open the comm port), "`SendMessage`" (which formats a request and sends it on the selected comm port), "`ReadResponse`" (waits for a response to arrive and decodes it), as well as "`Close_All_Datasets`". Using these functions, the system may be thoroughly tested.

5.3 Dataset Test Code

The dataset test panel was the first program written for debugging the dataset connection, during development of the dataset engine. This code has been revisited numerous times, as different problems were encountered with the interface hardware, in order to make it display more meaningful information about the link.

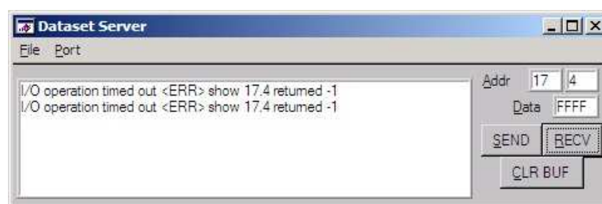

Figure 5.1: A screendump showing the dataset test panel. The last two attempts at communication were unsuccessful, as shown in the history display.

This code presents the user with a simple control panel (shown in Figure 5.1), in which dataset transactions may be made, and the results displayed. The user is able to set comm ports and baud rates with pull down menus. In order to debug remote TCP/IP connected systems, a version even exists whereby the server is run on the machine connected to the dataset buss, then a client is able to drive the server, and thus exercise the buss.

5.4 Receiver Monitor Panel

The receiver monitor panel is a large diagnostic utility, which displays many of the analog monitor points on a receiver, including such things as low noise amplifier bias conditions for every stage on every amplifier, dewar parameters, power supplies, etc. This is a large amount of information, so a selection must be made via a simple pull-down menu as to which band is being monitored at a given time. The user interface panel is shown in Figure 5.2.

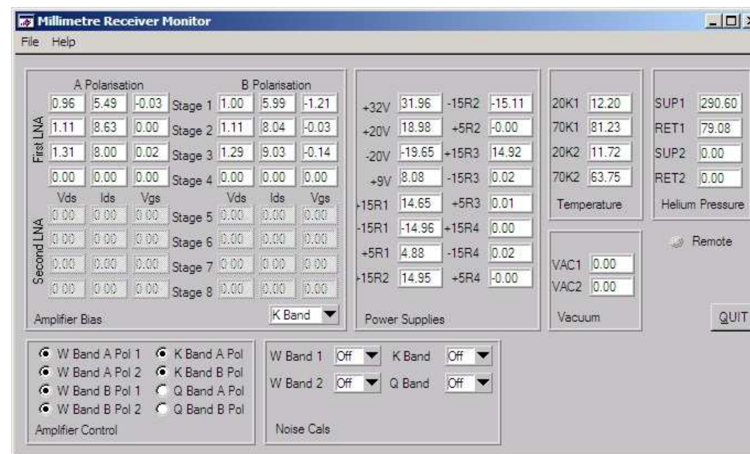

Figure 5.2: A screendump showing the millimetre receiver monitor panel. The panel is displaying actual telemetry, in this case from the receiver in CA02.

Considerable difficulty was encountered with this program in ensuring its responsiveness to the user interface. The program spends most of its time I/O blocked waiting for responses from dataset requests, and is able to respond to the user only occasionally. If a problem occurs, such as a non-responsive buss, the program would all but hang, waiting for buss time-outs.

A solution to this difficulty lay in running the acquisition process and the user interface in separate threads, which communicate via simple semaphores and a shared memory scheme. The acquisition process is thus able to run flat out, updating indicators as necessary, while a thread waits for user input.

This scheme of separate acquisition and user interface threads was so successful that it has been used in all subsequent data acquisition programs.

5.5 Water Vapour Radiometer Code

The water vapour radiometer data gathering program is a good example of the way small, simple code can grow in scope and size to incredible proportions. The program was originally written as a means of gathering data from the water vapour radiometer, via a National Instruments DAQ card, and dumping the data to disk, to facilitate stability tests in the lab. It was never intended to be used in the final acquisition system, but has run in various guises and on various hardware consistently ever since. One memorable test run involved cable-tying laptop PCs running windows to the frame for the L-band feed in the antenna vertex rooms, with a huge tangle of wires coming out of the PCMCIA slot on the PCs, and an ethernet switch similarly cable-tied nearby. Much was learnt in this test (over several months) about the deleterious effect of Ethernet hardware on L-band observations.

5.5.1 Server

The WVR server is the core of the program. It performs periodic reads of all analog channels in the water vapour radiometer, accurately time tags them, and stores them on a local disk file in .csv format, an ASCII format that is simple to import into an Excel spreadsheet. The general format is shown below:

```
Water Vapour Radiometer Logfile
15:05:33, 741.330000, 0.049686, ... -0.000944, -0.006514,
15:05:33, 741.430000, 0.049543, ... -0.000997, -0.006151,
15:05:33, 741.530000, 0.049562, ... -0.001054, -0.006247,
```

The first entry for each line gives the time for the sample in `HH:MM:SS` format. Next is the time from program start, in seconds, represented as a floating point number. This entry is accurate to within about 1 millisecond, dependant on the PC system clock, which may be locked up to an external reference using the ntp protocol. Subsequent entries are the voltage (in volts) for each of the sampled inputs.

It is perfectly possible to run instances of the server entirely stand-alone, without an ethernet link or client.

Most of the work involved in writing the server software was in ensuring that it would reliably take data under a wide range of conditions, including disk congestion (due perhaps to network access of previous or current data files), and dealing with the vagaries of running under Windows, where processes can be inexplicably suspended for hundreds of milliseconds for no good reason.

The server makes use of a dynamically allocated linked list queue, in order to buffer data, as shown below:

```
typedef struct List {
    double data[16];
    char time[9];
```

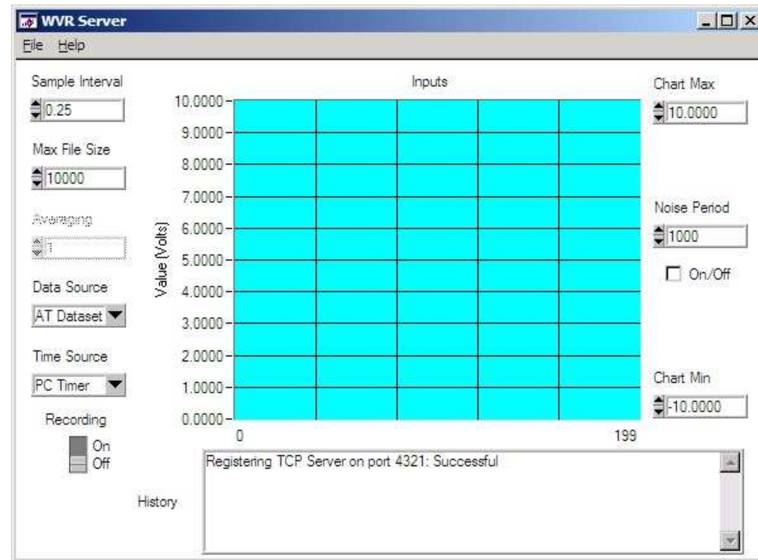

Figure 5.3: A screendump showing the water vapour radiometer server graphical user interface.

```
        double sec;
        struct List *next;
    } list;
```

The server is heavily threaded, in that one thread acquires data and places it in the queue, whilst another thread looks after the user interface. The queue can grow as large as necessary (up to the memory size of the computer). Every ten samples, the acquisition thread tries to open the data file and write out twenty samples (or else the contents of the buffer). In this way, the data file is left closed most of the time so other programs can access it while it is being written. The acquisition program will simply attempt to access the file each ten samples, with its FIFO buffer growing each time, until the file becomes available. At this point, the program writes out twenty records each ten samples, so the FIFO length drops back to zero fairly quickly without a sudden long data transfer disrupting the acquisition process.

In addition, a separate thread monitors the TCP/IP connection, and the user interface, and allows a user to start and stop the aquisition, change the sample interval, number of samples per record, etc.

5.5.2 Client

The water vapour radiometer client software acts as a simple remote control for the server. It includes all controls and displays present on the server panel, with the exception of the running chart recorder display, and allows all aspects of data acquisition to be controlled remotely from the central control room.

In order to minimise reliance on network connections, the actual data is dumped to a disk locally on the server machine, and not transferred to the client. Each file is

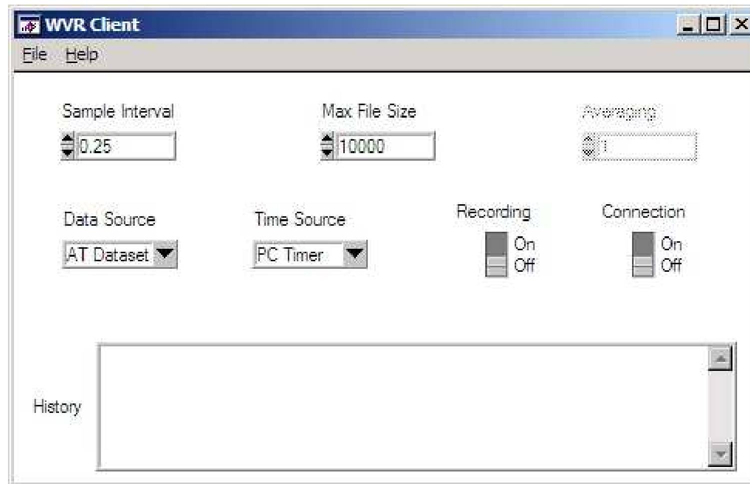

Figure 5.4: A screendump showing the water vapour radiometer client "remote control" panel.

limited to a number of samples, after which time the server closes the file and creates a new one, with a name based upon the current date and time. In this way, it is possible to transfer data from the server machine to the central site using normal operating system file transfer procedures, transparently to the acquisition process.

TCP packets to control the server acquisition are of the following form:

```
struct msg_type
    {
        char msgHD;
        unsigned char msglen;
        char msg[512];
    };
```

The header is a token to identify a command string. In this case it is simply '71' stored as a character. Next the message length is stored, and finally the actual data, as an ASCII string.

Control messages are of the form:

```
COMMAND DATA
```

For example, the following command sets the sample interval to 100msec:

```
INTERVAL 0.10
```

The server replies to requests with a similarly framed packet, echoing the request message and appending either `<OK>` or `<ERR>` depending on whether it was able to complete the request. For error conditions, it also attempts to provide some information as to what is wrong.

In addition, the server periodically updates the client as to the state of acquisition, echoing the number of samples taken and the file to which they have been written every hundred samples, providing some reassurance that the process is working.

5.6 Summary

A number of different programs have been written to debug and prove operation of different parts of the interfacing system. These programs were written in Labwindows/CVI, mainly due to familiarity with the language and the convenience of being able to run the diagnostics from a Windows notebook.

In the case of the water vapour radiometer, the test software served as the main acquisition system for a period of time, reliably gathering data from the radiometers.

With the exception of the basic dataset test panel, all other monitor programs make use of a threaded architecture to separate the acquisition and user interface portions of the program, ensuring that neither interferes adversely with the other.

Chapter 6

Systems Integration and Project Management

Planning is an unnatural process; it is much more fun to do something. The nicest thing about not planning is that failure comes as a complete surprise, rather than being preceded by a period of worry and depression.
Sir John Harvey-Jones

6.1 Overview

All these boards, modules, and software don't exist in a vacuum. They must be connected to one another in a robust, maintainable way, and integrated into the antennas.

Much of the top level design of this interfacing project was dictated by external factors, such as availability of the telescopes for engineering work, the presence of existing multimode fibres in the telescopes, etc.

This has proven to be the most difficult and time consuming part of the project, as it often involves difficult modifications to the telescopes, which must be done in a manner which does not affect observations.

6.2 Fibre Cabling

The millimetre receiver makes use of optical fibre cables already installed through the telescope structure, running from the pedestal room, at the base of the antenna, up through the azimuth and elevation cable wraps to the vertex room, where they terminate in the side of the sampler rack.

In order to make use of these fibres, they must be extended to the upper floor of the vertex room, and connected to the millimetre receiver package. To minimise duplication of effort, these fibres are run in the same duct work as single mode fibres carrying the local oscillator signal to and from the receiver.

The following tasks were necessary to make use of the current fibres:

- Splicing new connectors on to the ends of the existing fibres, which are compatible with the connectors used on the millimetre receiver.

- Installation of "Adaptaflex" flexible conduit between the sampler rack and top floor distribution panel.

- Design and installation of a box for the top floor, in which to break out the fibres for the millimetre receiver interface, millimetre receiver Local Oscillator, and C/X interface.

Splicing of fibres and installation of conduit was performed mainly by technicians on-site at Narrabri.

6.2.1 Fibre Mux

The fibre mux is a simple board that allows several fibres, plus a twisted pair buss, to communicate with the ACC via a single fibre pair.

This board allows the one fibre connection to control all three receivers (via fibre) plus the existing vertex room twisted pair dataset buss. This saves considerable expense by reducing the number of fibres, plus reducing the number of connections.

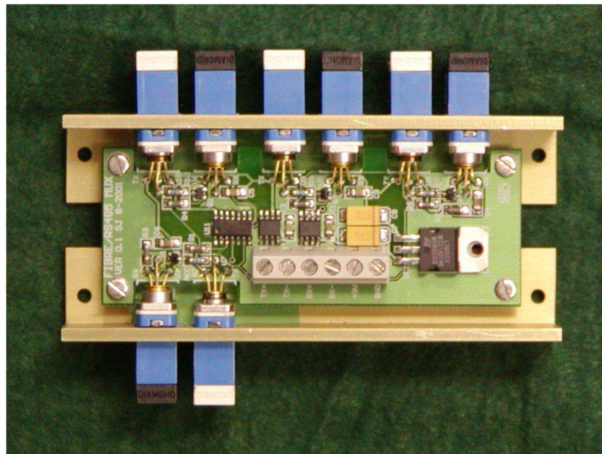

Figure 6.1: The fibre mux board. This board allows the use of three fibre connected devices, as well as a twisted pair RS-485 dataset buss on the one ACC fibre.

The design of this board was not especially difficult. All the board does is distribute and buffer the transmitted data from the ACC to all ports, and drive the receive data line back to the ACC with the logical OR of all inputs. In this way, requests from the ACC are transmitted to all datasets, and any reply is relayed back to the ACC. No attempt is made to prevent congestion, as the datasets should only reply when they are addressed.

6.2.2 Fibre Modems

The original proposal called for the development of a plug in fibre modem to occupy the ACC I/O chassis. Due to time constraints, this subproject has been delayed until after the millimetre receiver interfaces are otherwise commissioned.

In the meantime, the project makes use of Optical System Devices OSD-135 fibre modems, which serve well with the use of an adapter on both fibres, to the E-2000 standard used by the rest of the project.

6.3 Project Management Issues

This has been a long and at times difficult project. Challenges were overcome in many technical issues, such as how to achieve the required SNR in the water vapour radiometer interface, and also in tackling a large, complex project, involving many facets such as coordination of people, equipment, and time.

In order to make some sense of the project, the first step in the design process was to divide the project up into a number of “modules”, which could be more easily handled in isolation, rather than one large conglomerate mess. The specifications of each module were defined early on, along with the way the modules connected together.

Development of individual modules was done in a traditional waterfall method, whereby the specifications were thrashed out, then a then a prototype designed, constructed and debugged, and finally production units built for use on the array.

Perhaps the most difficult aspect of planning a project such as this is that when timescales are originally estimated, one has a tendency to treat the project at hand as though it is the only thing being done. In reality, other tasks and projects intrude, causing timescales to stretch beyond those originally predicted.

Another issue is that when working with a large instrument such as the Compact Array, one is not able to simply turn up to the telescope with some equipment and install it. Instead, development efforts must mesh with regular maintenance periods and installation shutdowns. Of course, these are the periods where staff availability at the telescopes is at its lowest, with everyone busy dealing with the deluge of problems that must be solved in the limited maintenance time.

The timelines developed in the proposal were adhered to in a general sense, if not in detail. The main divergence from the proposal was that the conversion interface front panel has been deferred to a later date, while the conversion interface itself was effectively split in two, with local oscillator control functions separated. This allowed the project to be completed within the allowable time, which was a hard limit in any case due to compact array scheduling.

6.4 Summary

Significant additional work, beyond the design and construction of circuit boards and the writing of software, are necessary before this equipment can be used in real life

on the Compact Array antennas. Much of this work is uninteresting and fiddly, and takes an inordinate amount of time to complete, especially when faced with significant obstacles in the allocation of resources to the task, and the synchronisation of work with other users of the facility.

The general development process followed in this project is one of top down design in the first instance, and then a simple waterfall process for sub-projects identified in the initial project outline. This process is a useful one for this type of equipment, in that it neatly divides up work early in the process, and limits risk to other sub-projects should difficulties be encountered with a design. The disadvantage with such a model is that the overall parameters of the design are effectively "cast in stone", and it is difficult to make large changes to the overall system.

Chapter 7

Conclusions and Future Work

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.
T.S. Eliot

7.1 Conclusions

It is no easy task to summarise work done over such a period of time, encompassing so many different disciplines. This is the first time I've been responsible for a project of this breadth and depth, and it has been a very valuable learning exercise.

Many of the techniques used in this project are new to my workplace, most especially the use of VHDL to design logic circuitry. Processes established for this project for the use of VHDL in designs have since been used in the development of other equipment.

Further, much of the equipment designed as part of this project have found uses in other applications. Most notably, the F83 interface has become a ubiquitous feature of new receiver designs, and has been used in other organisations besides the ATNF.

This project started life as a "what if" session late one night whilst commissioning interface equipment for the Parkes radiotelescope, and has since at times appeared almost to have a life of its own. Through applying project management principles, such as doing a work breakdown and timeline for the project fairly early on, much of the unpredictability of the project was tied down. Much was also learnt about the pitfalls associated with project management, such as the tendency for people to ignore external influences when estimating time for completion of a given task.

On the whole, this has been a wonderful learning experience. Being able to start with an idea and see it through to fruition as a significant project has been an exciting thing to do.

7.2 Future Work

As intimated in the introduction to this thesis, this project is part of ongoing work. As such, this may be seen as a "slice" of a much longer, larger project.

It is perhaps useful to bring out our crystal ball, and look into the future to see where the interfacing work within the ATNF will take us. The incorporation of the dataset engine within existing electronics has led us in a direction that shows great promise for future equipment.

As with many fields, much of the cost of equipment for the ATNF is taken up with interconnections. Using the original datasets necessitated running a huge number of connections across the backplane of a receiver or conversion system to control equipment. The provision of a simple, small, cheap dataset that can be hidden inside modules allows us to reduce the amount of digital and analog wiring to a bare minimum, with concomitant gains in system cost and reduction of noise pickup.

The close integration of this relatively high speed digital circuitry with sensitive analog and RF equipment means that a number of challenges must be met, most especially with reducing the RFI emanations from digital equipment, or with mitigating their effects.

One possible method of reducing the effect of RFI on analog and RF circuitry may be found in the use of spread spectrum clocks. The asynchronous serial comms used by the dataset is tolerant of significant mismatches between transmitter and receiver clocks. This allows us to dither the respective clocks, such that the energy in clock harmonics is spread over a wide frequency range, and is thus less of an issue.

Finally, whilst this project marks the end of my Bachelor of Engineering studies, it also marks the first significant project in my career. I hope to go on to further studies, and undertake many more projects as my career unfolds.

Bibliography

Abbott, D., and Hall, P., "A Stable Millimetre-Wave Water Vapour Radiometer", Dec 1999, Journal of Electrical and Electronics Engineering, Australia. Vol. 19, no. 4, pp. 213-225.

AMiBA Array for Microwave Background Anisotropy - Project Homepage, Academia Sinica Institute of Astronomy and Astrophysics, 2002 *http://www.asiaa.sinica.edu.tw/amiba/*

In System Programming Circuits for AT17 EEPROMS with Atmel and Xilinx FPGAs, *http://www.atmel.com/atmel/acrobat/doc3030.pdf*

Atmel FPGA Configurator Programming Kit Manual, *http://www.atmel.com/dyn/resources/prod_documents/DOC0642.PDF*

The Australia Telescope Compact Array Users' Guide, CSIRO ATNF, October 1999 *http://www.narrabri.atnf.csiro.au/observing/users_guide/html/atug.html*

Christiansen, W. & Högbom, J. "Radiotelescopes", 1969, Cambridge University Press, Cambridge. p233.

DIST Press Release. 1995, "Visionary Science Projects Keep Australia at the Leading Edge", *http://www.dist.gov.au/events/innovate/r1.html*

Ferris, R. "Introducing the AT Data-sets", 1991, Australia Telescope National Facility Internal Report.

Ferris, R. "AT Data Set Application Software Guide", 1997, Australia Telescope National Facility Internal Report.

Graves, G., Bowen, M., Jackson, S., and Sincliar, M., "The Conversion System for the Australia Telescope Millimetre-Wave Receiver System", 2002, Poster paper, Workshop on the Applications of Radio Science, Katoomba.

Hall, P., Kesteven, R., Beresford, R., Ferris, R. and Loone, D. "Monitoring and Protection Strategies for the Compact Array", 1992, Journal of Electrical and Electronics Engineering, Australia. IE Aust & IREE Aust. Vol 12, No. 2, pp 211-218.

Kernighan, B. and Ritchie, D. "The C Programming Language (2nd Ed.)", 1988, Prentice Hall, New Jersey.

Linear Technologies LTC1605-2CG 16 Bit ADC Data Sheet, *http://www.linear.com/pdf/160512.pdf*

Moorey, G., Gough, R., Graves, G., Leach, M., Sinclair, M., Bolton, R., Bowen, M., Kanoniuk, H., Reilly, L. and Jackson, S. "The Australia Telescope Millimetre Wave Receiver System", 2002, Proc. Workshop on the Applications of Radio Science.

"Labwindows/CVI Standard Libraries Reference Manual", 1996, National Instruments Corporation, Austin.

"Getting Started With Labwindows/CVI", 1996, National Instruments Corporation, Austin.

Reilly, L. "F33 Dataset Interface #1 Card", 1997, Australia Telescope National Facility Internal Report.

Sinclair, M., Graves, G., Gough, R., Moorey, G. "The Receiver System", 1992, Journal of Electrical and Electronics Engineering, Australia. IE Aust & IREE Aust. Vol 12, No. 2, pp 147-160.

Smith, D. "HDL chip design : a practical guide for designing, synthesizing, and simulating ASICs and FPGAs using VHDL or Verilog", 1996, Doone Publications, Madison.

Stevens, R. "TCP/IP Illustrated Volume 1: The Protocols", 1994, Addison-Wesley Publishing Co., Massachusetts.

Synopsis Inc. "VeriBest FPGA Synthesis VHDL Reference Manual", 1996, Synopsis.

Texas Instruments ADS1252 Data Sheet, *http://www-s.ti.com/sc/ds/ads1252.pdf*

White, D., and Mardiguian, M. "EMI Control, Methodology and Procedures", 1985, Interference Control Technologies, Illinois.

Wright, G., and Stevens, R. "TCP/IP Illustrated Volume 2: The Implementation", 1995, Addison-Wesley Publishing Co., Massachusetts.

Xilinx Inc. "Xilinx XACT Development System Libraries Guide", 1993, Xilinx.

Xilinx Inc. "The Programmable Gate Array Data Book", 2002, Xilinx.

Yalamanchili, S. "VHDL Starters Guide", 1998, Prentice Hall, New Jersey.

Appendix A

AT Dataset Engine VHDL Source

Include the source for the Dataset Engine.

A.1 UART Source

```
-- MNRF Interface VHDL Code

-- Asynchronous receiver/transmitter design

-- Revision: 1.3
-- Commenced: 23/8/99.
-- Last Modified: 5/6/01.
-- Author: Suzy Jackson sjackson@atnf.csiro.au

-- This package contains the bare bones hardware for an asynchronous
-- serial receiver (rx), and an asynchronous serial transmitter (tx)
-- which will hopefully be of some use to the MNRF generic interface.

-- A few notes about the design:

-- The current datasets use 38,400 bits per second (or 4800), 1 start
-- bit, 8 data bits, odd parity, and 1 stop bit.  I see no dramatic
-- need to rock the boat here, so lets start with something that does
-- the same...  Also it might be nice to allow for higher data rates.
-- With this in mind, the design should be OK for many standard
-- baudrates from 38400bps to 460.8Kbps using the same 3.6864MHz
-- clock.  Should baudrates outside this range be desired, the
-- crystal frequency can be changed, or alternatively an external
-- frequency divider could be used to drive the clk input.  The
-- Xilinx software gives a maximum clock rate of 25MHz for a
-- "typical" design, utilising a Spartan XCS40XL240-4, so with this
-- chip the baudrate may be pushed up to aroud 2.5Mbps.

-- Receiver Design:

-- At present, I've defined the clock to be 3.6864MHz.  This divides
-- by 96 to give 38400Hz.  We really need 8 clocks per bit, so I've
```

```
-- built in a nifty little predivide which can divide from 1 to 16,
-- giving us the following bitrates (given by bps =
-- fosc/(8*(prescale+1)))

--  Prescale:    1.2288MHz:    1.8432MHz:    3.6864MHz:    4.9152MHz:

--    0000       153.6Kbps     230.4Kbps     460.8Kbps     614.4Kbps
--    0001        76.8Kbps     115.2Kbps     230.4Kbps     307.2Kbps
--    0010        51.2Kbps      76.8Kbps     153.6Kbps     204.8Kbps
--    0011        38.4Kbps      57.6Kbps     115.2Kbps     153.6Kbps
--    0101        25.6Kbps      38.4Kbps      76.8Kbps     102.4Kbps
--    0111        19.2Kbps      28.8Kbps      57.6Kbps      76.8Kbps
--    1011        12.8Kbps      19.2Kbps      38.4Kbps      51.2Kbps
--    1111         9.6Kbps      14.4Kbps      28.8Kbps      38.4Kbps

-- This means that we really want to sample the data 4 prescaled
-- clocks after a start detection, and then every 8 prescaled clocks
-- to shift in the data, in order to guarantee that we'll be within
-- the bit even if the transmit and receive clocks vary.  If we
-- sample in exactly the middle of the first bit, then we can be out
-- by +/-5% (ie lots) before we get framing problems.

-- A start condition is defined by 0 on RXD for three divided clocks.

-- The p_err output is logic 1 for an error with odd parity.
-- Obviously, it'll be logic 0 for an error with even parity.  f_err
-- is the inverse of the stop bit, so it'll often :) be 1 if there's
-- a framing error.

-- The parity logic is done bit by bit in a serial fashion, rather
-- than all at once.  Thanks to my ADS class for coming up with this
-- neat trick to reduce the gate count.

-- parity and framing error outputs are held until the reset is
-- pulled (rev 1.2)

-- Once we have a byte, we pull full high.  Asserting clr will clear
-- this and arm us for the next byte.  Obviously, the values in
-- out_data, p_err, and f_err are only guaranteed when full is high.

-- Transmitter Design:

-- The transmitter takes the form of a simple 10 bit shift register,
-- shifting out 0, data(0), data(1)...data(7), odd parity, 1.  The
-- bits are shifted out on every 8th clock, so the clock rate for
-- this is the same as that for the receiver.

-- Random transmitter parity fixed (rev 1.3) by clearing the ptemp
-- bit at the start of each byte, rather than only when reset.

--*******************************************************************
--      Receiver Entity/Architecture Definition
```

```
--*****************************************************************

library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;
use ieee.numeric_std.all;

entity rx is port (
  rst, clr, clk, rxd: in std_logic;
  prescale: in std_logic_vector (3 downto 0);
  byte: out std_logic_vector(7 downto 0);
  p_err, f_err, full: out std_logic);
end rx;

architecture archrx of rx is

signal cnt_reg: unsigned (6 downto 0);
signal pre_cnt: unsigned (3 downto 0);
signal check: std_logic_vector (3 downto 0);
signal temp: std_logic_vector (7 downto 0);
signal ptemp: std_logic;

begin
  byte <= temp;
  process (clk, rst)
  begin
  if rst = '1' then
    -- reset condition
    full <= '0';
    cnt_reg <= "0000000";
    temp <= x"00";
    f_err <= '0';
    p_err <= '0';
    ptemp <= '0';
    pre_cnt <= "0000";
  elsif clk'event and clk = '1' then
    if clr = '1' then
      -- reset for next byte
      full <= '0';
      cnt_reg <= "0000000";
      temp <= x"00";
      f_err <= '0';
      p_err <= '0';
      ptemp <= '0';
      pre_cnt <= "0000";
      check <= rxd & rxd & rxd & rxd;
    elsif pre_cnt = 0 then
      if cnt_reg = 0 then
        -- waiting for start condition - shift data thru start
        -- detect register
        check(3 downto 0) <= check(2 downto 0) & rxd;
        if check = "1000" then
```

```
        -- input 0 for three successive clocks indicates valid
        -- start.
          cnt_reg <= "0000001";
        else
          cnt_reg <= "0000000";
        end if;
      elsif cnt_reg = 73 then
        -- check parity bit
        ptemp <= ptemp xor (not rxd);
        cnt_reg <= cnt_reg + 1;
      elsif cnt_reg = 81 then
        -- stop bit - assert full and check framing condition
        full <= '1';
        f_err <= not rxd;
        p_err <= ptemp;
        check <= "1111";
        cnt_reg <= cnt_reg + 1;
      elsif std_logic_vector(cnt_reg(2 downto 0)) = "001" then
        -- shift in bit
        temp(7 downto 0) <= rxd & temp(7 downto 1);
        ptemp <= ptemp xor rxd;
        cnt_reg <= cnt_reg + 1;
      else
        cnt_reg <= cnt_reg + 1;
      end if;
    end if;
    if pre_cnt = unsigned(prescale) then
       pre_cnt <= "0000";
    else
       pre_cnt <= pre_cnt + 1;
    end if;
  end if;
  end process;

end archrx;


--*************************************************************
--      Transmitter Entity/Architecture Definition
--*************************************************************

library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;
use ieee.numeric_std.all;

entity tx is port (
  rst, go, clk: in std_logic;
  prescale: in std_logic_vector(3 downto 0);
  byte: in std_logic_vector(7 downto 0);
  txd, empty: out std_logic);
end tx;
```

```
architecture archtx of tx is

signal cnt_reg: unsigned (3 downto 0);
signal pre_cnt: unsigned (6 downto 0);
signal temp: std_logic_vector (7 downto 0);
signal ptemp: std_logic;

begin
  process (rst, clk)
  begin
  if rst = '1' then
    -- reset condition
    temp <= x"00";
    cnt_reg <= "0000";
    pre_cnt <= "0000000";
    txd <= '1';
    empty <= '1';
    ptemp <= '0';
  elsif clk'event and clk = '1' then
    if pre_cnt = 0 then
      if cnt_reg = 0 then
        -- stop bit - assert empty, feed out stop bit, and stop
        empty <= '1';
        txd <= '1';
        if go = '1' then cnt_reg <= "0001";
        else cnt_reg <= "0000";
        end if;
      elsif cnt_reg = 1 then
        -- wait for go pulse to be deasserted;
        empty <= '0';
        if go = '1' then cnt_reg <= "0001";
        else cnt_reg <= "0010";
        end if;
      elsif cnt_reg = 2 then
        -- shift out start bit & initialise temp & ptemp
        txd <= '0';
        empty <= '0';
        temp <= byte;
        cnt_reg <= "0011";
        ptemp <= '0';
      elsif cnt_reg = 11 then
        -- shift out parity bit
        txd <= not ptemp;
        empty <= '0';
        cnt_reg <= "0000";
      else
        txd <= temp(0);
        ptemp <= ptemp xor temp(0);
        empty <= '0';
        temp(7 downto 0) <= '0' & temp(7 downto 1);
        cnt_reg <= cnt_reg + 1;
```

```
      end if;
    end if;
    if pre_cnt = unsigned(prescale & "111") then
      pre_cnt <= "0000000";
    else
      pre_cnt <= pre_cnt + 1;
    end if;
  end if;
  end process;

end archtx;
```

A.2 Dataset Source

```
-- MNRF Interface VHDL Code

-- Dataset RS485 Interface Section

-- Revision: 1.2
-- Commenced: 23/8/99.
-- Last Modified: 23/12/900
-- Author: Suzy Jackson sjackson@atnf.csiro.au

-- Requires:  uart.vhd

-- This package contains logic to implement a really raw Dataset,
-- including receiver FIFO, address detection, some error detection,
-- etc.

-- Inputs:

-- rst:  active high reset.
-- clk:  3.6864MHz clock (or 1.8432MHz with baudrates/2)
-- rxd:  asynchronous receive line (to interface, from ACC).
-- baud(3 downto 0): divider for baudrate - bps = 460800/(prescale+1)
-- dsa(4 downto 0): dataset address (should be hardwired to a number)

-- Bidirectional:

-- data(15 downto 0):  Data bus - tristate - gated by the wr output.

-- Outputs:

-- address(8 downto 0):  address being read (or written).
-- wr:  active high read*/write line.
-- stb:  data strobe
-- txd:  asynchronous transmit byte (from interface)
-- txd_en:  active high transmit enable
--          (used for lighting up RS485 transmitter)
-- par:  active high parity error signal.
-- err:  active high framing error signal.
```

```
-- The length and validity of these signals is entirely dependent on
-- the comms (serial) speed.  At a minimum (using a 38.4bps link at
-- maximum rate with a 3.6864MHz crystal) we have the following
-- diagrams:

-- Monitor Read:
--                   ___________________
--   Address:    XXXX___________________XXXX
--
--   Wr:         XXXX___________________XXXX
--                         ________
--   Stb:        _________|        |________
--                            ______
--   Data:       XXXX----XXXXX______>---XXXX

--                  1    2    3    45   6

--  T1-T2 min = 286 us (address setup time)
--  T2-T3 max = 286 us (read time - stb to data valid)
--  T2-T4 min = 572 us (minimum strobe length - data must remain valid)
--  T4-T5 min = 0 (minimum data hold time)

-- Control Write:
--                   ___________________
--   Address:    XXXX___________________XXXX
--                   ___________________
--   Wr:         XXXX                   XXXX
--                         ________
--   Stb:        _________|        |________
--                        ______________
--   Data:       XXXXXXXXX______________XXXX

--                  1    23        4    5

--  T1-T2 min = 286 us (address setup time)
--  T2-T3 min = 270 ns (minimum time between data valid and strobe edge)
--  T3-T4 min = 572 us (minimum strobe length - data valid)
--  T4-T5 min = 286 us (minimum data hold time)

-- rev 1.1:  Added err (framing error) and par (parity error) outputs,
--           so that we can monitor such things with LEDs.

-- rev 1.2:  Changed the timing of the address strobe (STB) output,
--           to guarantee that data is valid before its leading edge,
--           as some equipment (F35 and F61) erroneously latch data
--           on the leading edge of the address strobe.
--           Also removed the HBE signal, as it was not really correct.
--           A separate state machine is now used to adapt the 16 bit
--           dataset engine to an 8 bit buss.

--*****************************************************************
```

```
--      Dataset Entity/Architecture Definition
--******************************************************************

library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;  -- uncomment this line if synthesising
use ieee.numeric_std.all;  -- uncomment this line if simulating

entity dataset is port (
  rst, clk, rxd: in std_logic;
  baud: in std_logic_vector(3 downto 0);
  dsa: in std_logic_vector(4 downto 0);
  data: inout std_logic_vector(15 downto 0);
  add: out std_logic_vector(8 downto 0);
  stb, wr, txd, txd_en, par, err: out std_logic);
end dataset;

architecture archdataset of dataset is

  component rx port(
    rst, clr, clk, rxd: in std_logic;
    prescale: in std_logic_vector(3 downto 0);
    byte: out std_logic_vector(7 downto 0);
    p_err, f_err, full: out std_logic);
  end component;

  component tx port(
    rst, go, clk: in std_logic;
    prescale: in std_logic_vector(3 downto 0);
    byte: in std_logic_vector(7 downto 0);
    txd, empty: out std_logic);
  end component;

  constant ACK: std_logic_vector(7 downto 0):= "00000110";
  constant BEL: std_logic_vector(7 downto 0):= "00000111";
  constant NAK: std_logic_vector(7 downto 0):= "00010101";
  constant SYN: std_logic_vector(7 downto 0):= "00010110";
  constant ESC: std_logic_vector(7 downto 0):= "00011011";
  constant ASC0: std_logic_vector(7 downto 0):= "00110000";
  constant ASC1: std_logic_vector(7 downto 0):= "00110001";
  constant ASC2: std_logic_vector(7 downto 0):= "00110010";
  constant ASC3: std_logic_vector(7 downto 0):= "00110011";
  constant ASC4: std_logic_vector(7 downto 0):= "00110100";

  type rxstate_type is (idle, recv_dsa, recv_fn, recv_fn2,
                        recv_high, recv_high2, recv_low, recv_low2);
  type txreq_type is (idle, send_ack, send_nak);
  type txstate_type is (unload, idle, send_err, send_warn, send_ack,
                        send_high, send_high2, send_low, send_low2,
                        send_nak, cmd_err, cmd_warn);

  signal rxbyte, txbyte, error: std_logic_vector(7 downto 0);
```

```
  signal rxstate: rxstate_type;
  signal txreq: txreq_type;
  signal txstate: txstate_type;
  signal rxfull, p_err, f_err, txgo, rxgo, cmd, txempty: std_logic;
  signal bufdata: std_logic_vector(9 downto 0);
  signal dataout: std_logic_vector(15 downto 0);

--  for SER_RX: rx use entity work.rx;
--  for SER_TX: tx use entity work.tx;

begin

  SER_RX: rx port map (rst, rxgo, clk, rxd, baud, rxbyte, p_err,
                       f_err, rxfull);
  SER_TX: tx port map (rst, txgo, clk, baud, txbyte, txd, txempty);

  data <= dataout when (cmd = '1') else "ZZZZZZZZZZZZZZZZ";
  stb <= '1' when (txstate = send_ack) or (txstate = send_low) or
                  (txstate = send_low2) or (txstate = send_high) or
                  (txstate = send_high2) or (txstate = cmd_err) or
                  (txstate = cmd_warn) else '0';
  wr <= cmd;
  txd_en <= '0' when (txstate = idle) else '1';
  rxgo <= rxfull;
  par <= p_err;
  err <= f_err;

  receive: process (clk, rst)
  begin
  if rst = '1' then
    -- asynchronous reset condition
    rxstate <= idle;
    txstate <= idle;
    txreq <= idle;
    txgo <= '0';
    dataout <= x"0000";
    add <= "000000000";
    txbyte <= x"00";
    error <= x"00";
    cmd <= '0';

  elsif clk'event and clk = '1' then

    case rxstate is

      when idle =>  -- Wait for SYN character
        if rxfull = '1' then
          if (rxbyte = SYN and p_err = '0' and f_err = '0') then
            -- SYN character - next byte should be dataset address
            rxstate <= recv_dsa;
          else
            rxstate <= idle;
```

```
        end if;
      else
        rxstate <= idle;
      end if;

    when recv_dsa =>  -- Get Dataset Address byte
      -- note that we can’t get an escape in this byte.
      if rxfull = ’1’ then
        if (rxbyte(5 downto 1) = dsa(4 downto 0)
            and p_err = ’0’ and f_err = ’0’) then
          -- yay!  We’re being addressed
          rxstate <= recv_fn;
          add(8) <= rxbyte(0);
          cmd <= rxbyte(7);
        elsif (rxbyte = SYN and p_err = ’0’ and f_err = ’0’) then
          -- SYN character - abort.  Next byte should be dataset add
          rxstate <= recv_dsa;
        else
          -- not for us - wait for the next packet
          rxstate <= idle;
        end if;
      else
        rxstate <= recv_dsa;
      end if;

    when recv_fn =>  -- Get Function Address byte
      if rxfull = ’1’ then
        if (p_err = ’1’ or f_err = ’1’) then
          -- malformed byte - send back a NAK.
          error (1) <= ’1’;
          txreq <= send_nak;
          rxstate <= idle;
        elsif rxbyte = ESC then
          -- escape character - function address in next byte
          rxstate <= recv_fn2;
        elsif rxbyte = SYN then
          -- SYN character - abort.  Next byte should be dataset add
          error (2) <= ’1’;
          txreq <= send_nak;
          rxstate <= recv_dsa;
        else
          -- rxbyte contains function address
          add(7 downto 0) <= rxbyte(7 downto 0);
          rxstate <= recv_high;
        end if;
      else
        rxstate <= recv_fn;
      end if;

    when recv_fn2 =>  -- Get Function Address byte after ESC
      if rxfull = ’1’ then
        if (p_err = ’1’ or f_err = ’1’) then
```

```
          -- malformed byte - send back a NAK.
          error (1) <= '1';
          txreq <= send_nak;
          rxstate <= idle;
        elsif rxbyte = ASC0 then
          -- ascii 0 - byte is esc
          add(7 downto 0) <= ESC;
          rxstate <= recv_high;
        elsif rxbyte = ASC1 then
          -- ascii 1 - byte is syn
          add(7 downto 0) <= SYN;
          rxstate <= recv_high;
        elsif rxbyte = SYN then
          -- SYN character - abort.  Next byte should be dataset add
          error (2) <= '1';
          txreq <= send_nak;
          rxstate <= recv_dsa;
        else
          -- byte is broken - send back a NAK
          error (3) <= '1';
          txreq <= send_nak;
          rxstate <= idle;
        end if;
      else
        rxstate <= recv_fn2;
      end if;

    when recv_high =>  -- Get High data byte
      if rxfull = '1' then
        if (p_err = '1' or f_err = '1') then
          -- malformed byte - send back a NAK.
          error (1) <= '1';
          txreq <= send_nak;
          rxstate <= idle;
        elsif rxbyte = ESC then
          -- escape character - high_byte in next byte
          rxstate <= recv_high2;
        elsif rxbyte = SYN then
          -- SYN character - abort.  Next byte should be dataset add
          error (2) <= '1';
          txreq <= send_nak;
          rxstate <= recv_dsa;
        else
          -- byte contains high data byte
          dataout(15 downto 8) <= rxbyte(7 downto 0);
          rxstate <= recv_low;
        end if;
      else
        rxstate <= recv_high;
      end if;

    when recv_high2 =>  -- Get High data byte after ESC
```

```
          if rxfull = '1' then
            if (p_err = '1' or f_err = '1') then
              -- malformed byte - send back a NAK.
              error (1) <= '1';
              txreq <= send_nak;
              rxstate <= idle;
            elsif rxbyte = ASC0 then
              -- ascii 0 - byte is ESC
              dataout(15 downto 8) <= ESC;
              rxstate <= recv_low;
            elsif rxbyte = ASC1 then
              -- ascii 1 - byte is SYN
              dataout(15 downto 8) <= SYN;
              rxstate <= recv_low;
            elsif rxbyte = SYN then
              -- SYN character - abort.  Next byte should be dataset add
              error (2) <= '1';
              txreq <= send_nak;
              rxstate <= recv_dsa;
            else
              -- byte is broken - send back a NAK
              error (3) <= '1';
              txreq <= send_nak;
              rxstate <= idle;
            end if;
          else
            rxstate <= recv_high2;
          end if;

        when recv_low =>  -- Get Low data byte
          if rxfull = '1' then
            if (p_err = '1' or f_err = '1') then
              -- malformed byte - send back a NAK.
              error (1) <= '1';
              txreq <= send_nak;
              rxstate <= idle;
            elsif rxbyte = ESC then
              -- ESC character - low_byte in next byte
              rxstate <= recv_low2;
            elsif rxbyte = SYN then
              -- SYN character - abort.  Next byte should be dataset add
              error (2) <= '1';
              txreq <= send_nak;
              rxstate <= recv_dsa;
            else
              -- byte contains low data byte
              dataout(7 downto 0) <= rxbyte(7 downto 0);
              txreq <= send_ack;
              rxstate <= idle;
            end if;
          else
            rxstate <= recv_low;
```

```
      end if;

    when recv_low2 =>  -- Get Low data byte after ESC
      if rxfull = '1' then
        if (p_err = '1' or f_err = '1') then
          -- malformed byte - send back a NAK.
          error (1) <= '1';
          txreq <= send_nak;
          rxstate <= idle;
        elsif rxbyte = ASC0 then
          -- ascii 0 - byte is ESC
          dataout(7 downto 0) <= ESC;
          txreq <= send_ack;
          rxstate <= idle;
        elsif rxbyte = ASC1 then
          -- ascii 1 - byte is SYN
          dataout(7 downto 0) <= SYN;
          txreq <= send_ack;
          rxstate <= idle;
        elsif rxbyte = SYN then
          -- SYN character - abort.  Next byte should be dataset add
          error (2) <= '1';
          txreq <= send_nak;
          rxstate <= recv_dsa;
        else
          -- byte is broken - send back a NAK
          error (3) <= '1';
          txreq <= send_nak;
          rxstate <= idle;
        end if;
      else
        rxstate <= recv_low2;
      end if;

    when others =>
      rxstate <= idle;

  end case;

  case txstate is

    when send_ack =>  -- Send ACK code
      if (txempty = '1' and txgo = '0') then
        txbyte <= ACK;
        txgo <= '1';
        txreq <= idle;
        if cmd = '1' then
          -- command - no need to send out byte
          txstate <= cmd_err;
        else
          -- monitor - need to supply byte
          txstate <= send_high;
```

```
        end if;
      elsif txempty = '0' then
        txstate <= send_ack;
        txgo <= '0';
      else
        txstate <= send_ack;
        txgo <= '1';
      end if;

    when send_high =>  -- Send high byte
      if (txempty = '1' and txgo = '0') then
        txgo <= '1';
        if (data(15 downto 8) = ACK or data(15 downto 8) = NAK or
            data(15 downto 8) = BEL or data(15 downto 8) = ESC) then
          -- gotta send out an ESC
          txbyte <= ESC;
          txstate <= send_high2;
        else
          -- out goes the byte
          txstate <= send_low;
          txbyte(7 downto 0) <= data(15 downto 8);
        end if;
      elsif txempty = '0' then
        txstate <= send_high;
        txgo <= '0';
      else
        txstate <= send_high;
        txgo <= '1';
      end if;

    when send_high2 =>  -- Send high byte after ESC
      if (txempty = '1' and txgo = '0') then
        txgo <= '1';
        if data(15 downto 8) = ESC then
          -- send out ascii 0
          txbyte <= ASC0;
        elsif data(15 downto 8) = ACK then
          -- send out ascii 2
          txbyte <= ASC2;
        elsif data(15 downto 8) = BEL then
          -- send out ascii 3
          txbyte <= ASC3;
        else
          -- send out ascii 4
          txbyte <= ASC4;
        end if;
        txstate <= send_low;
      elsif txempty = '0' then
        txstate <= send_high2;
        txgo <= '0';
      else
        txstate <= send_high2;
```

```
        txgo <= '1';
      end if;

    when send_low =>  -- Send low byte
      if (txempty = '1' and txgo = '0') then
        txgo <= '1';
        if (data(7 downto 0) = ACK or data(7 downto 0) = NAK or
            data(7 downto 0) = BEL or data(7 downto 0) = ESC) then
          -- gotta send out an ESC
          txbyte <= ESC;
          txstate <= send_low2;
        else
          -- out goes the byte
          txstate <= unload;
          txbyte <= data(7 downto 0);
        end if;
      elsif txempty = '0' then
        txstate <= send_low;
        txgo <= '0';
      else
        txstate <= send_low;
        txgo <= '1';
      end if;

    when send_low2 =>  -- Send low byte after ESC
      if (txempty = '1' and txgo = '0') then
        txgo <= '1';
        if data(7 downto 0) = ESC then
          -- send out ascii 0
          txbyte <= ASC0;
        elsif data(7 downto 0) = ACK then
          -- send out ascii 2
          txbyte <= ASC2;
        elsif data(7 downto 0) = BEL then
          -- send out ascii 3
          txbyte <= ASC3;
        else
          -- send out ascii 4
          txbyte <= ASC4;
        end if;
        txstate <= unload;
      elsif txempty = '0' then
        txstate <= send_low2;
        txgo <= '0';
      else
        txstate <= send_low2;
        txgo <= '1';
      end if;

    when send_nak =>  -- Send NAK code
      if (txempty = '1' and txgo = '0') then
        txbyte <= NAK;
```

```
        txgo <= '1';
        txreq <= idle;
        txstate <= send_err;
      elsif txempty = '0' then
        txstate <= send_nak;
        txgo <= '0';
      else
        txstate <= send_nak;
        txgo <= '1';
      end if;

    when send_err =>  -- Send error codes
      if (txempty = '1' and txgo = '0') then
        txbyte <= error;
        error <= x"00";
        txgo <= '1';
        txreq <= idle;
        txstate <= send_warn;
      elsif txempty = '0' then
        txstate <= send_err;
        txgo <= '0';
      else
        txstate <= send_err;
        txgo <= '1';
      end if;

    when send_warn =>  -- Send warning codes - not implemented.
      if (txempty = '1' and txgo = '0') then
        txbyte <= x"00";
        txgo <= '1';
        txreq <= idle;
        txstate <= unload;
      elsif txempty = '0' then
        txstate <= send_warn;
        txgo <= '0';
      else
        txstate <= send_warn;
        txgo <= '1';
      end if;

    when cmd_err =>
      -- Send error codes for command (no error condition exists)
      if (txempty = '1' and txgo = '0') then
        txbyte <= x"00";
        txgo <= '1';
        txreq <= idle;
        txstate <= cmd_warn;
      elsif txempty = '0' then
        txstate <= cmd_err;
        txgo <= '0';
      else
        txstate <= cmd_err;
```

```
          txgo <= '1';
        end if;

      when cmd_warn =>
        -- Send warning codes for command (no error condition exists)
        if (txempty = '1' and txgo = '0') then
          txbyte <= x"00";
          txgo <= '1';
          txreq <= idle;
          txstate <= unload;
        elsif txempty = '0' then
          txstate <= cmd_warn;
          txgo <= '0';
        else
          txstate <= cmd_warn;
          txgo <= '1';
        end if;

      when unload =>
        -- make sure we flush the transmit buffer
        if (txempty = '1' and txgo = '0') then
          txstate <= idle;
        elsif txempty = '0' then
          txgo <= '0';
          txstate <= unload;
        else
          txstate <= unload;
          txgo <= '1';
        end if;

      when idle =>
        if txreq = send_ack then
          txstate <= send_ack;
        elsif txreq = send_nak then
          txstate <= send_nak;
        else txstate <= idle;
        end if;
      when others =>
        txstate <= idle;

    end case;

  end if;
  end process;

end archdataset;
```

A.3 F83 ADC Sequencer Source

```
-- MNRF Interface VHDL Code
```

```
-- F83 ADC Read Tool

-- Revision: 1.0
-- Commenced: 21/2/01.
-- Last Modified: 21/2/01
-- Author: Suzy Jackson sjackson@atnf.csiro.au

-- This package contains the hardware necessary to drive the
-- LTC1605-2 ADC as found on the F83 interface board.

-- Unlike adcread, which scales and offsets the result to maintain
-- some level of compatability with a D2 dataset, this code simply
-- presents the ADC data unmolested, in 16 bit signed format.

-- Inputs:

-- rst:      Active high reset signal.
-- clk:      Typically the 3.6864MHz system clock, but should be
--           fairly flexible.
-- adcrq:    ADC request input.  Should be driven from an address
--           decoder.
-- stb:      Address strobe.  Connect straight to the relevant
--           dataset module line.  This line in conjunction with the
--           ane line enables the data tri-states.
-- adcdata:  8 bit muxed data input from ADC.
-- busy:     Busy output from ADC (active low).

-- Outputs:

-- data:  16 bit Tri states containing result of the ADC conversion
-- anst:  ADC conversion start pulse.
-- adcen:  ADC enable line (active low)
-- anhigh:  Select between bytes of ADC result.

--*************************************************************
--      ADCRd16 Entity/Architecture Definition
--*************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;  -- uncomment this line if synthesising
--use ieee.numeric_std.all;  -- uncomment this line if simulating

entity adcrd16 is port (
  rst, clk, adcrq, stb, busy: in std_logic;
  adcdata: in std_logic_vector (7 downto 0);
  data: inout std_logic_vector (15 downto 0);
  anst, adcen, anhigh: out std_logic);
end adcrd16;

architecture archadc of adcrd16 is
```

```
type sttype is (start, high, low, check, idle, finished, acq);
signal state: sttype;
signal result: std_logic_vector (15 downto 0);
signal delay: unsigned (5 downto 0);

begin

  data <= result when (adcrq = '1' and stb = '1') else "ZZZZZZZZZZZZZZZZ";

  with state select
    anst <= '0' when start,
            '1' when others;
  with adcrq select
    adcen <= '1' when '0',
             '0' when others;
  with state select
    anhigh <= '1' when high,
              '0' when others;

  process (clk, rst)
    begin
    if rst = '1' then
      -- reset condition
      state <= idle;
      result <= x"0000";

    elsif clk'event and clk = '1' then
      case state is
        when idle =>  -- wait for next valid strobe
          delay <= "000000";
          if (adcrq = '1' and stb = '1') then
            state <= acq;
          else state <= idle;
          end if;
        when acq =>  -- Acquire delay - 64 clocks (8.7us at 7.3728MHz)
          delay <= delay + 1;
          if delay = 63 then state <= start;
          else state <= acq;
          end if;
        when start =>  -- ADC read - start conversion
          state <= check;
        when check => -- ADC read - wait for busy to fall.
          if (busy = '0') then state <= check;
          else state <= high;
          end if;
        when high =>  -- ADC read - latch high byte
          result (15 downto 8) <= adcdata;
          state <= low;
        when low =>  -- ADC read - latch low byte
          result (7 downto 0) <= adcdata;
          state <= finished;
        when finished =>  -- All done - waiting for stb to fall;
```

```
          if stb = '1' then state <= finished;
          else state <= idle;
          end if;
        when others =>
          state <= idle;

      end case;
    end if;
  end process;

end archadc;
```

A.4 F83 Bus Controller Source

```
-- MNRF Interface VHDL Code

-- Dataset 8 bit buss adapter

-- Revision: 1.0
-- Commenced: 27/2/2001.
-- Last Modified: 27/2/2001
-- Author: Suzy Jackson sjackson@atnf.csiro.au

-- This package contains a simple sequencer to drive an 8 bit buss from
-- the 16 bit buss provided by the dataset block.

-- Inputs:

-- rst:  active high reset.
-- clk:  3.6864MHz clock (nominally)
-- stb16: active high data strobe from dataset block
-- wr:  active high read*/write line from dataset block
-- en:  active high enable line.

-- Bidirectional:

-- Data_16 (15 downto 0):  16 bit data buss from dataset
-- Data_8 (7 downto 0):  8 bit data buss

-- Outputs:

-- stb8:  data strobe
-- hbe:  a line indicating whether the high byte (1) or low byte (0)
--       is being used.

-- Monitor Read:
--
--   Wr:         XXXX___________________________XXXX
--                         ----------------
--   Stb16:      _________|                |________
--                               -----------
```

```
--   Data16:     XXXX------<XXXXX___________>---XXXX
--                          ___
--   Hbe:        __________|   |____________________
--                           _   _
--   Stb8:       ___________| |_| |_________________
--                           __  __
--   Data8:      XXXX-------<__><__>------------XXXX

-- Control Write:
--                   ___________________________
--   Wr:         XXXX                           XXXX
--                         ________
--   Stb16:      _________|        |________________
--                        ______________
--   Data16:     XXXXXXXXXX_____________XXXXXXXXXXXX
--                                   ___
--   Hbe:        ___________________|   |___________
--                                    _   _
--   Stb8:       ____________________| |_| |________
--                                   ___ ___
--   Data8:      XXXXXXXXXXXXXXXXXXXX___X___XXXXXXXX


--*****************************************************************
--    buss8 Entity/Architecture Definition
--*****************************************************************

library ieee;
use ieee.std_logic_1164.all;

entity buss8 is port (
  rst, clk, stb16, wr, en: in std_logic;
  data16: inout std_logic_vector(15 downto 0);
  data8:  inout std_logic_vector (7 downto 0);
  stb8, hbe: out std_logic);
end buss8;

architecture archbus of buss8 is


  type state_type is (idle, rd_a, rd_b, rd_c, rd_d, rd_e, rd_f,
                      wr_a, wr_b, wr_c, wr_d, wr_e, wr_f, wr_g);
  signal state: state_type;
  signal data: std_logic_vector (15 downto 0);

begin


  data16 <= data when (wr = '0') and (stb16 = '1') and (en = '1') else
            "ZZZZZZZZZZZZZZZZ";

  hbe <= '1' when state = rd_a else
```

```
        '1' when state = rd_b else
        '1' when state = rd_c else
        '1' when state = wr_b else
        '1' when state = wr_c else
        '1' when state = wr_d else
        '0';

stb8 <= '1' when state = rd_b else
        '1' when state = rd_e else
        '1' when state = wr_c else
        '1' when state = wr_f else
        '0';

data8 <= data (15 downto 8) when state = wr_b else
         data (15 downto 8) when state = wr_c else
         data (15 downto 8) when state = wr_d else
         data (7 downto 0) when state = wr_e else
         data (7 downto 0) when state = wr_f else
         data (7 downto 0) when state = wr_g else
         "ZZZZZZZZ";

process (clk, rst)
begin
if rst = '1' then
  -- asynchronous reset condition
  state <= idle;

elsif clk'event and clk = '1' then

  case state is

    when idle =>  -- Waiting for something to happen
      if (wr = '0') and (stb16 = '1') and (en = '1') then state <= rd_a;
      elsif (wr = '1') and (stb16 = '1') and (en = '1') then state <= wr_a;
      else state <= idle;
      end if;
    when rd_a =>
      state <= rd_b;
    when rd_b =>
      data (15 downto 8) <= data8;
      state <= rd_c;
    when rd_c =>
      state <= rd_d;
    when rd_d =>
      state <= rd_e;
    when rd_e =>
      data (7 downto 0) <= data8;
      state <= rd_f;
    when rd_f =>
      if stb16 = '0' then state <= idle;
      else state <= rd_f;
      end if;
```

```
      when wr_a =>  --  Wait for stb16 to fall again
        if stb16 = '0' then state <= wr_b;
        else
          data <= data16;
          state <= wr_a;
        end if;
      when wr_b =>
        state <= wr_c;
      when wr_c =>
        state <= wr_d;
      when wr_d =>
        state <= wr_e;
      when wr_e =>
        state <= wr_f;
      when wr_f =>
        state <= wr_g;
      when wr_g =>
        state <= idle;
      when others =>
        state <= idle;

    end case;

  end if;
  end process;

end archbus;
```

A.5 WVR ADC Sequencer Source

```
-- MNRF Interface VHDL Code

-- WVR ADC Read Tool

-- Revision: 1.0
-- Commenced: 10/8/01.
-- Last Modified: 10/8/01
-- Author: Suzy Jackson sjackson@atnf.csiro.au

-- This package contains the hardware necessary to drive the ADS1252
-- ADC as found on the WVR interface board.

-- Inputs:

-- rst:      Active high reset signal.
-- clk:      3.6864MHz system clock.
-- add:      Dataset address lines (3 bits).
-- stb:      Address strobe.  Connect straight to the relevant dataset
--           module line.
--           This line in conjunction with the ane line enables the
--           data tri-states.
```

```
-- ane:      Address enable.  Connect to the address decoder.
-- adcdata:  Serial data input from ADC.

-- Outputs:

-- data:     24 bit Tri states containing result of the addressed ADC
--           conversion
-- adcclk:   ADC conversion clock.
-- adcsclk:  ADC serial clock.
-- muxadd:   Multiplexer address.

-- The ADC is a little tricky to control, both because its speed is
-- excessively slow, and because there is no simple way to start
-- conversions.  The ADC simply runs asynchronously, outputting data
-- when it has finished a conversion.

-- These limitations are dealt with in two ways.  First, all eight
-- channels are locally buffered.  The ADC sequencer simply updates all
-- eight registers every 20msec.

-- In order to do this, the ADC is run at a 1.6KHz conversion rate.
-- This allows eight conversions per channel, satisfying the ADC
-- digital filter settling requirement of six conversions.  All but the
-- eighth conversion are discarded.

-- This results in a clk frequency of 614,400 Hz, or 1/6th of the
-- oscillator frequency.

-- The mux is updated after each conversion.

--*****************************************************************
--      wvradc24 Entity/Architecture Definition
--*****************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;  -- uncomment this line if synthesising
--use ieee.numeric_std.all;  -- uncomment this line if simulating

entity wvradc24 is port (
  rst, clk, stb, ane, adcdata: in std_logic;
  add: in std_logic_vector (3 downto 0);
  data: inout std_logic_vector (23 downto 0);
  muxadd: out std_logic_vector (2 downto 0);
  mux1en, mux2en, adcclk, adcsclk: out std_logic);
end wvradc24;

architecture archadc of wvradc24 is

type sttype is (idle, dout, getdata, check, transfer);
signal state: sttype;
signal result0: std_logic_vector (23 downto 0);
```

```
signal result1: std_logic_vector (23 downto 0);
signal result2: std_logic_vector (23 downto 0);
signal result3: std_logic_vector (23 downto 0);
signal result4: std_logic_vector (23 downto 0);
signal result5: std_logic_vector (23 downto 0);
signal result6: std_logic_vector (23 downto 0);
signal result7: std_logic_vector (23 downto 0);
signal result8: std_logic_vector (23 downto 0);
signal result9: std_logic_vector (23 downto 0);
signal result10: std_logic_vector (23 downto 0);
signal result11: std_logic_vector (23 downto 0);
signal result12: std_logic_vector (23 downto 0);
signal result13: std_logic_vector (23 downto 0);
signal result14: std_logic_vector (23 downto 0);
signal result15: std_logic_vector (23 downto 0);
signal serclk: std_logic;
signal temp: std_logic_vector (23 downto 0);
signal waitcnt: unsigned (5 downto 0);
signal bit: unsigned (4 downto 0);
signal channel: unsigned (3 downto 0);
signal clkdiv: unsigned (2 downto 0);
signal sixth: unsigned (3 downto 0);

begin

  data <= result0 when (ane = '1' and stb = '1' and add = "0000") else
          result1 when (ane = '1' and stb = '1' and add = "0001") else
          result2 when (ane = '1' and stb = '1' and add = "0010") else
          result3 when (ane = '1' and stb = '1' and add = "0011") else
          result4 when (ane = '1' and stb = '1' and add = "0100") else
          result5 when (ane = '1' and stb = '1' and add = "0101") else
          result6 when (ane = '1' and stb = '1' and add = "0110") else
          result7 when (ane = '1' and stb = '1' and add = "0111") else
          result8 when (ane = '1' and stb = '1' and add = "1000") else
          result9 when (ane = '1' and stb = '1' and add = "1001") else
          result10 when (ane = '1' and stb = '1' and add = "1010") else
          result11 when (ane = '1' and stb = '1' and add = "1011") else
          result12 when (ane = '1' and stb = '1' and add = "1100") else
          result13 when (ane = '1' and stb = '1' and add = "1101") else
          result14 when (ane = '1' and stb = '1' and add = "1110") else
          result15 when (ane = '1' and stb = '1' and add = "1111") else
          "ZZZZZZZZZZZZZZZZZZZZZZZZ";

  muxadd <= std_logic_vector (channel (2 downto 0));
  mux1en <= not std_logic (channel(3));
  mux2en <= std_logic (channel(3));
  adcsclk <= serclk;

  process (clk, rst) begin
    if rst = '1' then
      adcclk <= '0';
      clkdiv <= "000";
```

```
    waitcnt <= "000000";
    state <= idle;
    bit <= "00000";
    channel <= "0000";
    serclk <= '0';
    sixth <= "0000";

  elsif clk'event and clk = '1' then

    -- generate adc clock (614400 Hz)
    if clkdiv = 2 then
      adcclk <= '1';
      clkdiv <= clkdiv + 1;
    elsif clkdiv = 5 then
      clkdiv <= "000";
      adcclk <= '0';
    else clkdiv <= clkdiv + 1;
    end if;

    case state is
      when idle =>  -- wait for data line to go low
        waitcnt <= "000000";
        if adcdata = '0' then state <= dout;
        else state <= idle;
        end if;
      when dout =>  -- wait 64 clocks.
        if adcdata = '1' then
          if waitcnt = 63 then
            state <= getdata;
            bit <= "00000";
          else
            waitcnt <= waitcnt + 1;
            state <= dout;
          end if;
        else state <= dout;
        end if;
      when getdata =>  -- data available - clock into temp register.
        if bit = 24 then
          state <= check;
          serclk <= '0';
        elsif serclk = '0' then
          temp <= temp (22 downto 0) & not adcdata;
          bit <= bit + 1;
          state <= getdata;
          serclk <= '1';
        else
          serclk <= '0';
          state <= getdata;
        end if;
      when check =>
        -- we only want the sixth conversion on a given mux address...
        sixth <= sixth + 1;
```

```
        if sixth = 15 then state <= transfer;
        else state <= idle;
        end if;
      when transfer =>  -- transfer temp into relevant result register.
        sixth <= "0000";
        if (ane = '0' or stb = '0') then
          -- make sure we can't conflict with a read of the register
          if channel = 0 then result0 <= temp;
          elsif channel = 1 then result1 <= temp;
          elsif channel = 2 then result2 <= temp;
          elsif channel = 3 then result3 <= temp;
          elsif channel = 4 then result4 <= temp;
          elsif channel = 5 then result5 <= temp;
          elsif channel = 6 then result6 <= temp;
          elsif channel = 7 then result7 <= temp;
          elsif channel = 8 then result8 <= temp;
          elsif channel = 9 then result9 <= temp;
          elsif channel = 10 then result10 <= temp;
          elsif channel = 11 then result11 <= temp;
          elsif channel = 12 then result12 <= temp;
          elsif channel = 13 then result13 <= temp;
          elsif channel = 14 then result14 <= temp;
          else result15 <= temp;
          end if;
          channel <= channel + 1;
          state <= idle;
        else state <= transfer;
        end if;
      when others =>
        state <= idle;

    end case;
  end if;
end process;

end archadc;
```

Appendix B

F83 Interface Schematics

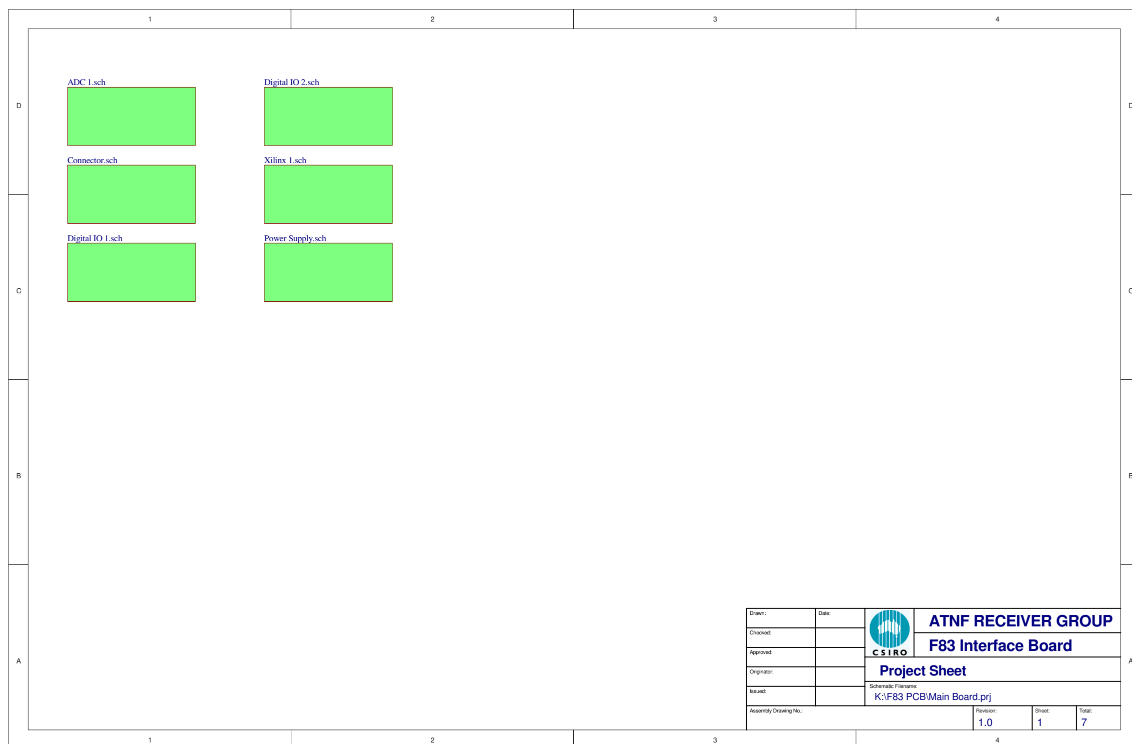

Figure B.1: F83 interface project sheet

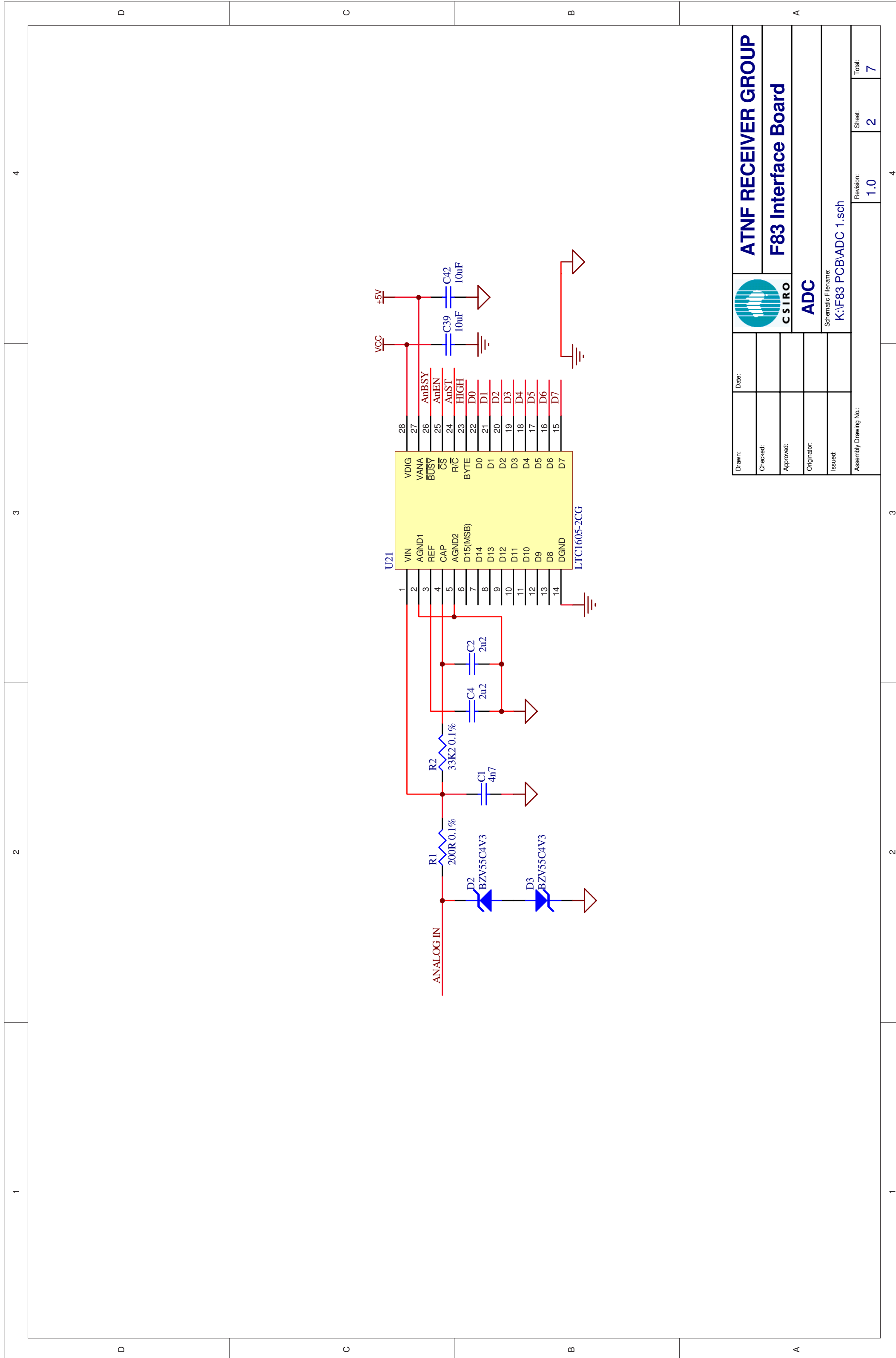

Figure B.2: F83 interface ADC schematic

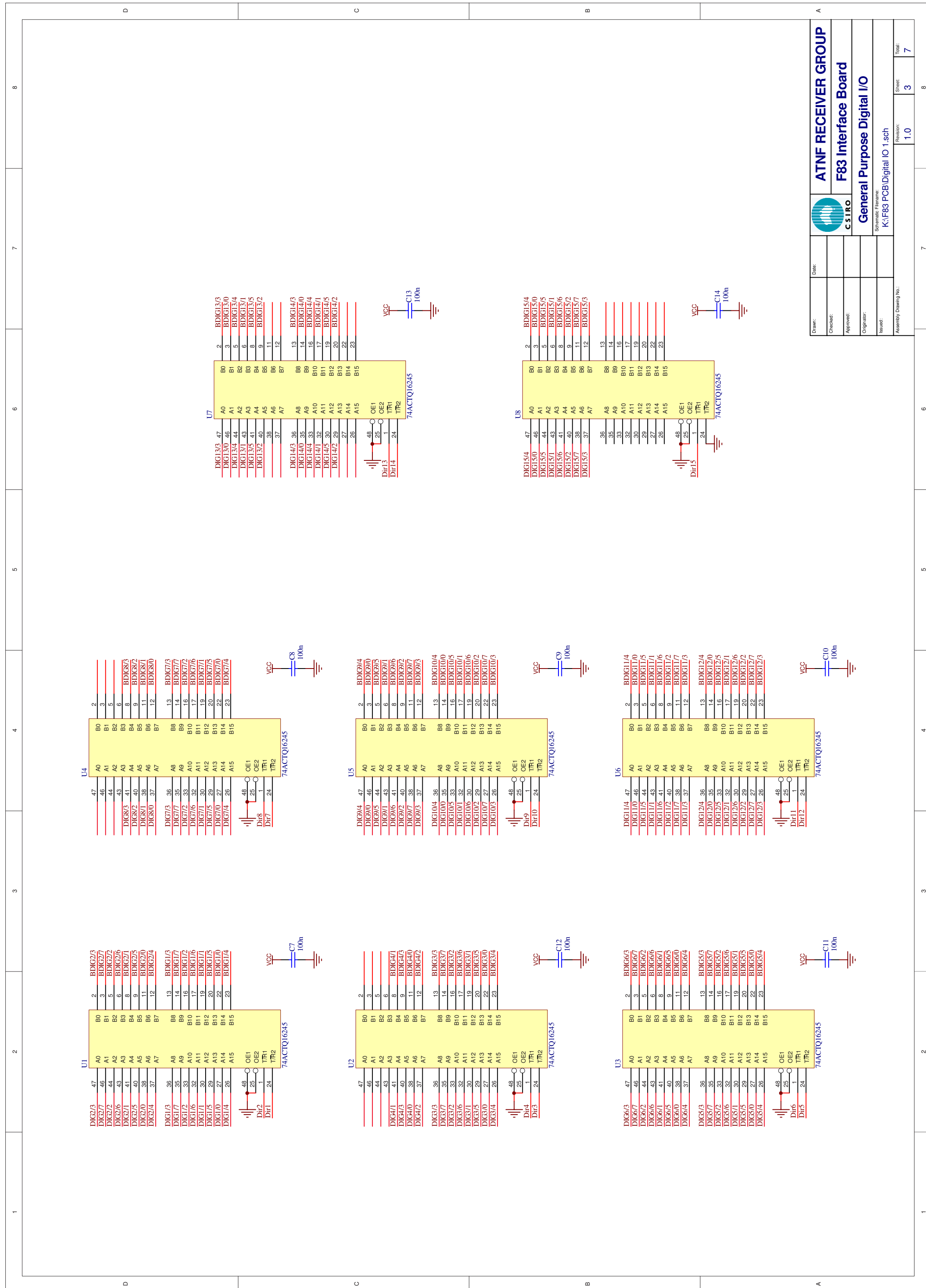

Figure B.3: F83 interface digital I/O 1 schematic

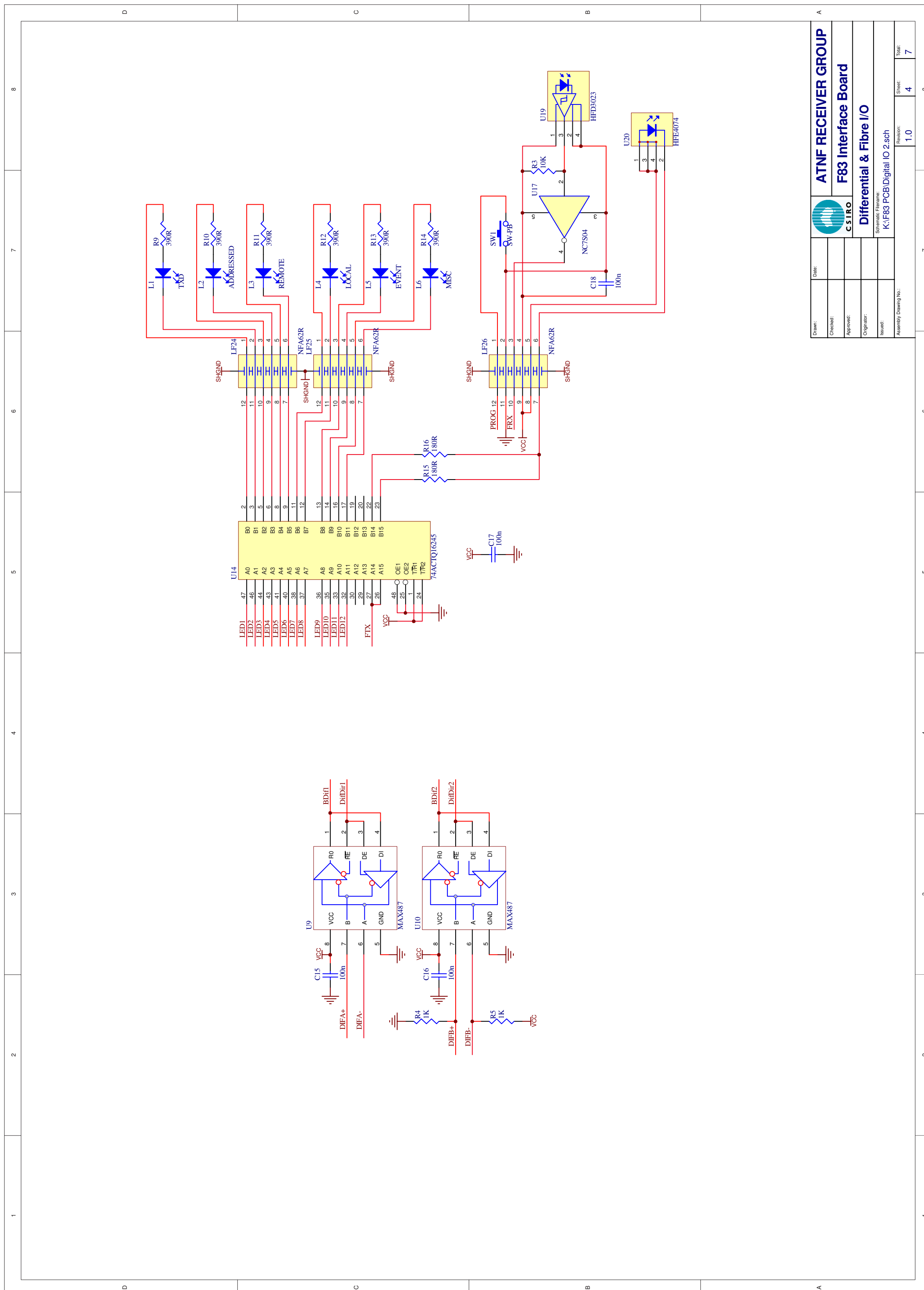

Figure B.4: F83 interface digital I/O 2 schematic

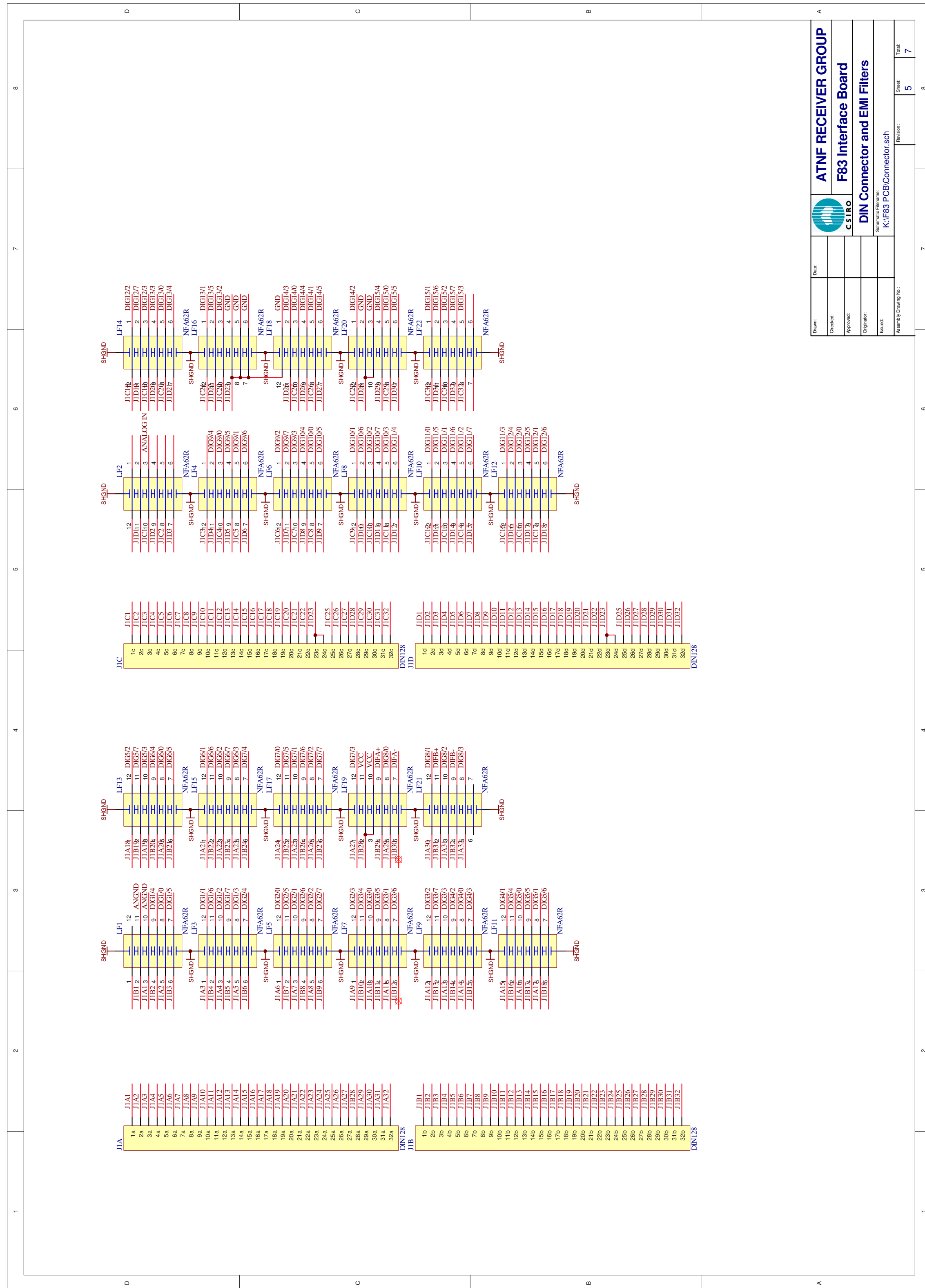

Figure B.5: F83 interface RFI filtering schematic

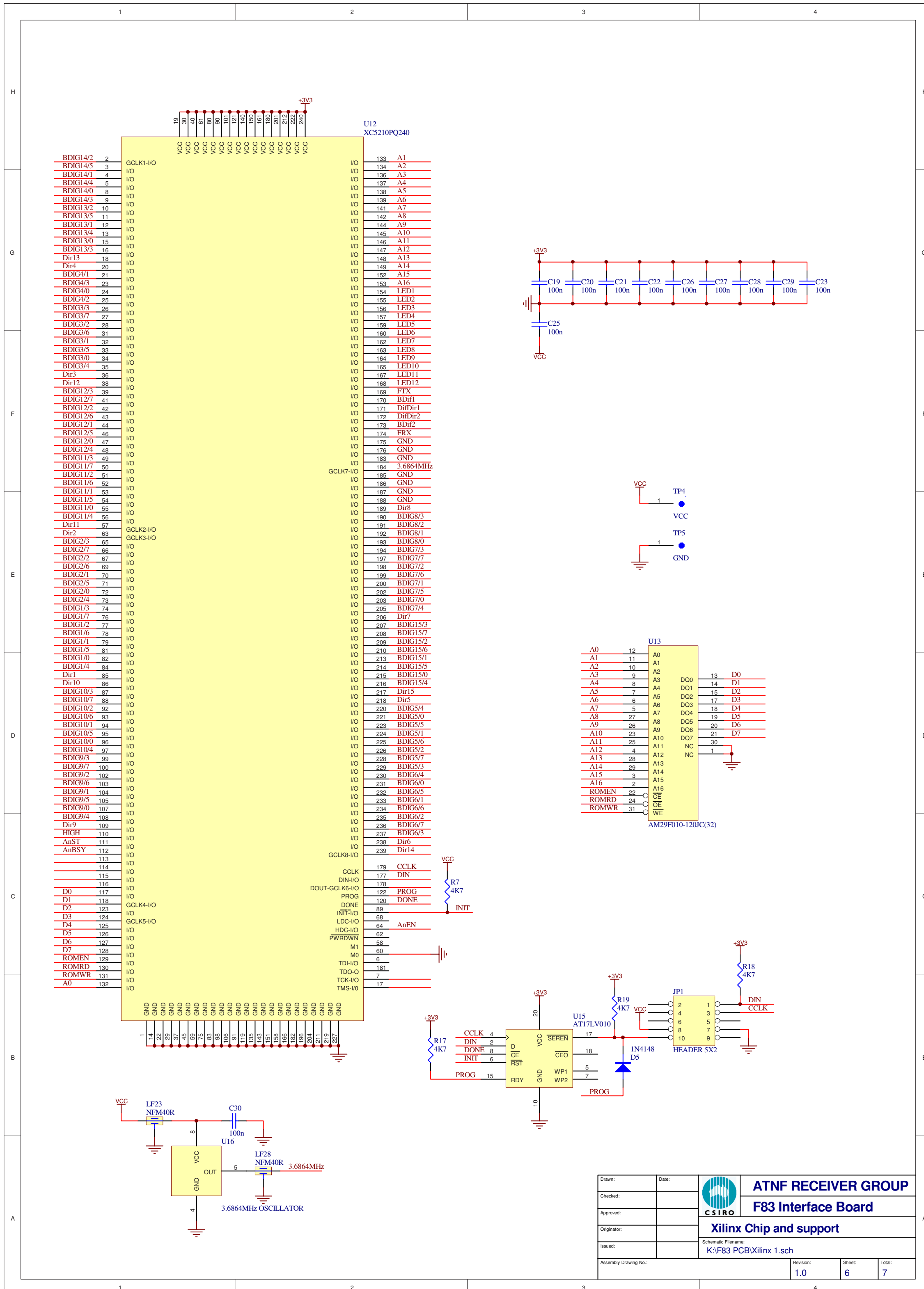

Figure B.6: F83 interface Xilinx support schematic

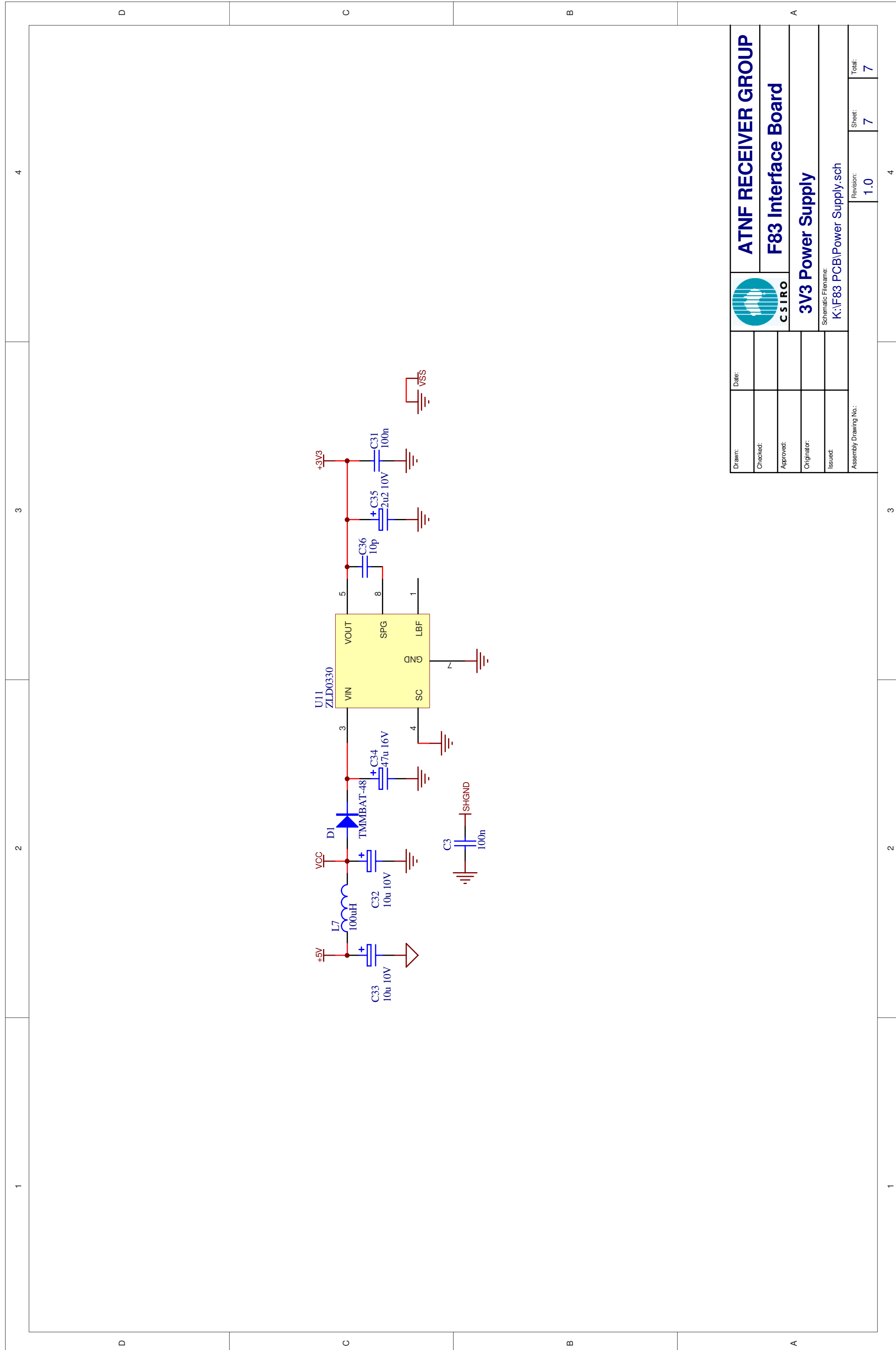

Figure B.7: F83 interface power supply schematic

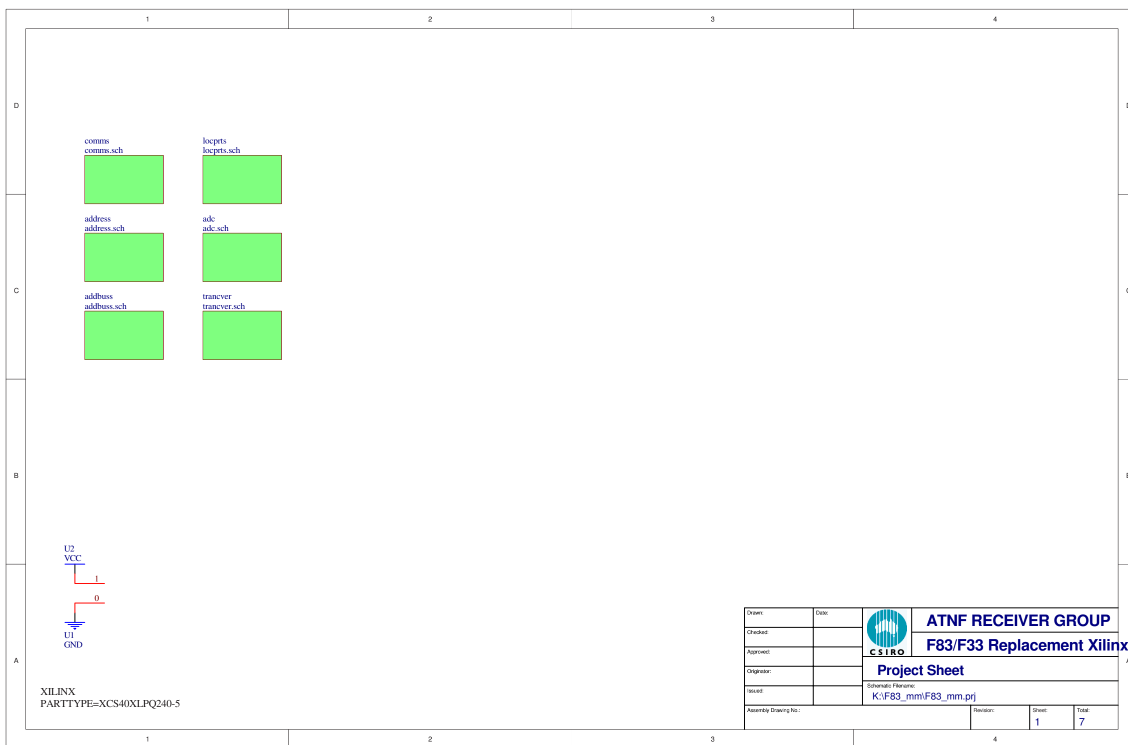

Figure B.8: F83 interface Xilinx project sheet

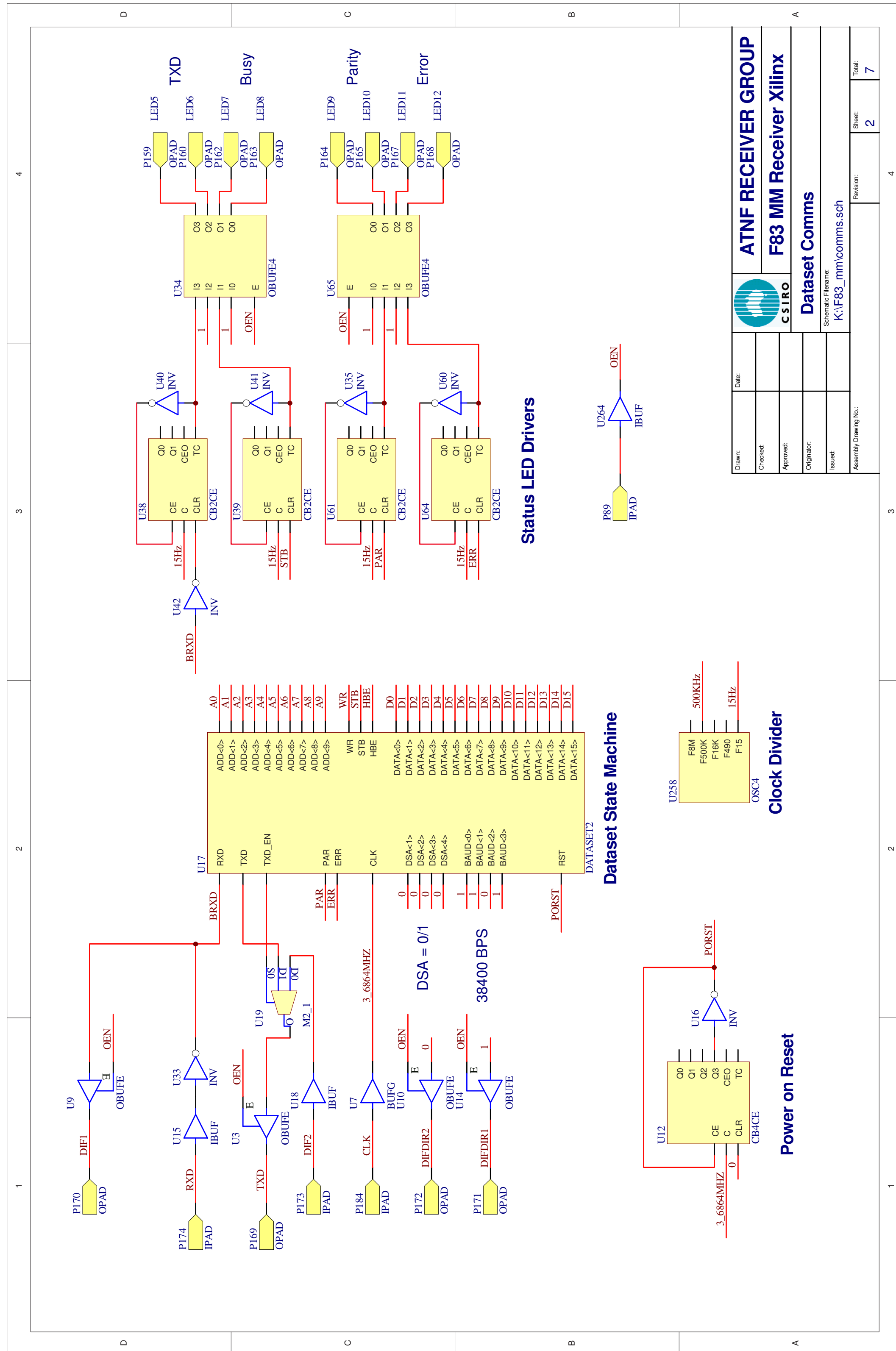

Figure B.9: F83 interface Xilinx comms schematic

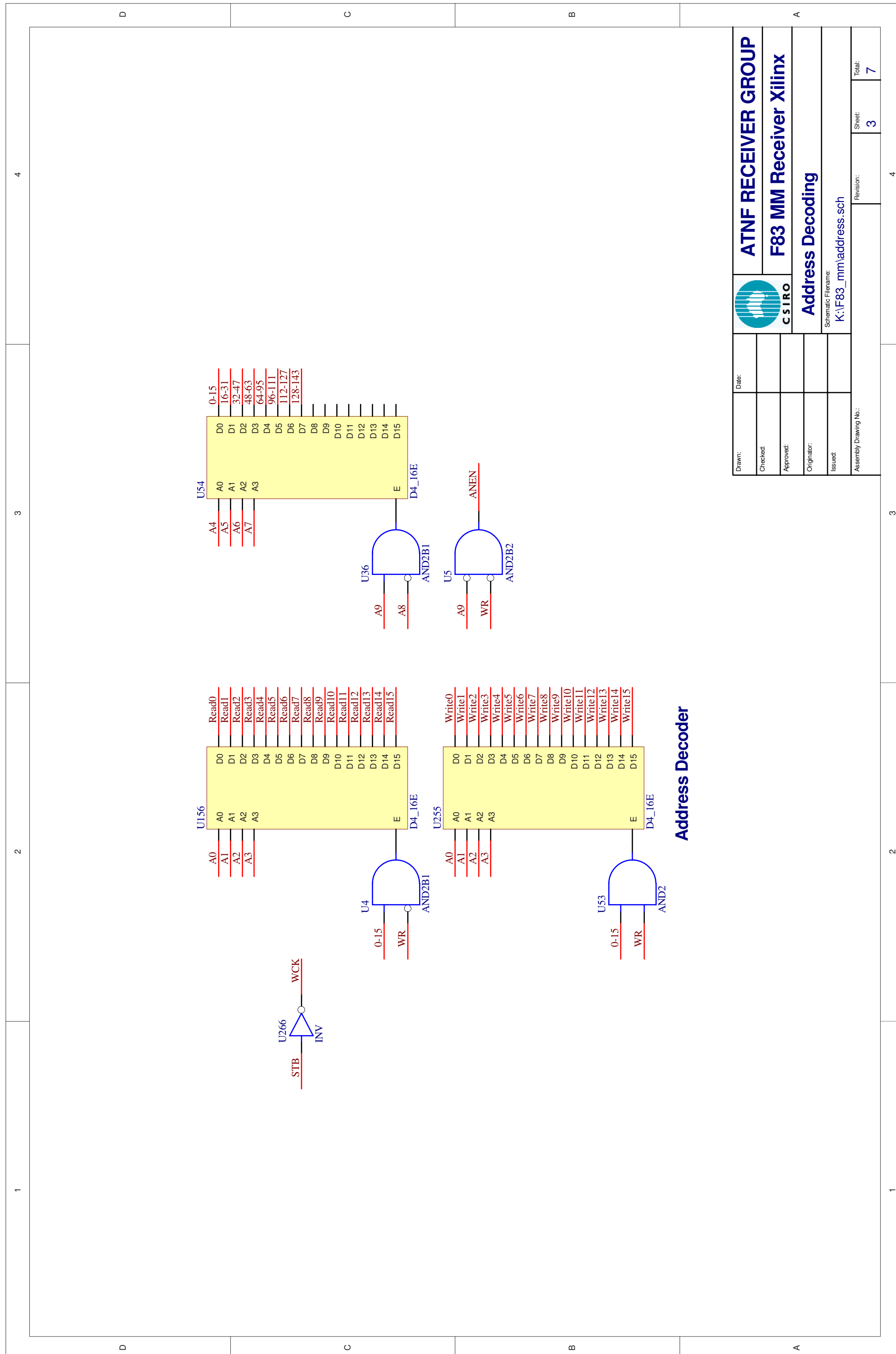

Figure B.10: F83 interface Xilinx address decode schematic

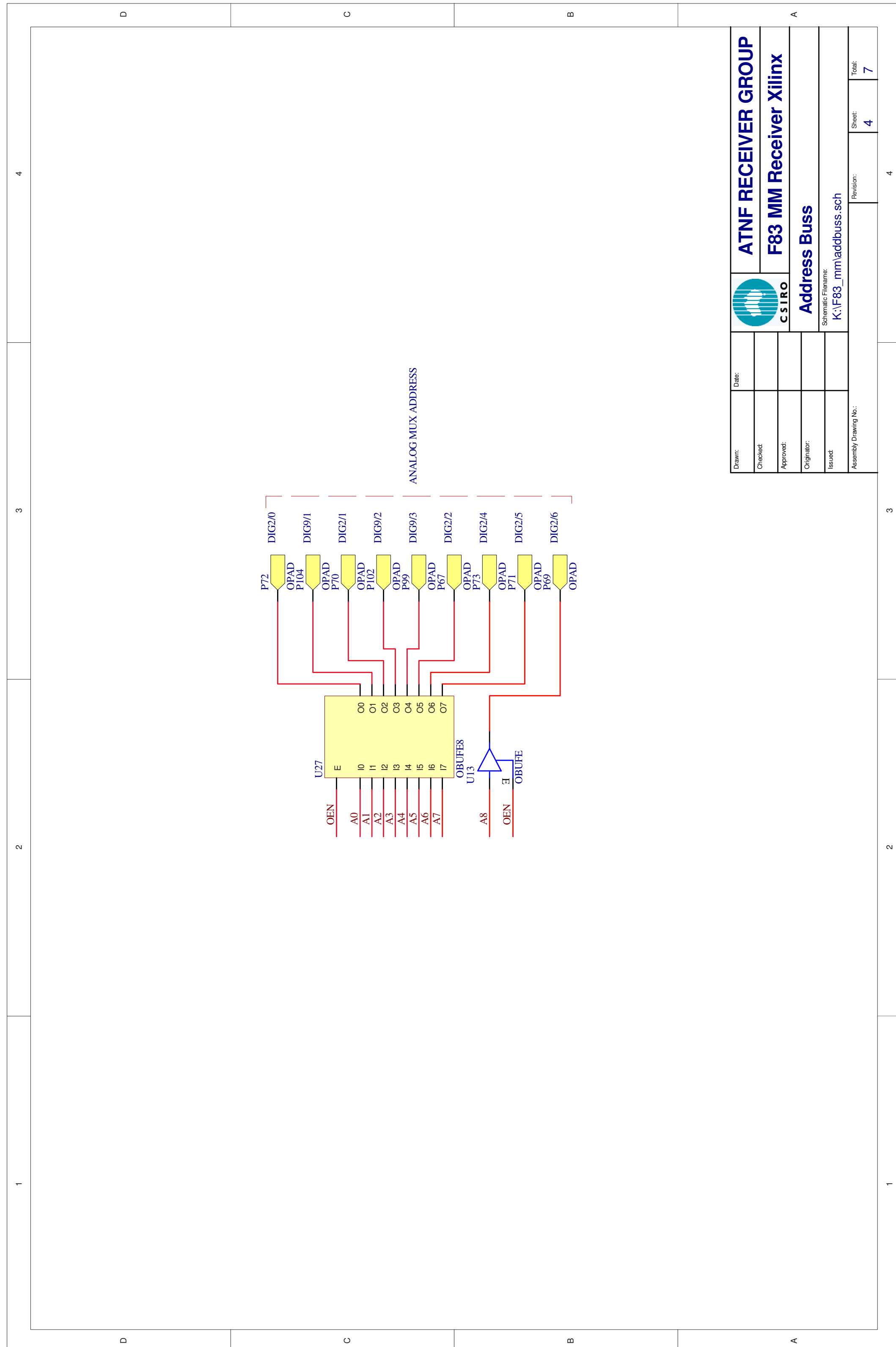

Figure B.11: F83 interface Xilinx address buss schematic

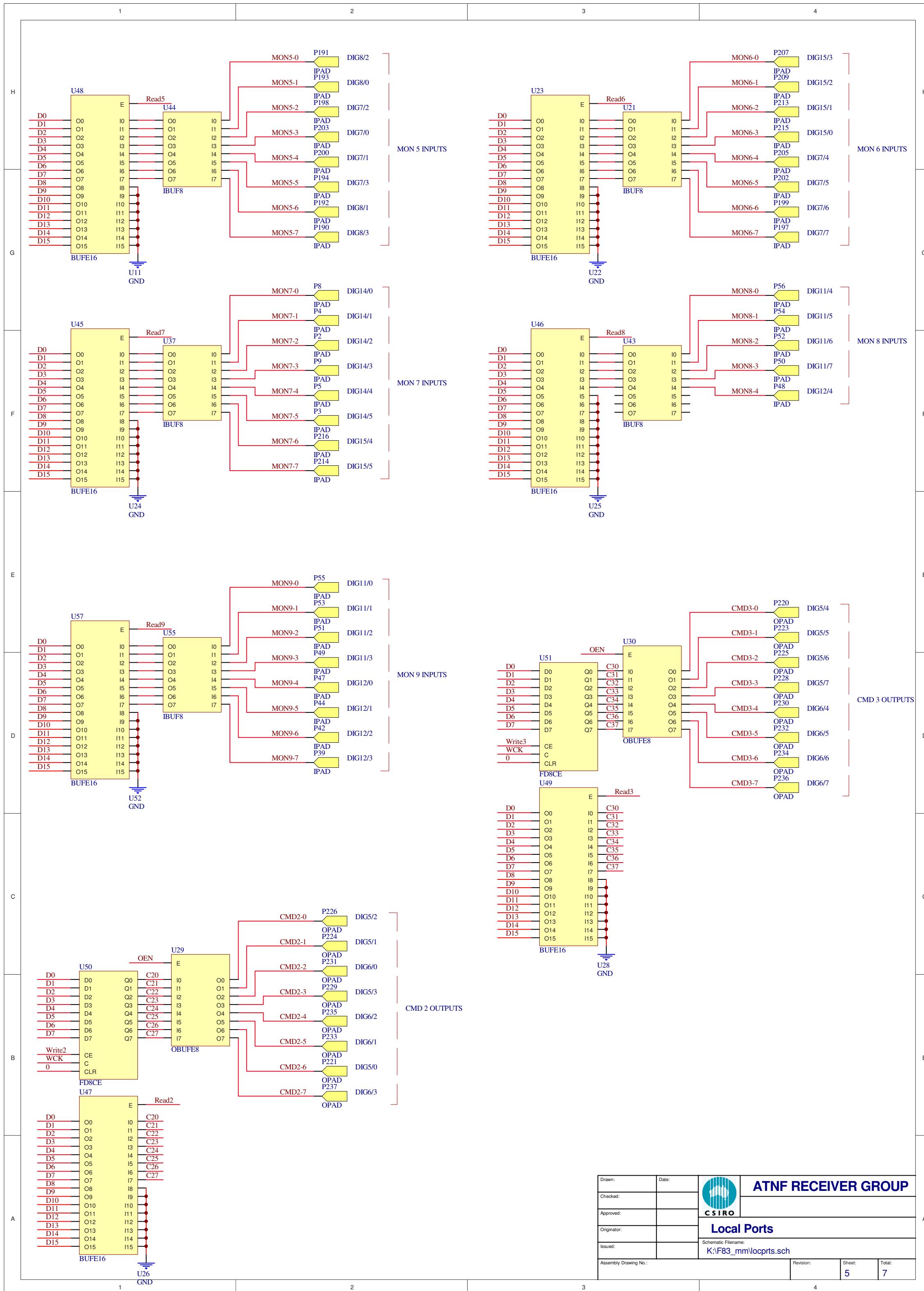

Figure B.12: F83 interface Xilinx local ports schematic

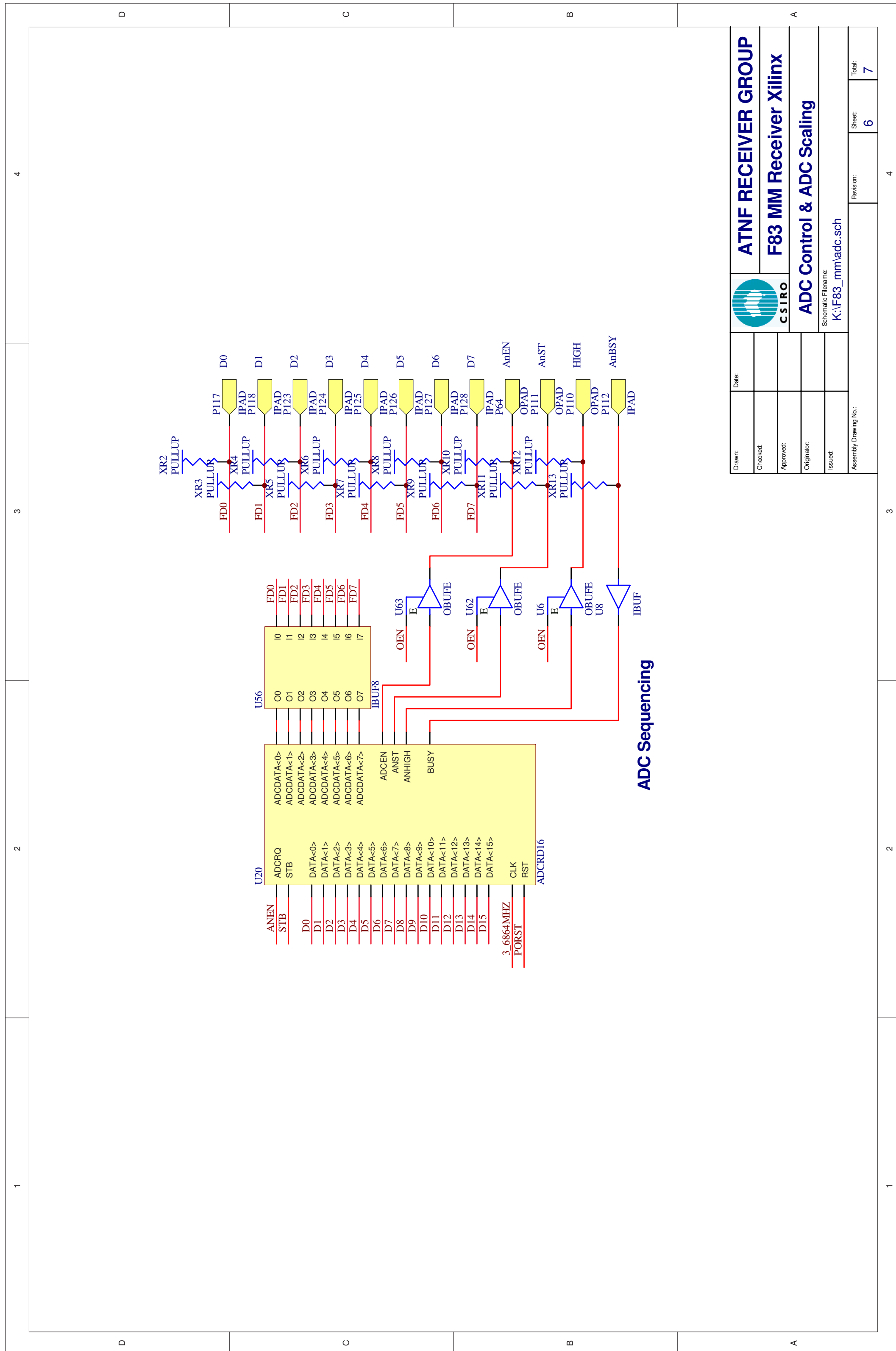

Figure B.13: F83 interface Xilinx ADC sequencing schematic

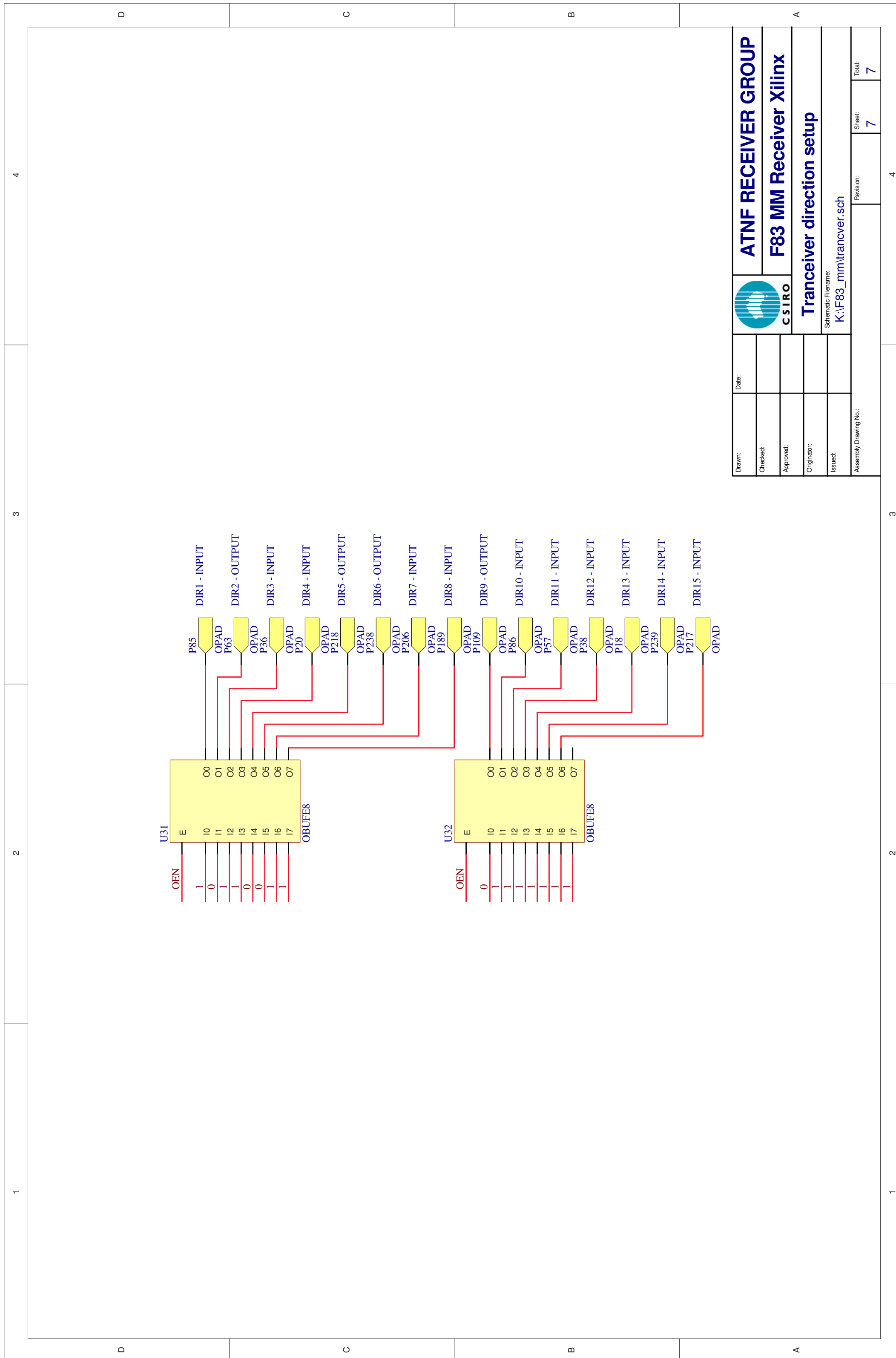

Figure B.14: F83 interface Xilinx transceiver direction schematic

Appendix C

WVR Interface Schematics

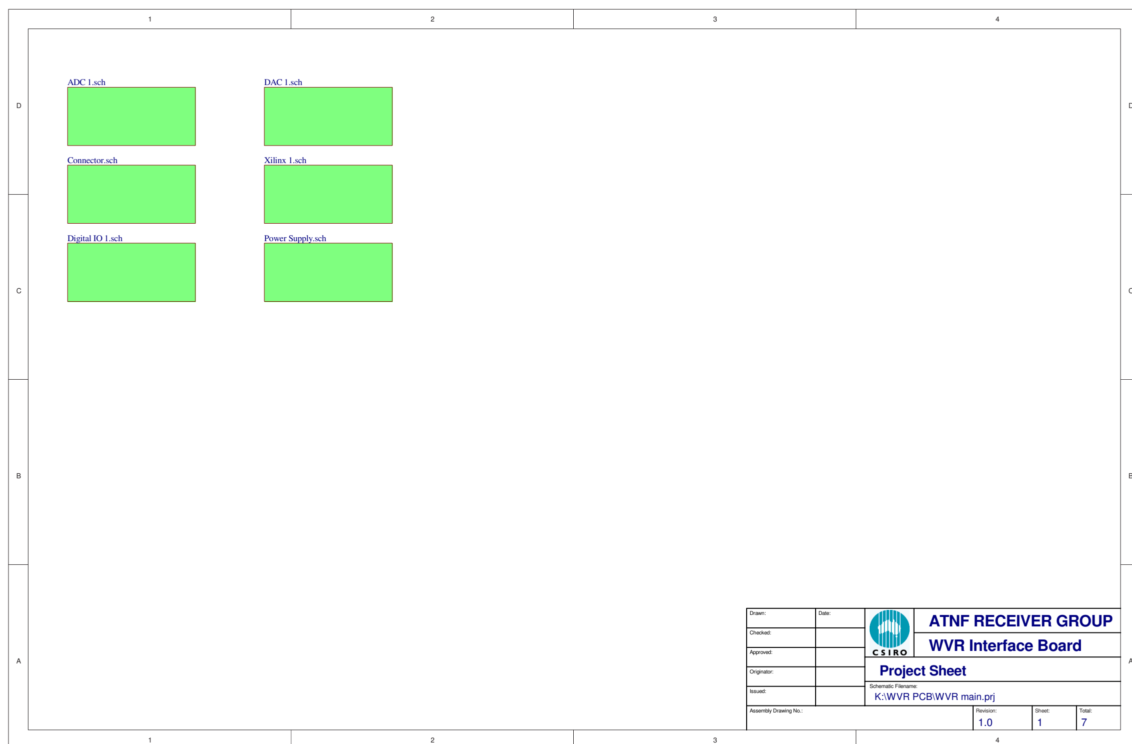

Figure C.1: WVR interface project sheet

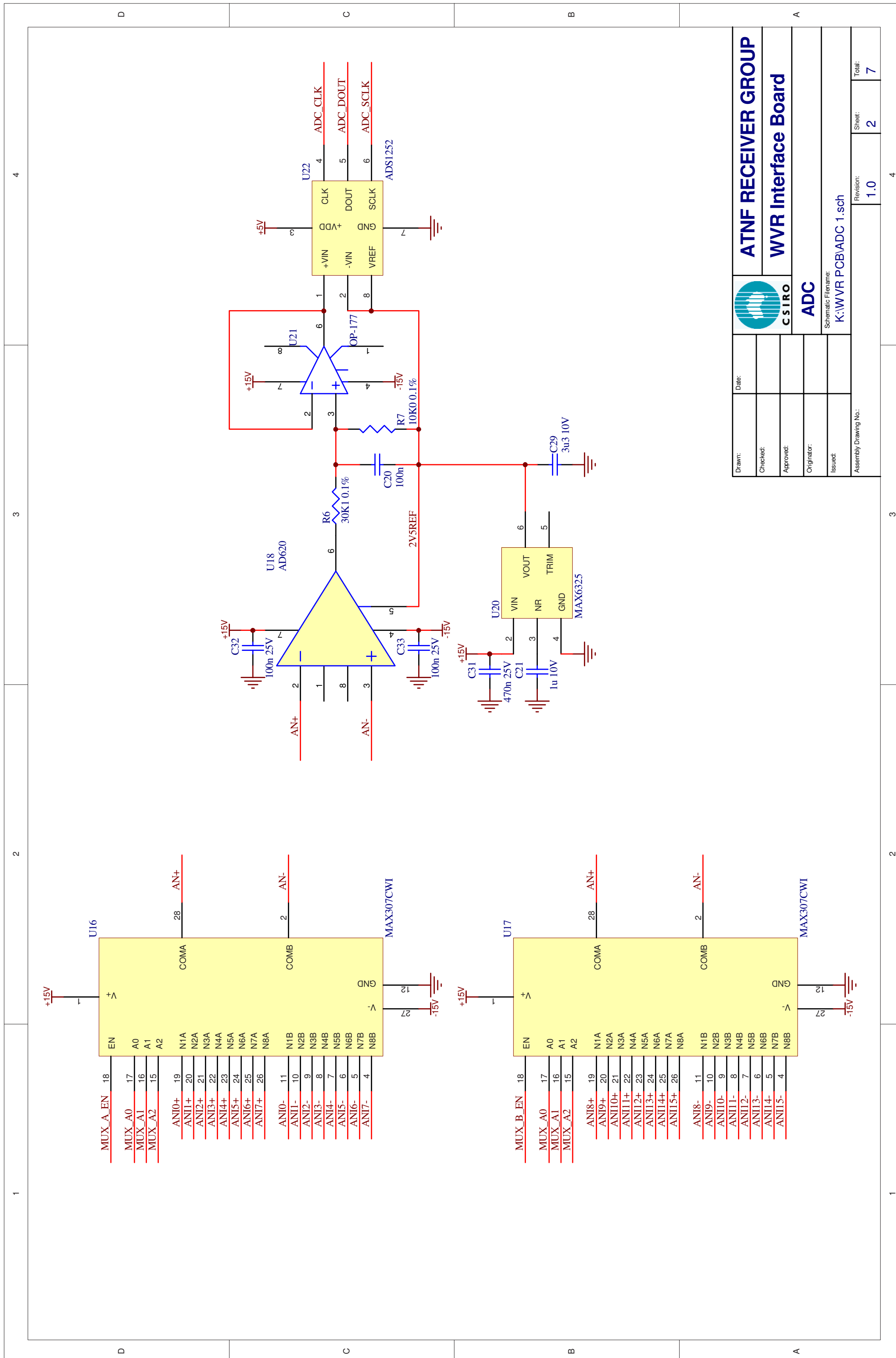

Figure C.2: WVR interface ADC schematic

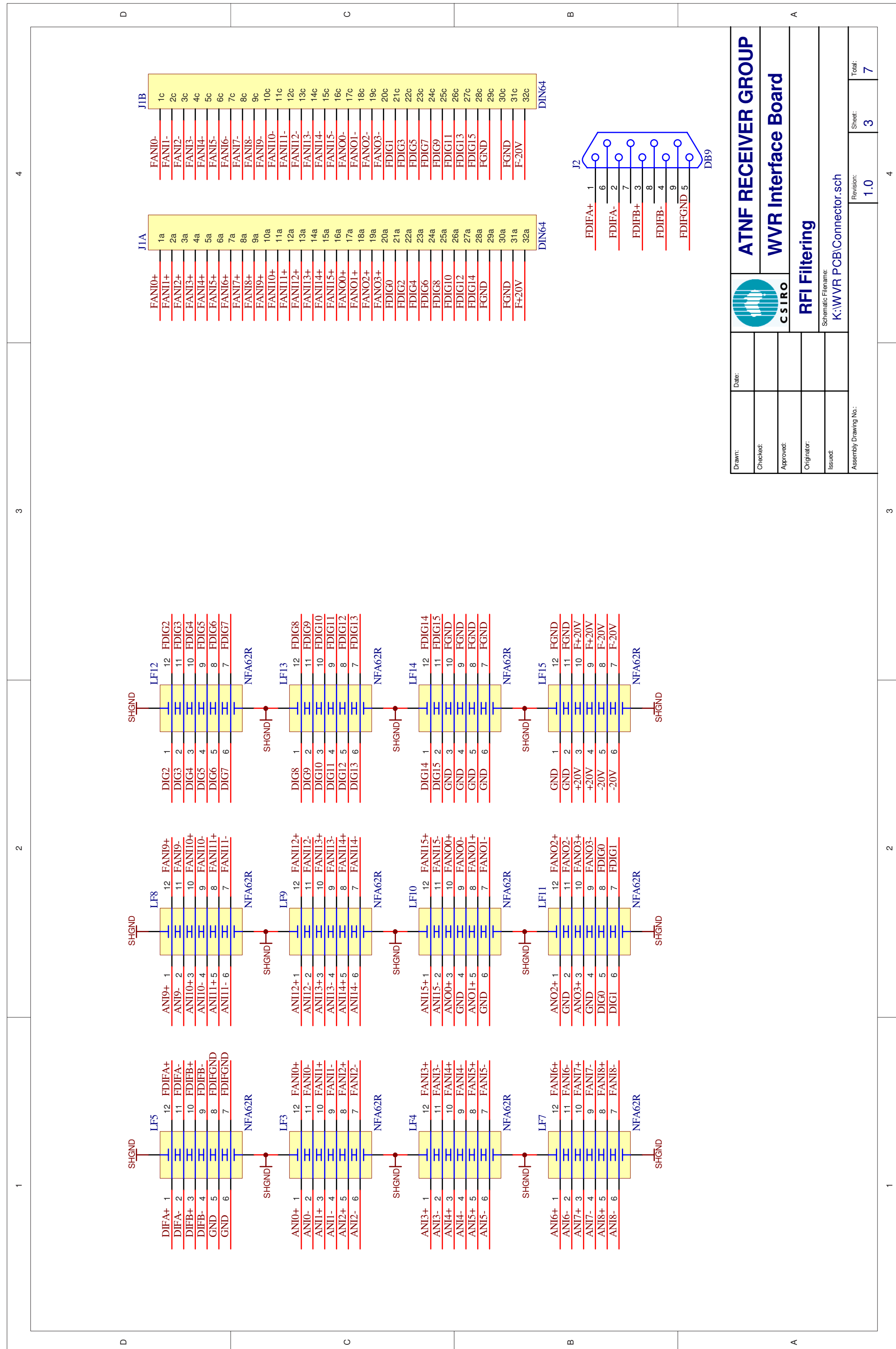

Figure C.3: WVR interface RFI filtering schematic

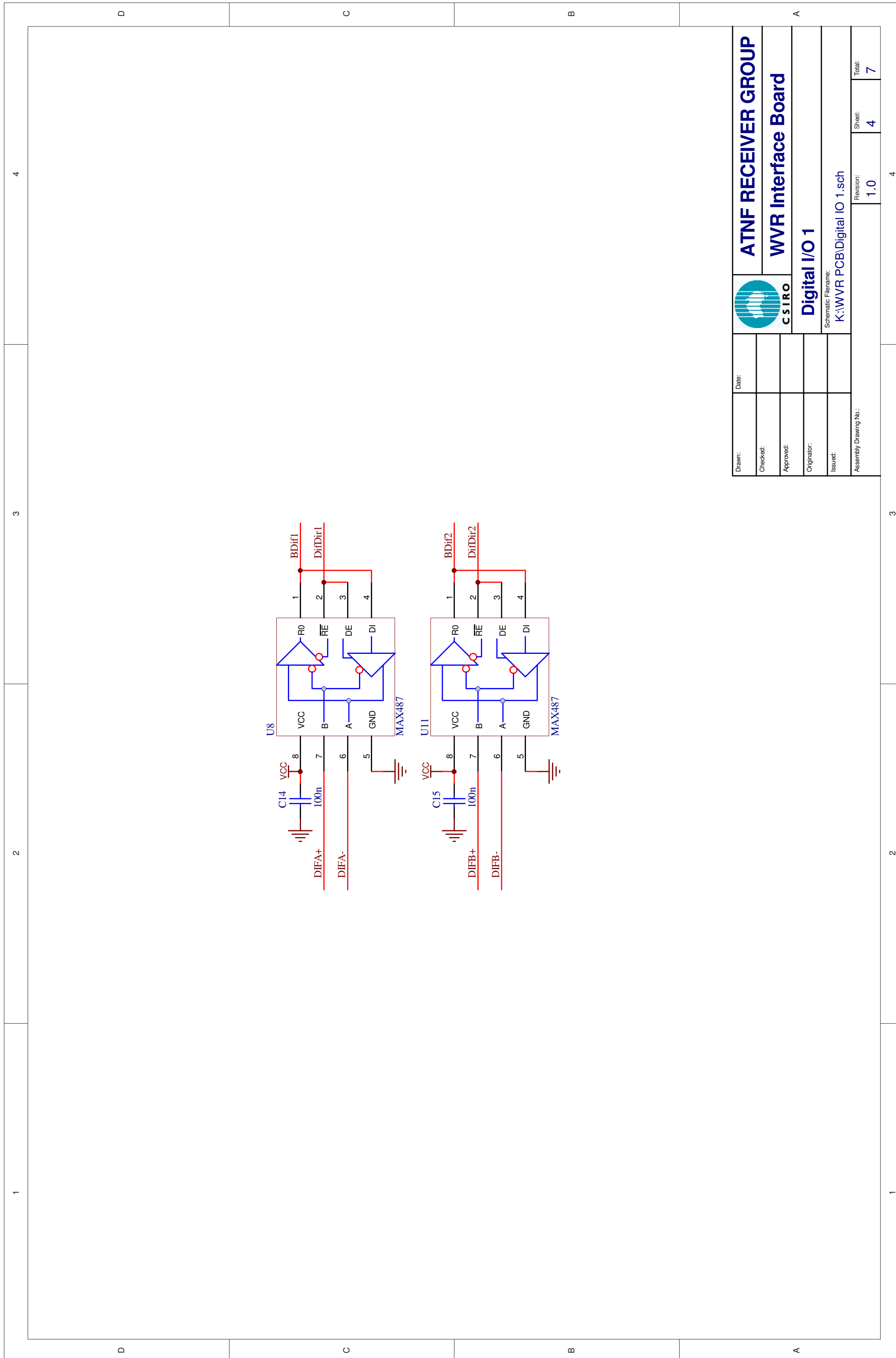

Figure C.4: WVR interface digital I/O schematic

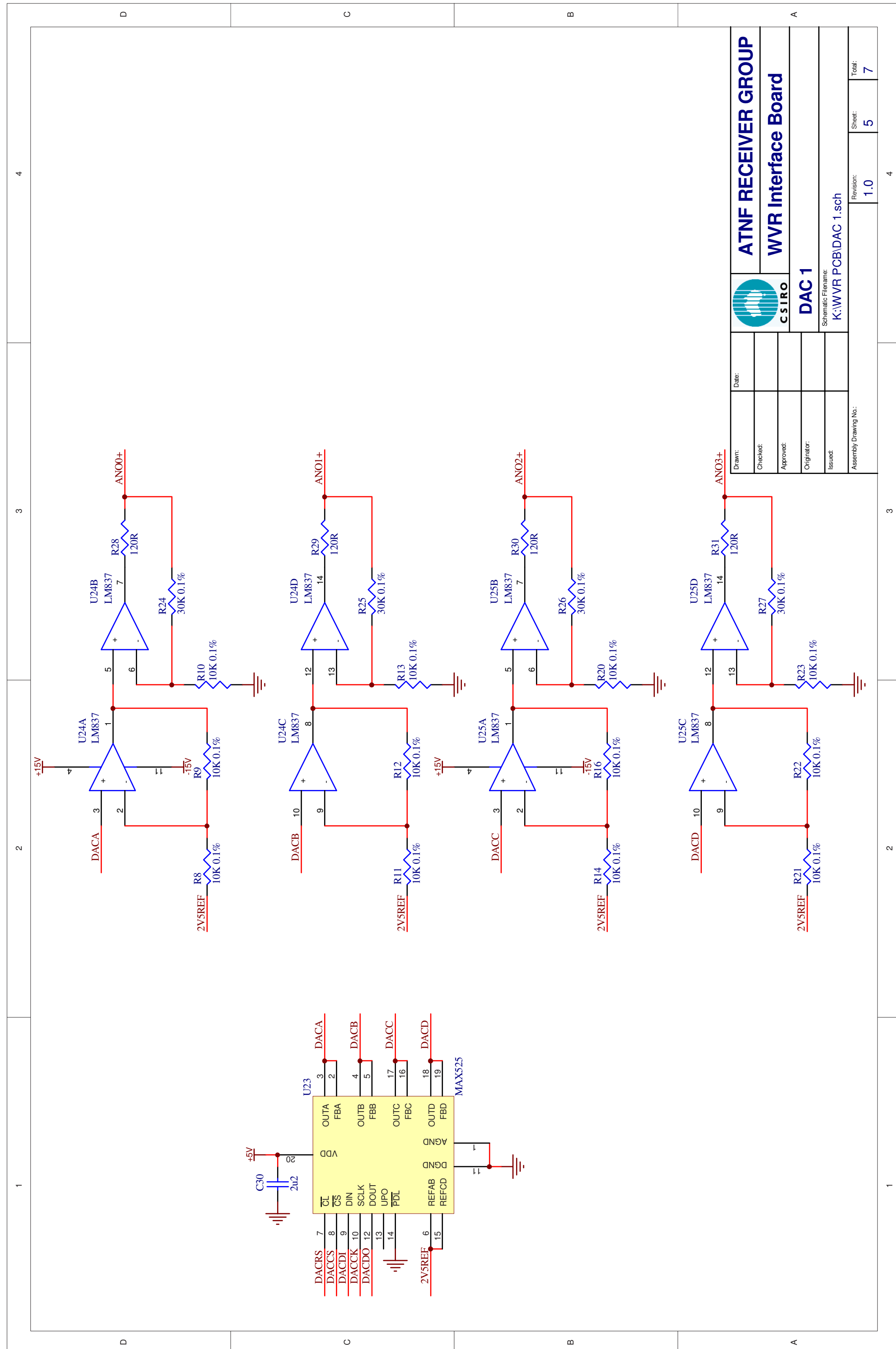

Figure C.5: WVR interface DAC schematic

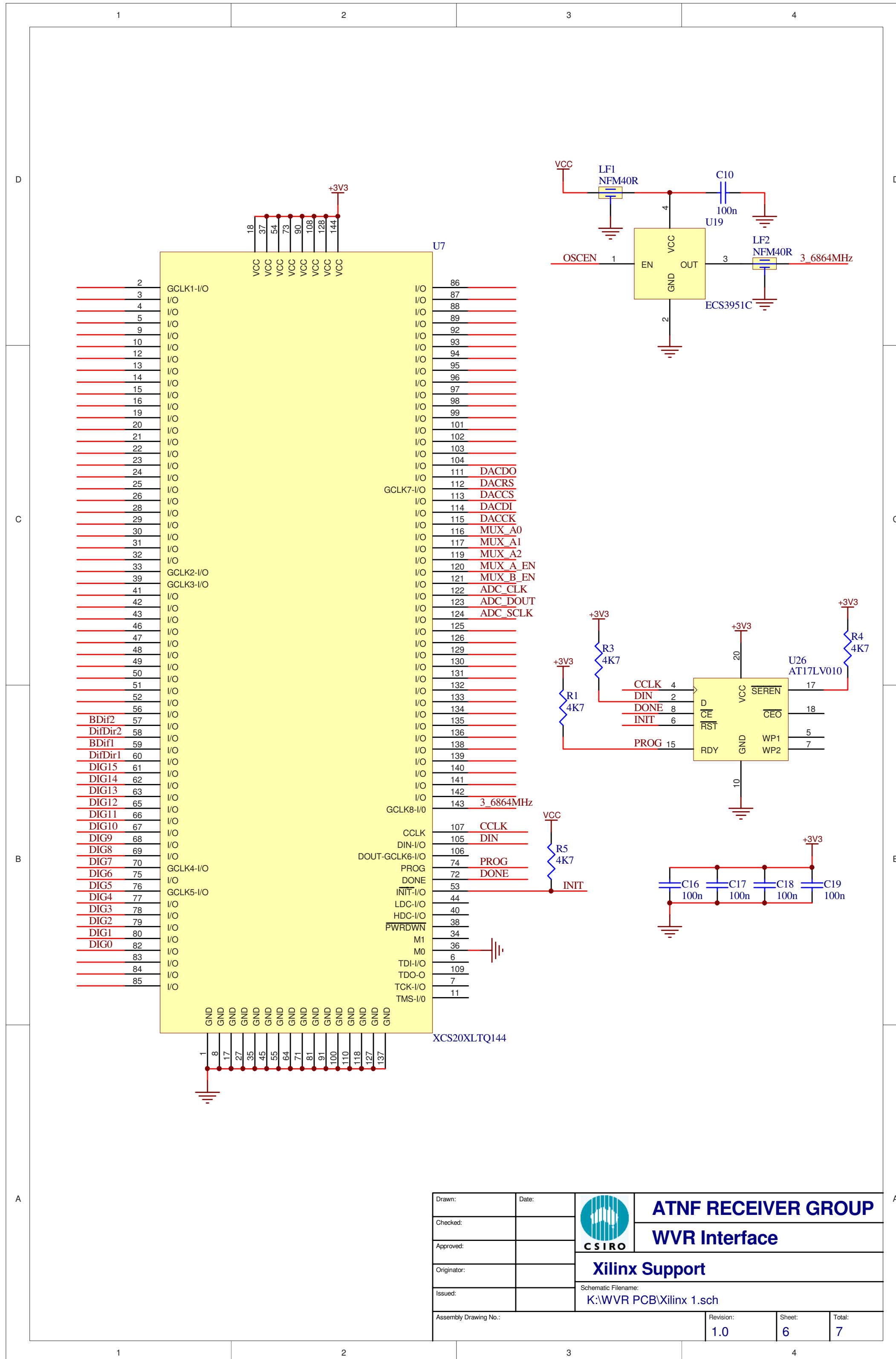

Figure C.6: WVR interface Xilinx support schematic

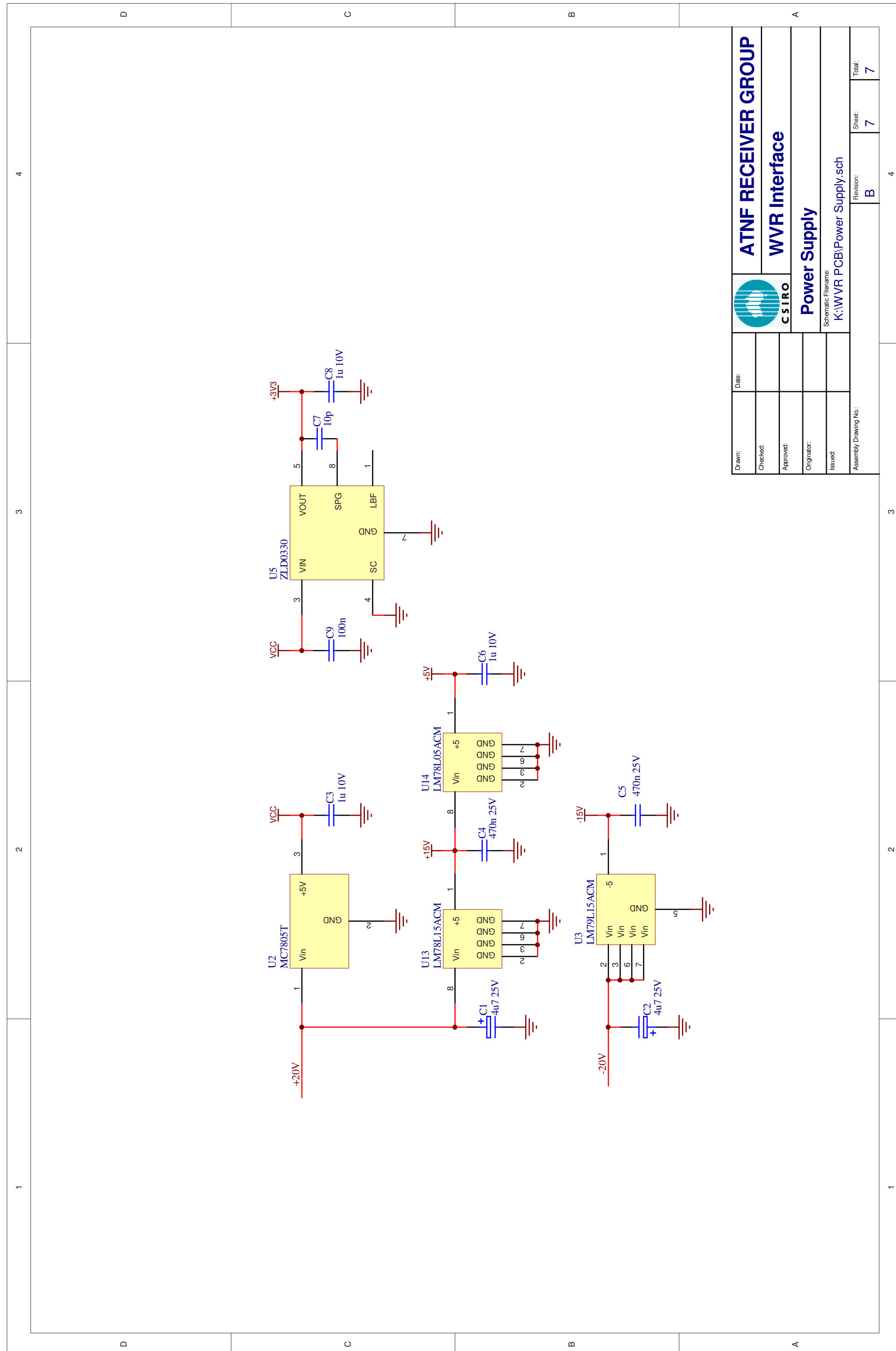

Figure C.7: WVR interface power supply schematic

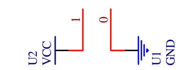

Figure C.8: WVR interface Xilinx schematic

Appendix D

Conversion Interface Schematics

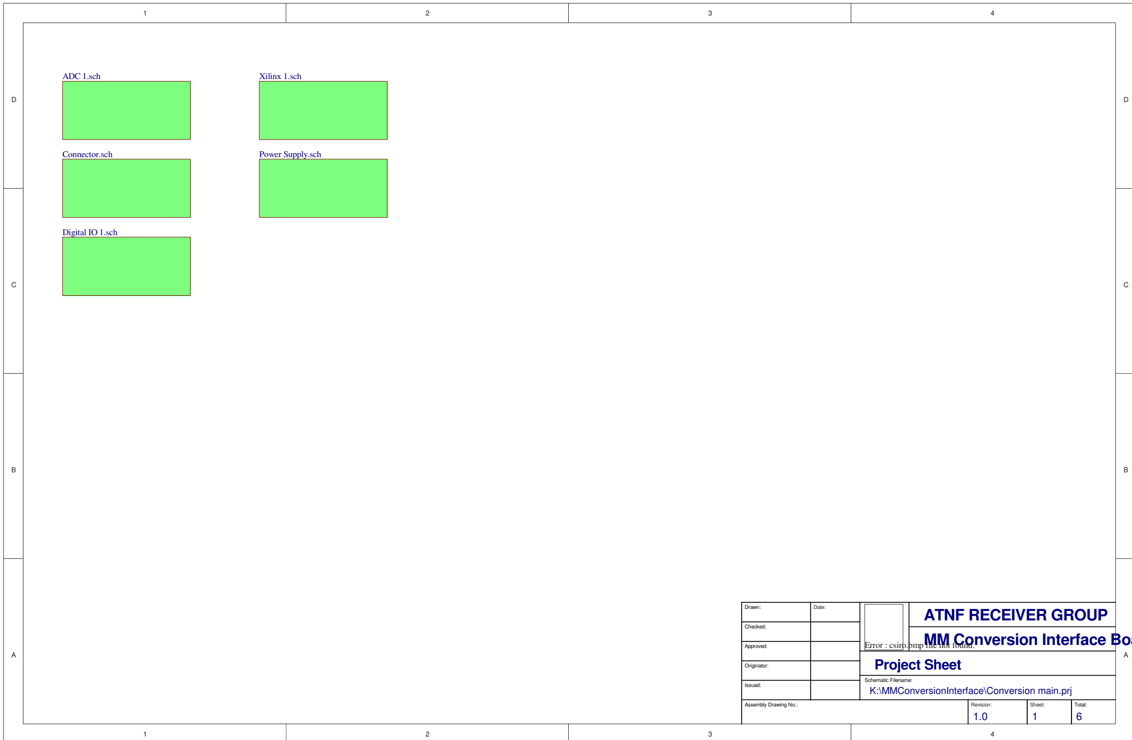

Figure D.1: MM conversion interface project sheet

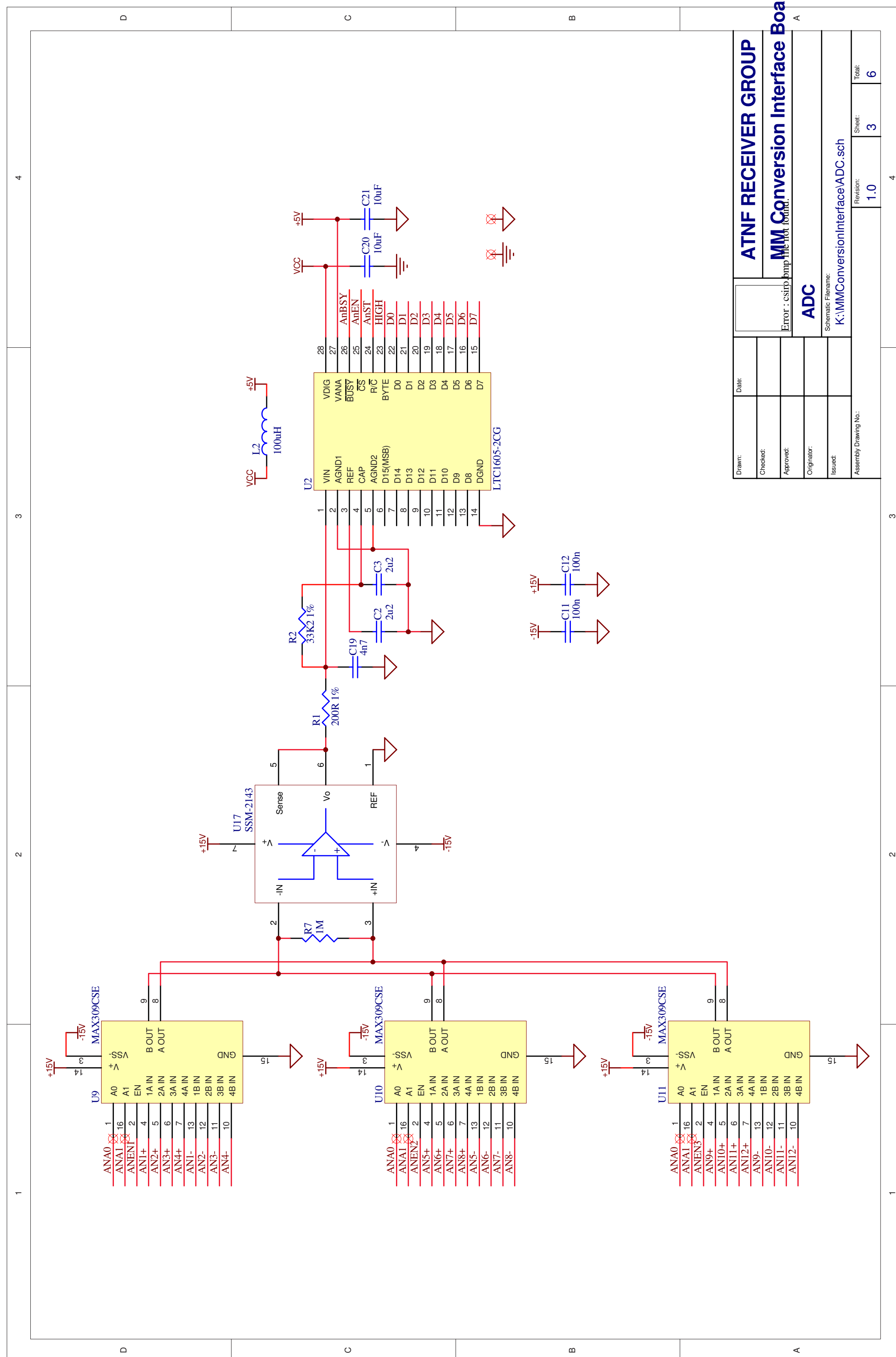

Figure D.2: MM conversion interface ADC schematic

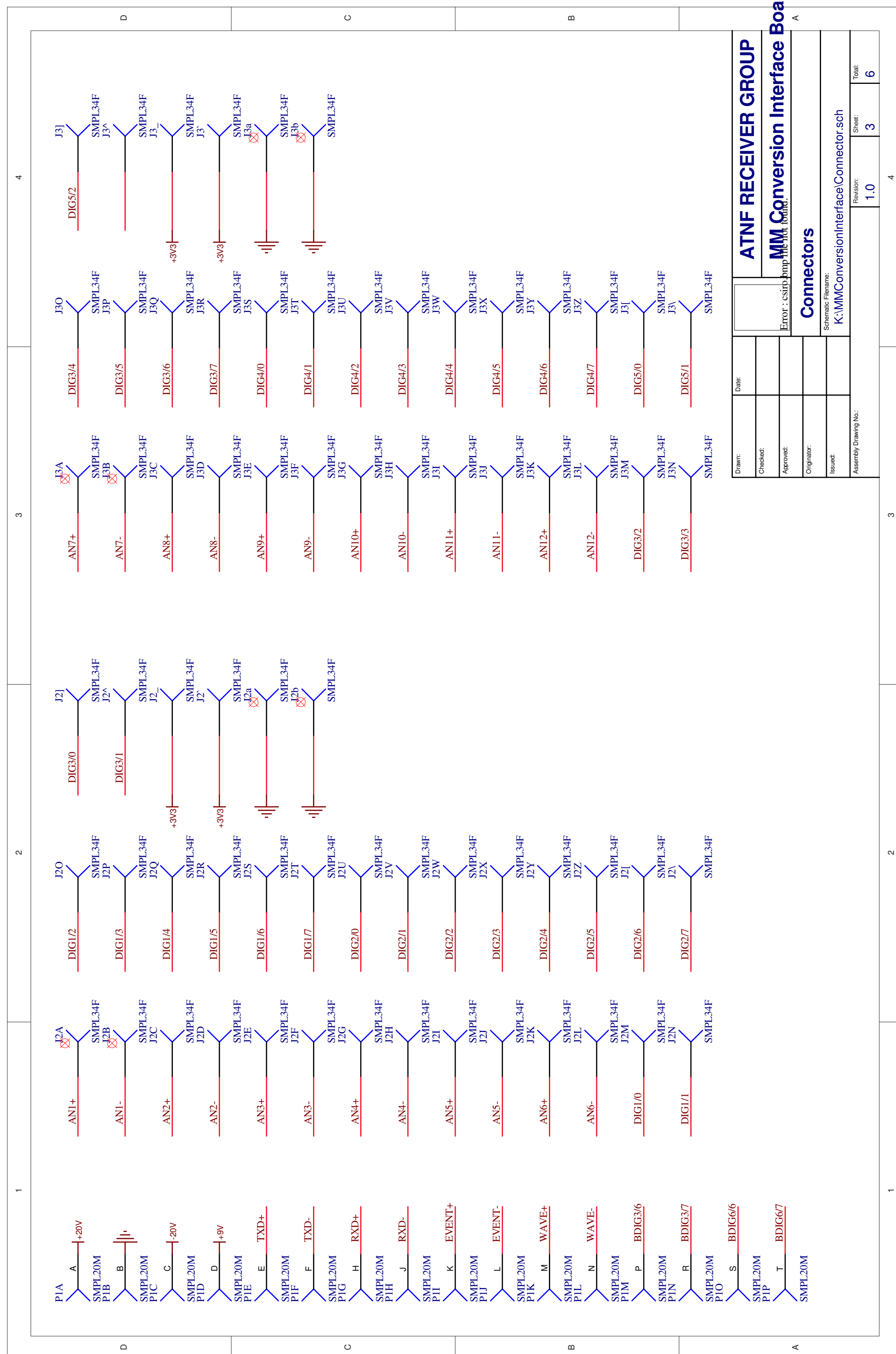

Figure D.3: MM conversion interface connector schematic

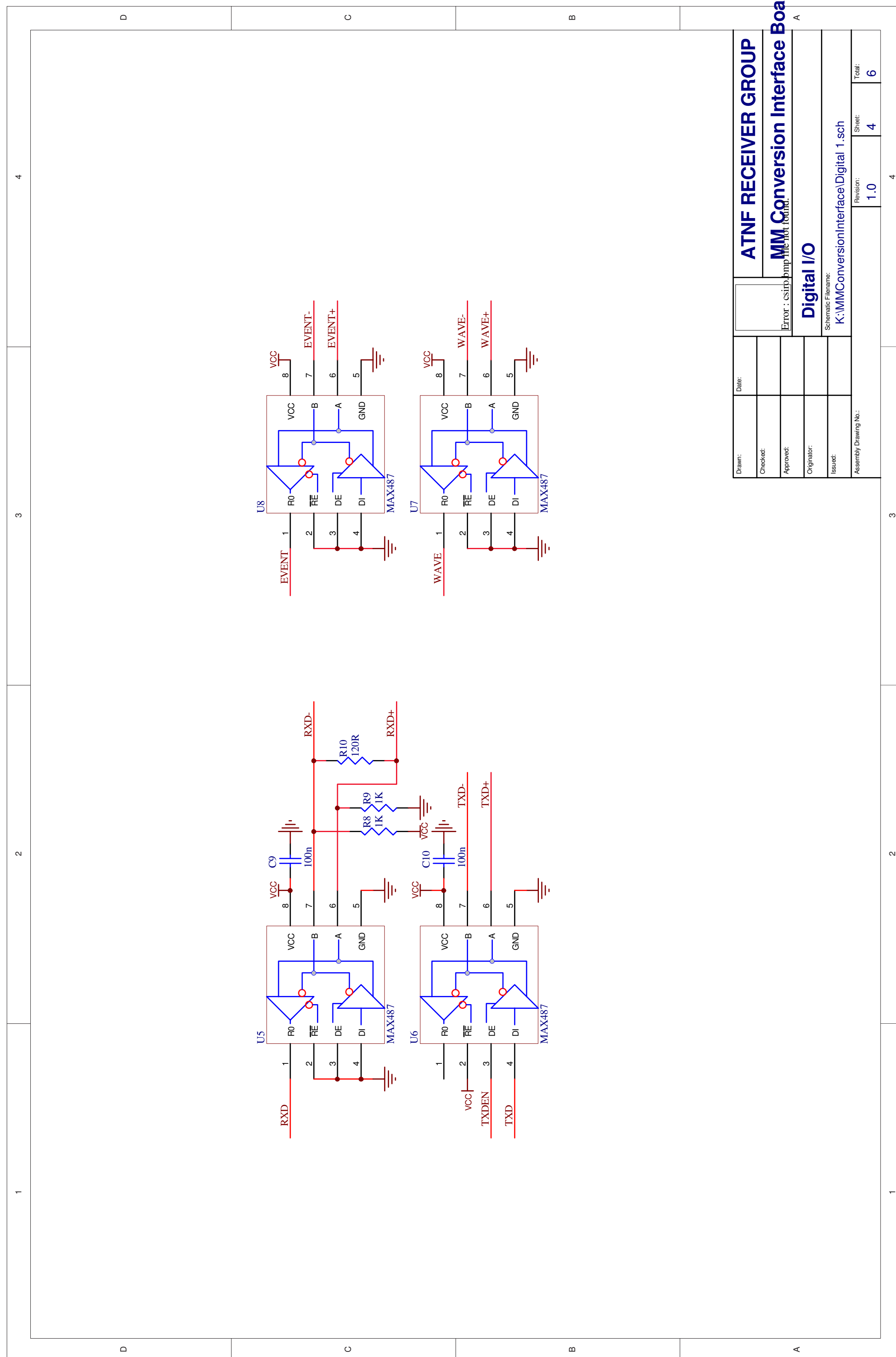

Figure D.4: MM conversion interface digital I/O schematic

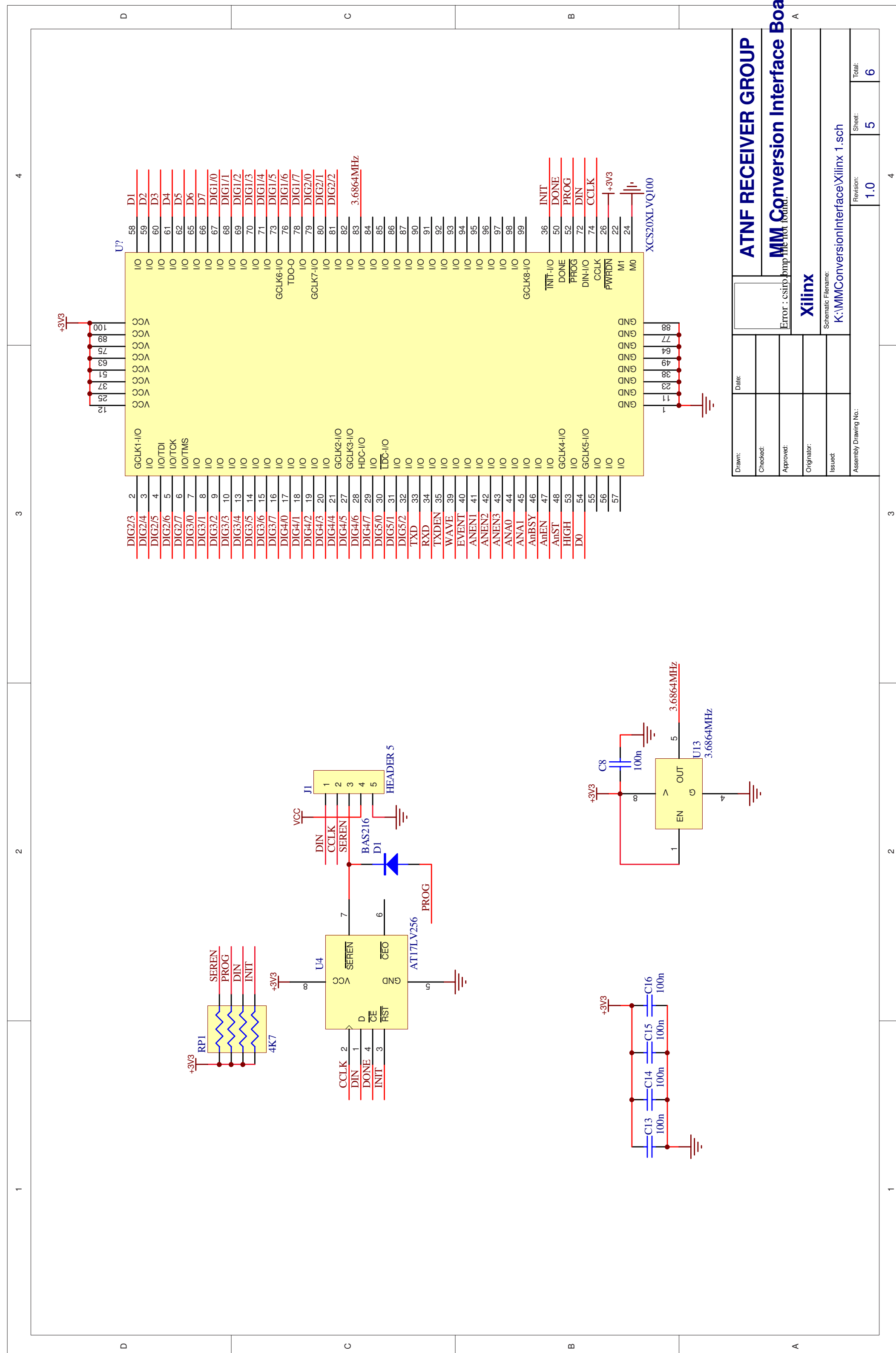

Figure D.5: MM conversion interface Xilinx schematic

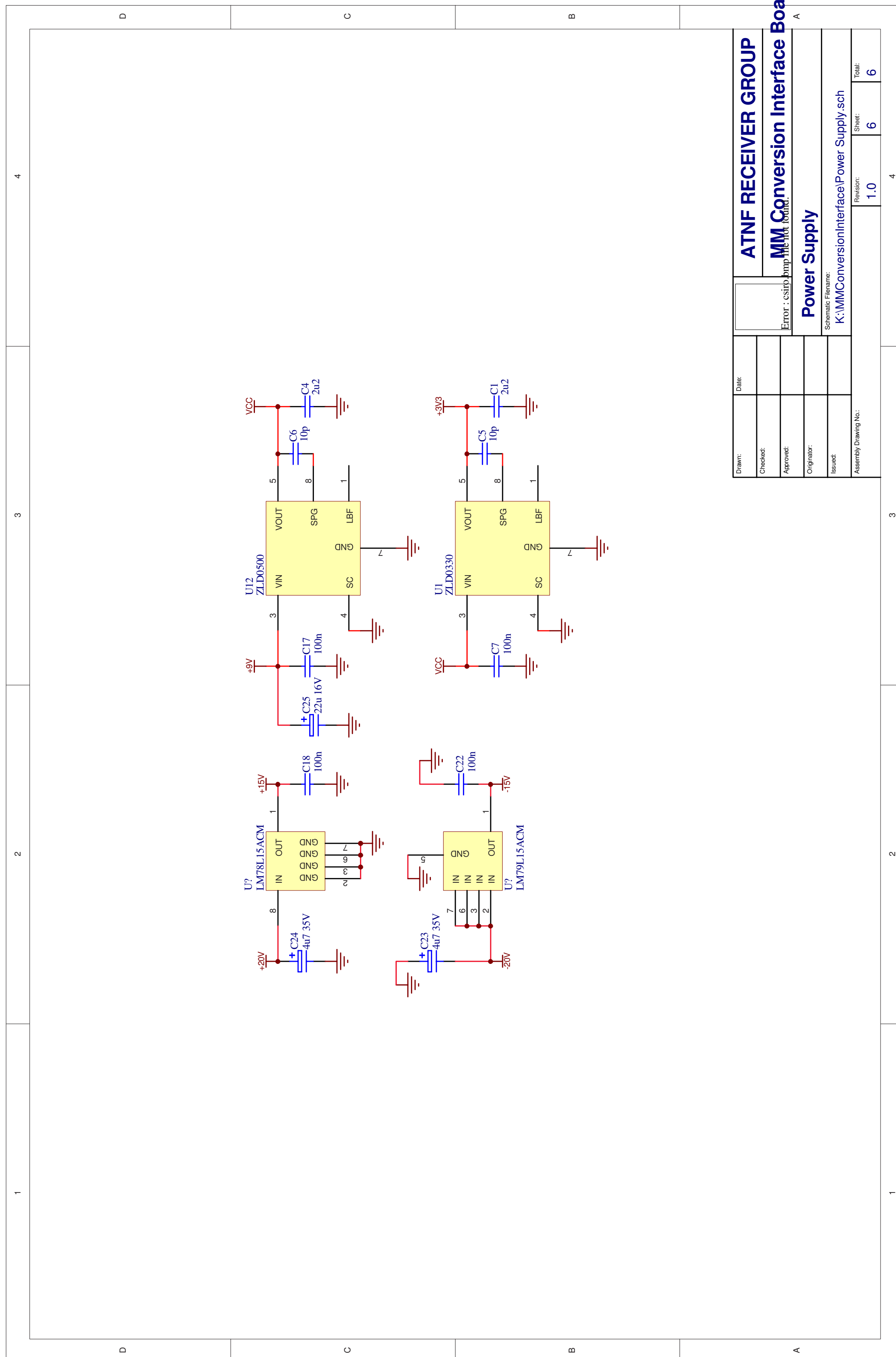

Figure D.6: MM conversion interface power supply schematic

Appendix E

LO Interface Schematics

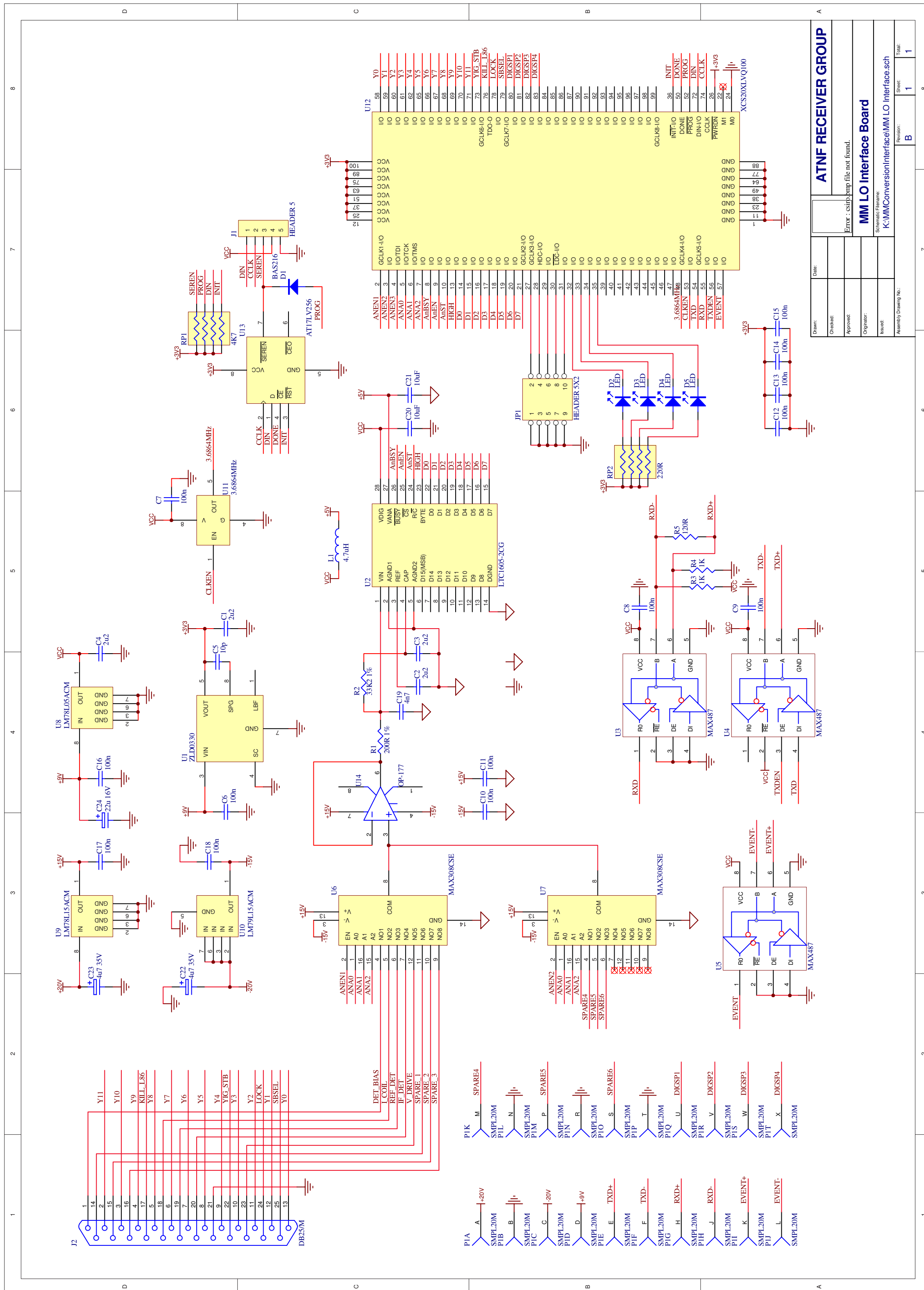

Figure E.1: MM local oscillator interface schematic

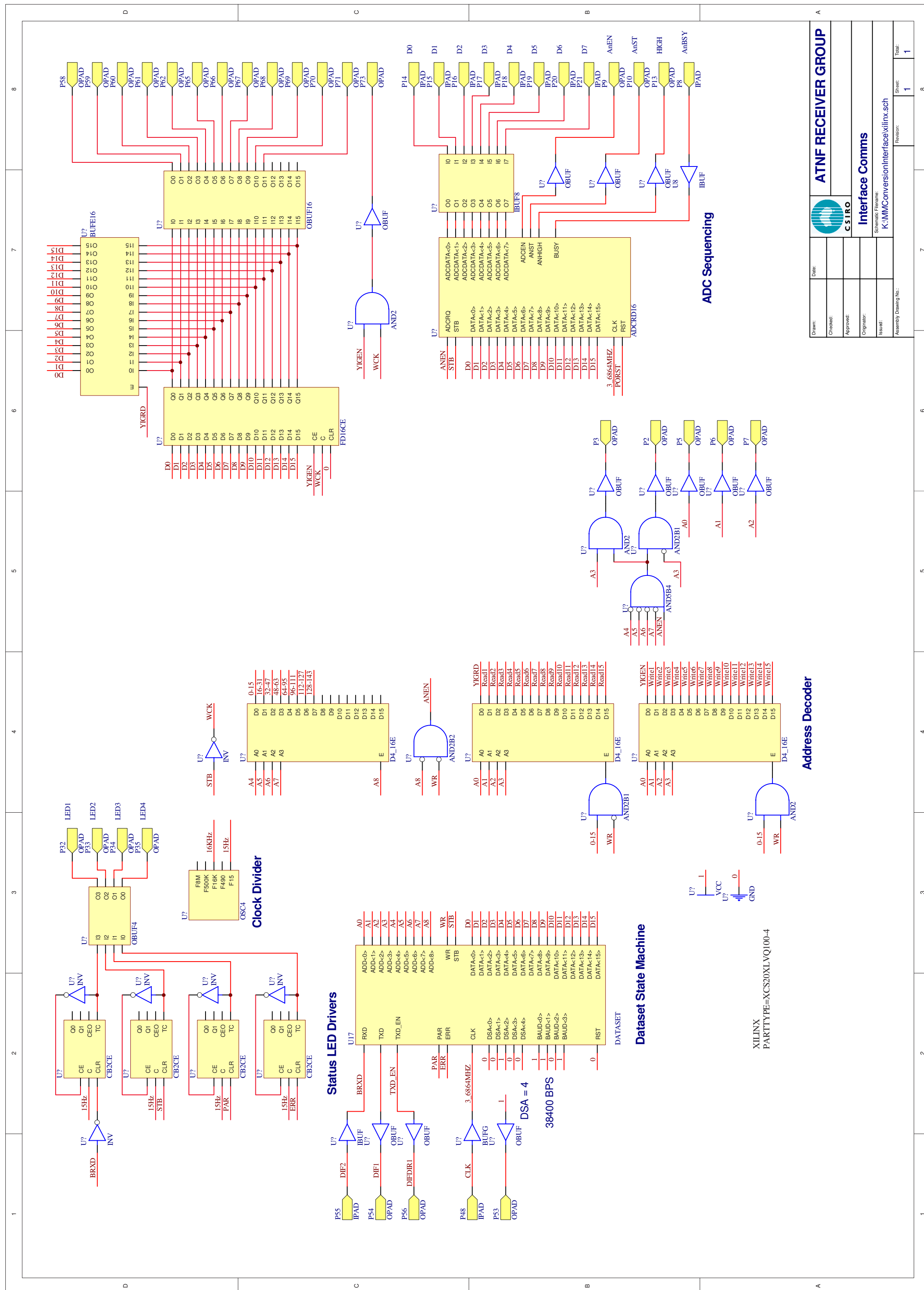

Figure E.2: MM local oscillator interface Xilinx schematic

Appendix F

Fibre Mux Schematics

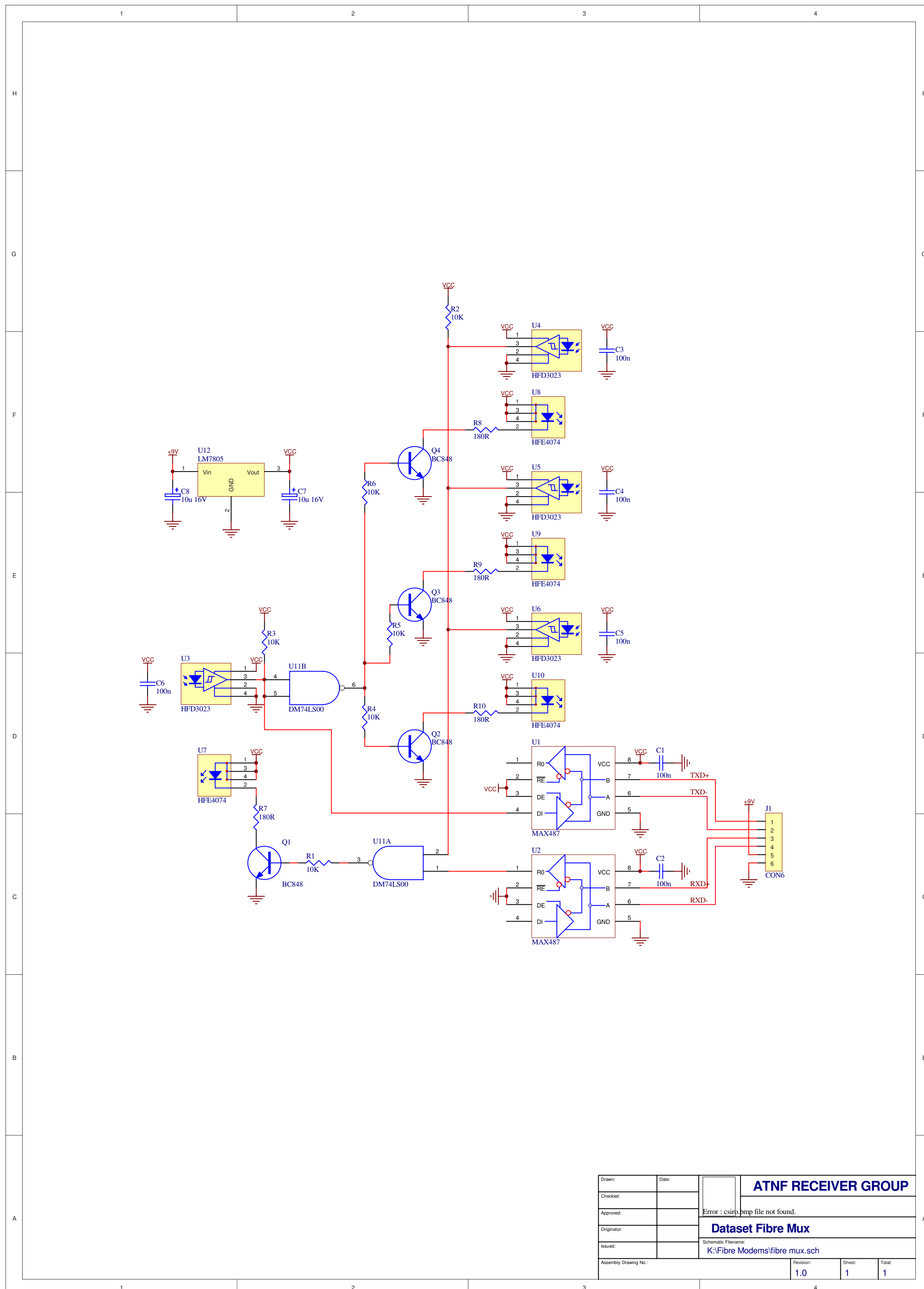

Figure F.1: MM fibre mux schematic

Appendix G

Labwindows/CVI Source Code

G.1 Dataset Test Panel

G.1.1 dataset serv.c

```
/*************************************************************************/
/*                                                                       */
/*  Labwindows Dataset Bus TCP/IP Server.                                */
/*                                                                       */
/*  Revision 0.1 by Suzy Jackson <sjackson@atnf.csiro.au>                */
/*                                                                       */
/*  Allows computers to access a dataset bus using TCP/IP.               */
/*                                                                       */
/*  Protocol is:  command dataset.address [data]                         */
/*      command is either "set" or "show"                                */
/*          dataset is an ascii string containing the dataset name       */
/*          (eg "c13_ds")                                                */
/*          address is an ascii integer containing the register address  */
/*          data is an ascii integer containing data to be loaded.       */
/*                                                                       */
/*  The server responds with either                                      */
/*      <OK> command dataset.address [data]                              */
/*  or                                                                   */
/*      <ERR> command dataset address [data] returned error              */
/*          where error is the error number returned by the relevant     */
/*          dataset comms routine.                                       */
/*                                                                       */
/*************************************************************************/

#include <rs232.h>
#include <utility.h>
#include <ansi_c.h>
#include <cvirte.h>     /* Needed if linking in external compiler */
#include <userint.h>
#include <tcpsupp.h>
#include "dset_96.h"    /* Dataset serial comms routines */
#include "dataset_server.h"
```

```
#define CONTROL 0xC0000000
#define MONITOR 0x40000000

#define ESC     0x1b
#define SYN     0x16
#define NAK     0x15
#define ACK     0x06
#define BEL     0x07

#define PARITY 1    /* odd  */
#define DATA_BITS 8
#define STOP_BITS 1
#define IN_QUEUE_SIZE 64
#define OUT_QUEUE_SIZE 64

#define MSGHD 71
#define CONTROL 0xC0000000
#define MONITOR 0x40000000
#define ACK     0x06


struct loboss_msg
  {
    char msgHD;
    unsigned char msglen;
    char msg[256];
  };

int port, port_open, panelHandle, TCPHandle;

void int_to_bytes(int num, char *byte)
{
  short i;
  for (i=0;i<=3;i++)
  {
    byte[i]=num;
    num=(num >> 8);
  }
}


unsigned int bytes_to_int(char *byte, int num_bytes)
{
  short i;
  unsigned int num = byte[0];
  unsigned char ch;
  for (i=1;i<num_bytes;i++)
  {
    num=(num << 8);
    ch = byte[i];
    num += ch;
```

```
  }
  return(num);
}

int Initialise_Dataset(int ds_address, int port, int baud)
{
/* Error codes:             0   success
                            -1  comms error
*/
  int data;
  extern int port_open;
  if (!port_open)
  {
    if (OpenComConfig (port, "", baud, PARITY, DATA_BITS,
                       STOP_BITS, IN_QUEUE_SIZE, OUT_QUEUE_SIZE))
      return(-1);
      port_open=1;
  }
  FlushOutQ (port);
  FlushInQ (port);
  SetComTime (port, 0.5);
  return(0);
}

void Close_All_Datasets(port)
{
  CloseCom(port);
}

int ReadResponse(int* response)     /*  TESTED OK  */
{
  extern int port;
  char read_buff[3], flag, txferstring[40];
  ComRd (port, &flag, 1);
  if (rs232err)
  {
    SetCtrlVal (panelHandle, PANEL_HISTORY, GetRS232ErrorString (rs232err));
    SetCtrlVal (panelHandle, PANEL_HISTORY, " ");
    return(-1);
  }
  sprintf (txferstring, "%X ", flag&0xff);
  SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
  ComRd (port, read_buff, 2);
  sprintf (txferstring, "%X %X ", read_buff[0]&0xff, read_buff[1]&0xff);
  SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
  if (flag == ACK || flag == BEL)
  {
    if (read_buff[0] == ESC)
    {
      Decode(read_buff);
      ComRd(port, &read_buff[1],1);
      sprintf (txferstring, "%X ", read_buff[1]&0xff);
```

```
      SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
    }
    if (read_buff[1] == ESC)
    {
      ComRd(port, &read_buff[2], 1);
      sprintf (txferstring, "%X ", read_buff[2]&0xff);
      SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
      Decode(&read_buff[1]);
    }
    *response = bytes_to_int(read_buff, 2);
    while (GetInQLen (port) >0) {
      ComRd (port, read_buff, 1);
      sprintf (txferstring, "%X ", read_buff[0]&0xff);
      SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
    }
    return(0);
  }
  else {
    while (GetInQLen (port) >0) {
      ComRd (port, read_buff, 1);
      sprintf (txferstring, "%X ", read_buff[0]);
      SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
    }
    return (-2);
  }
}

void Decode(char* esc_seq)  /*  TESTED OK  */
{
  switch (esc_seq[1])
  {
    case '0':
      esc_seq[0] = ESC;
      break;
    case '1':
      esc_seq[0] = SYN;
      break;
    case '2':
      esc_seq[0] = ACK;
      break;
    case '3':
      esc_seq[0] = BEL;
      break;
    case '4':
      esc_seq[0] = NAK;
      break;
    default:
      break;
  }
}

int SendMessage(int message)    /* TESTED OK */
```

```
{
  extern int port;
  char write_buff[10], byte[4], current;
  int index = 0, count = 3;
  write_buff[index++] = SYN;
  int_to_bytes(message, byte);
  while(index < 10)
  {
    if (count >= 0)
    {
      switch (current = byte[count--])
      {
        case ESC:
          write_buff[index++] = ESC;
          write_buff[index++] = '0';
          break;
        case SYN:
          write_buff[index++] = ESC;
          write_buff[index++] = '1';
          break;
        default:
          write_buff[index++] = current;
          break;
      }
    }
    else
      write_buff[index++] = NULL;
  }
  if (ComWrt(port, write_buff, 10) != 10) return(-1);
  else return(0);
}

int Dataset_Out(int ds_address, int control_point, int data_out)
{

/* Error codes:            0   success
                           -1  comms error
                           -2  dataset error
                           -3  invalid control point
                           -4  invalid data out
*/
  int message, err;
  char reply[3], txferstring[40];
  if (control_point < 0 || control_point > 511) return(-3);
  if (data_out < 0 || data_out > 0xffff) return(-4);
  message = CONTROL + (ds_address << 25) + (control_point << 16)
            + data_out;
  if (err = SendMessage(message)) return(err);
  ComRd(port, reply, 3);
  if (rs232err)
  {
    SetCtrlVal (panelHandle, PANEL_HISTORY,
```

```
                GetRS232ErrorString (rs232err));
    SetCtrlVal (panelHandle, PANEL_HISTORY, " ");
  }
  else {
    sprintf (txferstring, "%X %X %X ", reply[0]&0xff, reply[1]&0xff,
             reply[2]&0xff);
    SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
  }
  while (GetInQLen (port) >0) {
    ComRd (port, reply, 1);
    sprintf (txferstring, "%X ", reply[0]&0xff);
    SetCtrlVal (panelHandle, PANEL_HISTORY, txferstring);
  }
  if (reply[0] != ACK) return(-2);
  else return(0);
}

int Dataset_In(int ds_address, int monitor_point, int *data_in)
{

/* Error codes:             0   success
                            -1  comms error
                            -2  dataset error
                            -3  invalid control point
*/
  int message, err, response;
  if (monitor_point < 0 || monitor_point > 511) return(-3);
  message = MONITOR + (ds_address << 25) + (monitor_point << 16);
  if (err = SendMessage(message)) return(err);
  if (err = ReadResponse(&response)) return(err);
  *data_in = (response & 0xffff);
  return(0);
}


int dsetlookup (char *dataset)

/* Provides a quick conversion from dataset name to dataset number. */

{
  if (!strcmp (dataset, "c11_ds")) return 10;
  if (!strcmp (dataset, "c13_ds")) return 16;
  if (!strcmp (dataset, "c31_ds")) return 17;
  if (!strcmp (dataset, "c41_ds")) return 18;
  /* we don't understand the name, so let's assume it's a decimal */
  return atoi(dataset);
}


int CVICALLBACK TCPServerCallback (unsigned handle, int event, int error,
                                   void *callbackData)
```

```
{
  char command [12], dataset [12], reqstring [256];
  char *temp;
  int err = 0, dataSize = 256, address, data;
  struct loboss_msg buffer;
  TCPHandle = handle;
  switch (event) {
    case TCP_CONNECT:
      TCPHandle = handle;
      SetCtrlVal (panelHandle, PANEL_HISTORY,
                  "New connection established\n");
      break;
    case TCP_DATAREADY:
      if ((dataSize = ServerTCPRead(TCPHandle, &buffer, dataSize, 1000))
          == -kTCP_ConnectionClosed) {
        SetCtrlVal (panelHandle, PANEL_HISTORY,
                    "TCP Read Error - connection closed\n");
      }
      if (buffer.msglen>2) {
        /* extract the message from the buffer */
        memcpy (reqstring, &buffer.msg, buffer.msglen-2);
        /* append a NULL so that it looks like a string */
        reqstring [buffer.msglen-2]=0;
        /* remove the . between the dataset name and register address */
        if (temp=strchr(reqstring,'.')) *temp = ' ';
        /* suck out the gist of the message */
        sscanf (reqstring,"%s %s %d %d", command, dataset, &address,
                &data);
        if (!strcmp (command,"show")) {
          if (err = Dataset_In (dsetlookup (dataset), address, &data)) {
            sprintf (buffer.msg, "<ERR> show %s.%d returned %d\n",
                     dataset, address, err);
          }
          else {
            sprintf (buffer.msg, "<OK> show %s.%d %d", dataset, address,
                     data);
          }
        }
        else if (!strcmp (command, "set")) {
          if (err = Dataset_Out (dsetlookup (dataset), address, data)) {
            sprintf (buffer.msg, "<ERR> set %s.%d %d returned %d",
                     dataset, address, data, err);
          }
          else {
            sprintf (buffer.msg, "<OK> set %s.%d %d", dataset, address,
                     data);
          }
        }
        else {
          sprintf (buffer.msg, "<ERR> %s is not a valid message",
                   reqstring);
          err = -1;
```

```
        }
        buffer.msglen = strlen (buffer.msg) + 2;
        buffer.msgHD = MSGHD;
        SetCtrlVal (panelHandle, PANEL_HISTORY, &buffer.msg);
        SetCtrlVal (panelHandle, PANEL_HISTORY, "\n");
        if (ServerTCPWrite (TCPHandle, &buffer, buffer.msglen, 1000)<0) {
          SetCtrlVal (panelHandle, PANEL_HISTORY,
                      "TCP Write Error - connection closed\n");
        }
      }
      sprintf (buffer.msg, "<done> %d",err);
      buffer.msglen = strlen (buffer.msg) + 2;
      buffer.msgHD = MSGHD;
      if (ServerTCPWrite (TCPHandle, &buffer, buffer.msglen, 1000)<0) {
        SetCtrlVal (panelHandle, PANEL_HISTORY,
                    "TCP Write Error - connection closed\n");
      }
      break;
    case TCP_DISCONNECT:
      SetCtrlVal (panelHandle, PANEL_HISTORY,
                  "Connection closed by client\n");
      break;
  }
  return (0);
}


int main (int argc, char *argv[])
{
  char miscstring [11];
  int errval;
  if (InitCVIRTE (0, argv, 0) == 0)
    return -1;  /* out of memory */
  if ((panelHandle = LoadPanel (0, "dataset_server.uir", PANEL)) < 0)
    return -1;
  DisplayPanel (panelHandle);
  RunUserInterface ();
  return 0;
}


int CVICALLBACK Shutdown (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
  switch (event) {
    case EVENT_COMMIT:
      QuitUserInterface (0);
      break;
  }
  return 0;
}
```

```
void CVICALLBACK Menu (int menuBar, int menuItem, void *callbackData,
        int panel)
{
  int err;
  char errstring [80], hoststring [256], miscstring [11];
  switch (menuItem) {
    case MENU_FILE_QUIT:
      QuitUserInterface (0);
      break;
    case MENU_SETUP_COM1:
    case MENU_SETUP_COM2:
      SetMenuBarAttribute (menuBar, MENU_SETUP_COM1, ATTR_CHECKED, 0);
      SetMenuBarAttribute (menuBar, MENU_SETUP_COM2, ATTR_CHECKED, 0);
      SetMenuBarAttribute (menuBar, menuItem, ATTR_CHECKED, 1);
      if (menuItem == MENU_SETUP_COM1) port = 1;
      else port = 2;
      PromptPopup ("Enter Baudrate",
        "Please type the baudrate being used by the datasets.\n
        (generally either 4800 or 38400)",
        miscstring, 10);
      if ((err = Initialise_Dataset(0, port, atoi(miscstring)))== -1) {
        /* Note we don't care if the dataset doesn't respond */
        SetCtrlVal (panelHandle, PANEL_HISTORY,
                    "Unable to open com port.\n");
        SetMenuBarAttribute (menuBar, menuItem, ATTR_CHECKED, 0);
      }
      break;
    case MENU_SETUP_TCP:
      PromptPopup ("Enter Port Number", "Please enter the port number\n
                   0 for no TCP", miscstring, 10);
      SetWaitCursor (1);
      if (atoi(miscstring)) {
        SetCtrlVal (panelHandle, PANEL_HISTORY, "Registering server: ");
        if (RegisterTCPServer(atoi(miscstring), TCPServerCallback, NULL)<0)
        {
          SetCtrlVal (panelHandle, PANEL_HISTORY, "Failed\n");
        }
        else SetCtrlVal (panelHandle, PANEL_HISTORY, "Successful\n");
      }
      SetWaitCursor (0);
      break;
    default:
      break;
  }
}

int CVICALLBACK send (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
  int address, data, dataset, err;
  struct loboss_msg buffer;
```

```
  switch (event) {
    case EVENT_COMMIT:
      GetCtrlVal (panelHandle, PANEL_dsa, &dataset);
      GetCtrlVal (panelHandle, PANEL_fa, &address);
      GetCtrlVal (panelHandle, PANEL_data, &data);
      if (err = Dataset_Out (dataset, address, data)) {
        sprintf (buffer.msg, "<ERR> set %d.%d %d returned %d",
                 dataset, address, data, err);
      }
      else {
        sprintf (buffer.msg, "<OK> set %d.%d %d", dataset, address, data);
      }
      SetCtrlVal (panelHandle, PANEL_HISTORY, &buffer.msg[0]);
      SetCtrlVal (panelHandle, PANEL_HISTORY, "\n");
      break;
  }
  return 0;
}

int CVICALLBACK recv (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
  int address, data, dataset, err;
  struct loboss_msg buffer;
  switch (event) {
    case EVENT_COMMIT:
      GetCtrlVal (panelHandle, PANEL_dsa, &dataset);
      GetCtrlVal (panelHandle, PANEL_fa, &address);
      if (err = Dataset_In (dataset, address, &data)) {
        sprintf (buffer.msg, "<ERR> show %d.%d returned %d",
                 dataset, address, err);
      }
      else {
        sprintf (buffer.msg, "<OK> show %d.%d %d", dataset, address, data);
      }
      SetCtrlVal (panelHandle, PANEL_data, data);
      SetCtrlVal (panelHandle, PANEL_HISTORY, &buffer.msg[0]);
      SetCtrlVal (panelHandle, PANEL_HISTORY, "\n");
      break;
  }
  return 0;
}

int CVICALLBACK clear (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
  switch (event) {
    case EVENT_COMMIT:
      FlushOutQ (port);
      FlushInQ (port);
      break;
  }
```

```
  return 0;
}
```

G.2 Receiver Monitor Panel

G.2.1 mm rx.c

```
/*****************************************************************************/
/*                      CSIRO Australia Telescope                             */
/*                          Receiver Group                                    */
/*                                                                            */
/*  Title:          Millimetre Receiver Monitor Program V0.1                  */
/*  File:           mm_rx.c                                                   */
/*  Description:    main routines.                                            */
/*  Author:         Suzy Jackson                                              */
/*  Last Modified:  22-6-01                                                   */
/*****************************************************************************/


/*****************************************************************************
    Includes
*****************************************************************************/

#include <rs232.h>
#include <cvirte.h>
#include <stdio.h>
#include <string.h>
#include <userint.h>
#include <utility.h>
#include "panel.h"                                      /* GUI defines */
#include "Dset_96.h"

#define CONTROL 0xC0000000
#define MONITOR 0x40000000

#define ESC     0x1b
#define SYN     0x16
#define NAK     0x15
#define ACK     0x06
#define BEL     0x07


int CVICALLBACK GetDataThread (void *functionData);

/*****************************************************************************
    Global variables (not inc GUI ones)
*****************************************************************************/

int port = 1, baud = 115200, port_open = 0;     /* useful info about the datasets */
int panelHandle, aboutHandle, menuHandle;
int MonitorGo = 1;                              /* Used to stop monitor process */
```

```
/*****************************************************************************
    Main Routine
*****************************************************************************/

int main (int argc, char *argv[])

{
    if (InitCVIRTE (0, argv, 0) == 0) return -1;
    /* load all panels into memory */
    if ((panelHandle = LoadPanel (0, "panel.uir", PANEL)) < 0)
        return -1;
    if ((aboutHandle = LoadPanel (0, "panel.uir", ABOUT)) < 0)
        return -1;
    if ((menuHandle = LoadMenuBar (panelHandle, "panel.uir", MENU)) < 0)
        return -1;
    /* insert the compile date in the about_date string */
    #ifdef __STDC__
    SetCtrlVal (aboutHandle, ABOUT_DATE, __DATE__);
    #endif
    /* display the introductory panel */
    DisplayPanel (aboutHandle);
    /* open the port */
    {   if (OpenComConfig (port, "", baud, 1, 8, 1, 64, 64))
        return(-1);
        port_open=1;
    }
    FlushOutQ (port);
    FlushInQ (port);
    /* start the data acquisition thread */
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE,
                                   GetDataThread, NULL, NULL);
    Delay (2.0);
    /* now hide the about screen and bring up the main panel */
    HidePanel (aboutHandle);
    DisplayPanel (panelHandle);
    /* run the GUI, so people can control things. */
    RunUserInterface ();
}

int Dataset_Out(int ds_address, int control_point, int data_out)
{

/* Error codes:            0   success
                           -1  comms error
                           -2  dataset error
                           -3  invalid control point
                           -4  invalid data out
*/
    int message, err;
    char reply[3];
```

```
    if (control_point < 0 || control_point > 511) return(-3);
    if (data_out < 0 || data_out > 0xffff) return(-4);
    message = CONTROL + (ds_address << 25) + (control_point << 16) + data_out;
    if (err = SendMessage(message)) return(err);
    ComRd(port, reply, 3);
    if (reply[0] != ACK) return(-2);
    else return(0);
}

int Dataset_In(int ds_address, int monitor_point, int *data_in)
{

/* Error codes:            0   success
                           -1  comms error
                           -2  dataset error
                           -3  invalid control point
*/
    int message, err, response;

    if (monitor_point < 0 || monitor_point > 511) return(-3);
    message = MONITOR + (ds_address << 25) + (monitor_point << 16);
    if (err = SendMessage(message)) return(err);
    if (err = ReadResponse(&response)) return(err);
    *data_in = (response & 0xffff);
    return(0);
}


/******************************************************************************
    GetDataThread
        Reads the analog monitor points, and
        displays the result on the panel meters.
******************************************************************************/

int CVICALLBACK GetDataThread (void *functionData)

{
    int regdata, err, i, band;
    int dsplkup[48] = { PANEL_VDS_1A, PANEL_VDS_2A, PANEL_VDS_3A, PANEL_VDS_4A,
                        PANEL_VGS_1A, PANEL_VGS_2A, PANEL_VGS_3A, PANEL_VGS_4A,
                        PANEL_IDS_1A, PANEL_IDS_2A, PANEL_IDS_3A, PANEL_IDS_4A,
                        PANEL_VDS_1B, PANEL_VDS_2B, PANEL_VDS_3B, PANEL_VDS_4B,
                        PANEL_VGS_1B, PANEL_VGS_2B, PANEL_VGS_3B, PANEL_VGS_4B,
                        PANEL_IDS_1B, PANEL_IDS_2B, PANEL_IDS_3B, PANEL_IDS_4B,
                        PANEL_VDS_5A, PANEL_VDS_6A, PANEL_VDS_7A, PANEL_VDS_8A,
                        PANEL_VGS_5A, PANEL_VGS_6A, PANEL_VGS_7A, PANEL_VGS_8A,
                        PANEL_IDS_5A, PANEL_IDS_6A, PANEL_IDS_7A, PANEL_IDS_8A,
                        PANEL_VDS_5B, PANEL_VDS_6B, PANEL_VDS_7B, PANEL_VDS_8B,
                        PANEL_VGS_5B, PANEL_VGS_6B, PANEL_VGS_7B, PANEL_VGS_8B,
                        PANEL_IDS_5B, PANEL_IDS_6B, PANEL_IDS_7B, PANEL_IDS_8B };
    int wlkup[48] = { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13,
                      14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
```

```
                  28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
                  42, 43, 44, 45, 46, 47 };
int klkup[24] = { 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
                  62, 63, 64, 65, 66, 67, 68, 69, 70, 71 };
int qlkup[24] = { 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
                  86, 87, 88, 89, 90, 91, 92, 93, 94, 95 };

int dewlkup[26] = { 96,  97,  98,  99,  100, 101, 102, 103, 104, 105, 106,
                    107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117,
                    118, 119, 120, 121 };
int dewdlkup[26] = { PANEL_32V, PANEL_20V, PANEL__20V, PANEL_9V, PANEL_15R1,
                     PANEL__15R1, PANEL_5R1, PANEL_15R2, PANEL__15R2,
                     PANEL_5R2, PANEL_15R3, PANEL__15R3, PANEL_5R3,
                     PANEL_15R4, PANEL__15R4, PANEL_5R4, PANEL_20K1,
                     PANEL_70K1, PANEL_20K2, PANEL_70K2, PANEL_SUP1,
                     PANEL_RET1, PANEL_SUP2, PANEL_RET2, PANEL_VAC1,
                     PANEL_VAC2 };
for (;;) {
    GetCtrlVal (panelHandle, PANEL_MON_LNA, &band);
    switch (band) {
        case 0:
            for (i=0;i<24;i++) {
                if (MonitorGo) {
                    Dataset_In (0, klkup[i], &regdata);
                    SetCtrlVal (panelHandle, dsplkup[i],
                                ((double) (short) regdata)/8192);
                }
            }
            break;
        case 1:
            for (i=0;i<24;i++) {
                if (MonitorGo) {
                    Dataset_In (0, qlkup[i], &regdata);
                    SetCtrlVal (panelHandle, dsplkup[i],
                                ((double) (short) regdata)/8192);
                }
            }
            break;
        default:
            for (i=0;i<48;i++) {
                if (MonitorGo) {
                    Dataset_In (0, wlkup[i], &regdata);
                    SetCtrlVal (panelHandle, dsplkup[i],
                                ((double) (short) regdata)/8192);
                }
            }
            break;
    }
    for (i=0;i<26;i++) {
        if (MonitorGo) {
            Dataset_In (0, dewlkup[i], &regdata);
            SetCtrlVal (panelHandle, dewdlkup[i],
```

```
                        ((double) (short) regdata)/8192);
            }
        }
        if (MonitorGo) {
            Dataset_In (1, 7, &regdata);
            SetCtrlVal (panelHandle, PANEL_KA, (regdata&0x01));
            SetCtrlVal (panelHandle, PANEL_KB, (regdata&0x02));
            SetCtrlVal (panelHandle, PANEL_QA, (regdata&0x04));
            SetCtrlVal (panelHandle, PANEL_QB, (regdata&0x08));
            SetCtrlVal (panelHandle, PANEL_WA1, (regdata&0x10));
            SetCtrlVal (panelHandle, PANEL_WA2, (regdata&0x20));
            SetCtrlVal (panelHandle, PANEL_WB1, (regdata&0x40));
            SetCtrlVal (panelHandle, PANEL_WB2, (regdata&0x80));
        }
    }
}


/*****************************************************************************
    Shutdown
        Called when the user hits the quit button.
*****************************************************************************/

int CVICALLBACK Shutdown (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)

{
    switch (event) {
        case EVENT_COMMIT:
            QuitUserInterface (0);
            break;
    }
    return 0;
}


/*****************************************************************************
    band_change
        Called when the user changes the displayed band.
*****************************************************************************/

int CVICALLBACK band_change (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)

{
    int band, i;
    int lookup[24] = {  PANEL_VDS_5A, PANEL_VDS_6A, PANEL_VDS_7A, PANEL_VDS_8A,
                        PANEL_VGS_5A, PANEL_VGS_6A, PANEL_VGS_7A, PANEL_VGS_8A,
                        PANEL_IDS_5A, PANEL_IDS_6A, PANEL_IDS_7A, PANEL_IDS_8A,
                        PANEL_VDS_5B, PANEL_VDS_6B, PANEL_VDS_7B, PANEL_VDS_8B,
                        PANEL_VGS_5B, PANEL_VGS_6B, PANEL_VGS_7B, PANEL_VGS_8B,
                        PANEL_IDS_5B, PANEL_IDS_6B, PANEL_IDS_7B, PANEL_IDS_8B };
```

```
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panel, control, &band);
            for (i=0;i<24;i++)
                SetCtrlAttribute (panel, lookup[i], ATTR_DIMMED, (band!=2));
            break;
    }
    return 0;
}



/*****************************************************************************
    Menu
        Code to make sense of the pulldown menus.
*****************************************************************************/

void CVICALLBACK Menu(int menubar, int menuItem, void *callbackData, int panel)

{
    char about_string [200];
    int temp;
    switch (menuItem) {
        case MENU_FILE_QUIT:
            QuitUserInterface(0);
            break;
        case MENU_HELP_ABOUT:
            DisplayPanel (aboutHandle);
            break;
    }
}



/*****************************************************************************
    CLOSE_ABOUT
        Shuts down the about window.
*****************************************************************************/

int CVICALLBACK CLOSE_ABOUT (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    switch (event) {
        case EVENT_COMMIT:
            HidePanel (aboutHandle);
            break;
    }
    return 0;
}
```

```
/******************************************************************************
    control
        Called when the user changes a control bit.
******************************************************************************/


int CVICALLBACK control (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    switch (event) {
        case EVENT_COMMIT:

            break;
    }
    return 0;
}
```

G.3 Water Vapour Radiometer Server

G.3.1 wvr.c

```
/****************************************************************************/
/*                      CSIRO Australia Telescope                           */
/*                          Receiver Group                                  */
/*                                                                          */
/*  Title:          Water Vapour Radiometer Program V1.0                    */
/*  File:           wvr.c                                                   */
/*  Description:    main routines.                                          */
/*  Author:         Suzy Jackson                                            */
/*  Last Modified:  14-8-2000                                               */
/****************************************************************************/


/****************************************************************************
    Includes
****************************************************************************/

#include <tcpsupp.h>
#include <cvirte.h>
#include <stdio.h>
#include <string.h>
#include <userint.h>
#include <utility.h>
#include "panel.h"
#include <rs232.h>
#include "dset_96.h"
#include <easyio.h>
#include <ansi_c.h>
```

```
/****************************************************************************
    Hardware Defines - these values need to reflect hardware reality
****************************************************************************/

#define MSGHD 71

#define MONITOR    0x40000000

#define DAQ_Device 1
#define DAQ_MAX 10
#define DAQ_MIN -10
#define Noise_Port "0"
#define Noise_Line 0

#define DSA 2
#define PORT 1
#define BAUD 38400


/****************************************************************************
    Structures
****************************************************************************/

/* a structure to store values we've read while we're waiting to write them
   to a file */

typedef struct List {
    double data[16];
    char time[9];
    double sec;
    struct List *next;
} list;

struct msg_type
    {
        char msgHD;
        unsigned char msglen;
        char msg[512];
    };

/****************************************************************************
    Prototypes
****************************************************************************/

list* Add_To_List (list *head, char *time, double sec, double *data);
int Dataset_In(int ds_address, int monitor_point, int *data_in);
int CVICALLBACK GetDataThread (void *functionData);


/****************************************************************************
    Global variables
```

```
*************************************************************************/

int port = PORT, port_open, panelHandle, menuHandle, aboutHandle,
    file_len = 99999999, list_len = 0;
FILE *logfile;
char logbase [MAX_PATHNAME_LEN] = {"wvr_data"};
char logpath [MAX_PATHNAME_LEN];
list *head = NULL;
int TCPHandle, TCPConnect = 0, getdataflag = 0;


/*************************************************************************
    TCPServerCallback
        Called when data is waiting on the TCP stack.
*************************************************************************/

int CVICALLBACK TCPServerCallback (unsigned handle, int event, int error,
                                   void *callbackData)

{
    char inbuf[512], outbuf[300], command[80], path[256];
    int file_size, update_rate, device, dataSize = 512, i, intvalue;
    float floatvalue;
    list * curr;
    struct msg_type bufdata;

    switch (event) {
        case TCP_CONNECT:
            TCPHandle = handle;
            SetCtrlVal (panelHandle, PANEL_HISTORY,
                        "New connection established\n");
            TCPConnect = 1;
            break;
        case TCP_DATAREADY:
            if ((dataSize = ServerTCPRead(TCPHandle, &bufdata,
                dataSize, 1000)) == -kTCP_ConnectionClosed) {
                SetCtrlVal (panelHandle, PANEL_HISTORY,
                            "TCP Read Error - connection closed\n");
                TCPConnect = 0;
            }
            /* extract the message from the buffer */
            memcpy (inbuf, &bufdata.msg, bufdata.msglen-2);
            /* append a NULL so it looks like a string */
            inbuf[bufdata.msglen-2]=0;
            sscanf (inbuf,"%s ", command);
            if (!strcmp (command,"INTERVAL")) {
                sscanf (inbuf,"%s %f", command, &floatvalue);
                SetCtrlVal (panelHandle, PANEL_INTERVAL, floatvalue);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
            }
            else if (!strcmp (command, "FILE_SIZE")) {
                sscanf (inbuf,"%s %d", command, &intvalue);
```

```
                SetCtrlVal (panelHandle, PANEL_FILE_SIZE, intvalue);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
            }
            else if (!strcmp (command, "DEVICE")) {
                sscanf (inbuf,"%s %d", command, &intvalue);
                SetCtrlVal (panelHandle, PANEL_DEVICE, intvalue);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
                DEVICE (panelHandle, PANEL_DEVICE, EVENT_COMMIT,
                        NULL, NULL, NULL);

            }
            else if (!strcmp (command, "TIMESOURCE")) {
                sscanf (inbuf,"%s %d", command, &intvalue);
                SetCtrlVal (panelHandle, PANEL_TIMESOURCE, intvalue);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
            }
            else if (!strcmp (command, "GO")) {
                sscanf (inbuf,"%s %d", command, &intvalue);
                SetCtrlVal (panelHandle, PANEL_GO, intvalue);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
                START (panelHandle, PANEL_GO, EVENT_COMMIT,
                       NULL, NULL, NULL);
            }
            else if (!strcmp (command, "AVERAGE")) {
                sscanf (inbuf,"%s %d", command, &intvalue);
                SetCtrlVal (panelHandle, PANEL_AVERAGE, intvalue);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
            }
            else if (!strcmp (command, "LOGBASE")) {
                sscanf (inbuf,"%s %s", command, logbase);
                sprintf (bufdata.msg, "%s <done>\n",inbuf);
            }
            else {
                sprintf (bufdata.msg, "<ERR> invalid message\n");
            }
            SetCtrlVal (panelHandle, PANEL_HISTORY, bufdata.msg);
            bufdata.msglen = strlen (bufdata.msg) + 2;
            bufdata.msgHD = MSGHD;
            if (ServerTCPWrite (TCPHandle, &bufdata, bufdata.msglen,
                1000)<0) {
                MessagePopup ("ERROR", "TCP write error.");
            }
            break;
        case TCP_DISCONNECT:
            SetCtrlVal (panelHandle, PANEL_HISTORY,
                        "Connection closed by client\n");
            TCPConnect = 0;
            break;
    }
    return (0);
}
```

```
/***************************************************************************
    Main Routine
***************************************************************************/

int main (int argc, char *argv[])

{
    list *curr;
    if (InitCVIRTE (0, argv, 0) == 0) return -1;
    /* load all panels into memory */
    if ((panelHandle = LoadPanel (0, "panel.uir", PANEL)) < 0)
        return -1;
    if ((aboutHandle = LoadPanel (0, "panel.uir", ABOUT)) < 0)
        return -1;
    if ((menuHandle = LoadMenuBar (panelHandle, "panel.uir", MENU)) < 0)
        return -1;
    /* Set the compile date */
    #ifdef __STDC__
        SetCtrlVal (aboutHandle, ABOUT_DATE, __DATE__);
    #endif
    /* start the data acquisition thread */
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, GetDataThread,
                                   NULL, NULL);
    /* bring up the main panel */
    DisplayPanel (panelHandle);
    SetCtrlVal (panelHandle, PANEL_HISTORY,
                "Registering TCP Server on port 4321: ");
    if (RegisterTCPServer(4321, TCPServerCallback, NULL)<0)
        SetCtrlVal (panelHandle, PANEL_HISTORY, "Failed\n");
    else SetCtrlVal (panelHandle, PANEL_HISTORY, "Successful\n");
    /* run the GUI, so people can control things. */
    RunUserInterface ();
    /* we're leaving, so close up behind us,
       and free any memory we've malloc'd  */
    while (head) {
        curr = head->next;
        free (head);
        head = curr;
    }
    return 0;
}


/***************************************************************************
    Shutdown
        Called when the user hits the quit button.
***************************************************************************/

int CVICALLBACK Shutdown (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
```

```
{
    switch (event) {
        case EVENT_COMMIT:
            QuitUserInterface (0);
            break;
    }
    return 0;
}


/*****************************************************************************
    Menu
        Code to make sense of the pulldown menus.
*****************************************************************************/

void CVICALLBACK Menu(int menubar, int menuItem, void *callbackData,
                      int panel)

{
    switch (menuItem) {
        case MENU_FILE_LOGFILE:
            FileSelectPopup ("","*.csv","*.csv","Log File",VAL_OK_BUTTON,
                             0,0,1,1,logbase);
            /* we need to remove the .csv, as this is added later, along with
               the current date and time, when the file is first created. */
            if (strchr(logbase,'.')) *strchr(logbase,'.')=NULL;
            break;
        case MENU_FILE_QUIT:
            QuitUserInterface (0);
            break;
        case MENU_HELP_ABOUT:
            DisplayPanel (aboutHandle);
            break;
    }
}


/*****************************************************************************
    Dateset_In
        Provides input from a dataset without address translation.
*****************************************************************************/

int Dataset_In(int ds_address, int monitor_point, int *data_in)
{

/* Error codes:            0   success
                           -1  comms error
                           -2  dataset error
                           -3  invalid control point
*/
    int message, err, response;
```

```
    if (monitor_point < 0 || monitor_point > 511) return(-3);
    message = MONITOR + (ds_address << 25) + (monitor_point << 16);
    if (err = SendMessage(message)) return(err);
    if (err = ReadResponse(&response)) return(err);
    *data_in = (response & 0xffff);
    return(0);
}


/*****************************************************************************
    Add_To_List
        Adds the current data to a linked list, checks whether we have
        enough data to archive, and if we do writes the list out to a file
        before trashing it.
*****************************************************************************/

list* Add_To_List (list *head, char *time, double sec, double *data)

{
    list *new, *curr;
    static list *last;
    int i, hours, minutes, seconds, file_size, dump, n;
    char *date, linebuf [256];
    struct msg_type bufdata;
    GetCtrlVal (panelHandle, PANEL_FILE_SIZE, &file_size);
    /* create new list item */
    new = malloc(sizeof(list));
    /* copy values into item */
    strcpy (new->time, time);
    for (i=0;i<16;i++) new->data[i]=data[i];
    new->sec=sec;
    /* give item a null pointer */
    new->next = NULL;
    /* is this the first time?  yes - point head to item */
    if (!head) {
        head = new;
        list_len = 0;
    }
    /* no - point last item to item */
    else last->next = new;
    last = new;
    list_len ++;
    /* do we have enough items to offload to a file? */
    if (list_len%10 == 0) {
        /* first check if it's time for a new file */
        if (file_len>=file_size) {
            /* Generate the new filename, and write the header */
            date = DateStr ();
            sscanf (head->time,"%d:%d:%d",&hours,&minutes,&seconds);
            sprintf (logpath, "%s-%s-%02d-%02d-%02d.csv", logbase, date,
                     hours, minutes, seconds);
            if (!(logfile = fopen (logpath, "w"))) {
```

```
            sprintf (bufdata.msg,"ERROR - Unable to open %s\n",logpath);
            SetCtrlVal (panelHandle, PANEL_HISTORY, bufdata.msg);
            bufdata.msglen = strlen (bufdata.msg) + 2;
            bufdata.msgHD = MSGHD;
            if (TCPConnect) {
                if (ServerTCPWrite (TCPHandle, &bufdata,
                    bufdata.msglen, 1000)<0) {
                    MessagePopup ("ERROR", "TCP write error.");
                }
            }
            else MessagePopup ("ERROR", "Cannot open file");
        }
        else {
            file_len = 0;
            fprintf (logfile,"Water Vapour Radiometer Logfile\n");
            fclose (logfile);
            sprintf (bufdata.msg,"Opened %s\n",logpath);
            SetCtrlVal (panelHandle, PANEL_HISTORY, bufdata.msg);
            bufdata.msglen = strlen (bufdata.msg) + 2;
            bufdata.msgHD = MSGHD;
            if (TCPConnect) {
                if (ServerTCPWrite (TCPHandle, &bufdata,
                    bufdata.msglen, 1000)<0) {
                    MessagePopup ("ERROR", "TCP write error.");
                }
            }
            ClearStripChart (panelHandle, PANEL_INPUTS);
        }
    }
    if (logfile = fopen (logpath, "a")) {
    /* Now offload the buffer to the file.  Note that if we're unable to
       open the file, we don't worry about it, but just try again next
       time.  This way, we're able to cope with ppl accessing the file
       we're writing to with no loss of data. */
        for (dump=20;dump&&head;dump--) {
            /* we offload up to 20 elements at a time, or the whole list,
               whichever is less. This way, we don't risk missing the
               next read because we're busy dumping a large number of
               records, but are still able to "catch up" reasonably
               quickly, if we haven't been able to access the file for a
               while. */
            fprintf (logfile, "%s, %f, ",head->time, head->sec);
            for (i=0;i<16;i++) fprintf (logfile, "%f, ",head->data[i]);
            fprintf (logfile, "\n");
            curr = head->next;
            free (head);
            head = curr;
            list_len --;
            file_len ++;
        }
        /* close up behind us, so that others can access the file */
        fclose (logfile);
```

```
            sprintf (bufdata.msg,"Logged %d samples\n",file_len);
            GetNumTextBoxLines(panelHandle, PANEL_HISTORY, &n);
            if (n>1) {
                GetTextBoxLine(panelHandle, PANEL_HISTORY, n-2, linebuf);
                if (!strncmp(bufdata.msg,"Logged",6)
                    && !strncmp(linebuf,"Logged",6))
                    DeleteTextBoxLine(panelHandle,PANEL_HISTORY,n-2);
                SetCtrlVal (panelHandle, PANEL_HISTORY, bufdata.msg);
            }
            bufdata.msglen = strlen (bufdata.msg) + 2;
            bufdata.msgHD = MSGHD;
            if (TCPConnect) {
                if (ServerTCPWrite (TCPHandle, &bufdata,
                    bufdata.msglen, 1000)<0) {
                    MessagePopup ("ERROR", "TCP write error.");
                }
            }
        }
    }
    return head;
}


/*****************************************************************************
    TIMER
        Called when it's time to log a sample.
*****************************************************************************/

int CVICALLBACK TIMER (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    getdataflag = 1;
    return (0);
}


/*****************************************************************************
    GetDataThread
        Runs continuously as a separate thread.  When getdataflag is set by
        main thread, it clears the flag, then gets a sample, and adds it to
        the list, as well as displaying it on the chart.  Is able to deal
        with National Instruments DAQ cards (8 channel, 16 bit) as well as
        the AT WVR interface (16 channel, 24 bit).
*****************************************************************************/

int CVICALLBACK GetDataThread (void *functionData)

{
    struct msg_type bufdata;
    char *curr_time;
    char errstring[40];
    int i, j, intdata, loworder, device, average, timesource, noise_interval,
```

```
        noise_en, err;
    short temp[8196], status = 0;
    short chanvector[8] = {0,1,2,3,4,5,6,7};
    short gainvector[8] = {1,1,1,1,1,1,1,1};
    double data[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    double curr_sec;
    for (;;) {
        if (getdataflag) {
            getdataflag = 0;
            GetCtrlVal (panelHandle, PANEL_DEVICE, &device);
            GetCtrlVal (panelHandle, PANEL_AVERAGE, &average);
            GetCtrlVal (panelHandle, PANEL_NOISE, &noise_interval);
            GetCtrlVal (panelHandle, PANEL_NOISE_EN, &noise_en);
            curr_time = TimeStr();
            curr_sec = Timer();
            if (device) {
                /* set the DAQ card process up */
//              SCAN_Setup (DAQ_Device, 8, chanvector,gainvector);
                /* SCAN_Start hangs if we give it 1 scan.  For both cases,
                   we take samples at 20 us intervals (50KHz), and scan each
                   set at 200 us intervals, but for the single scan case, we
                   actually take 2 scans and ignore the second. */
//              if (average==1)
//                  SCAN_Start (DAQ_Device, &temp[0], 16, 1, 20, 1, 200);
//              else
//                  SCAN_Start (DAQ_Device, &temp[0],
//                              average*8, 1, 20, 1, 200);
                /* wait for the process to finish */
//              while (!status) DAQ_Check (DAQ_Device, &status, &i);
                /* average the result */
//              for (j=0;j<average;j++)
//                  for (i=0;i<8;i++)
//                      data[i]+=temp[i+j*8]/(3276.8*average);
//              WriteToDigitalLine (DAQ_Device, Noise_Port, Noise_Line, 8,
//                                  0, !(noise_en &&
//                                  ((file_len+list_len)%noise_interval==0)));
            }
            else {
                /* The AT Dataset interface uses the WVR interface card, with
                   24 bit ADC. The ADC samples 16 inputs, but at a refresh
                   rate of only 6.25Hz.  Thus no averaging is employed (nor
                   should it be needed) */
                for (i=0;i<16;i++) {
                    if (err = Dataset_In (DSA, i, &intdata)){
                        data[i]=-11;
                        sprintf (bufdata.msg, "Dataset_In returned %d\n", err);
                        SetCtrlVal (panelHandle, PANEL_HISTORY, bufdata.msg);
                        if (TCPConnect) {
                            bufdata.msglen = strlen (bufdata.msg) + 2;
                            bufdata.msgHD = MSGHD;
                            if (ServerTCPWrite (TCPHandle, &bufdata,
                                bufdata.msglen, 1000)<0) {
```

```
                                MessagePopup ("ERROR", "TCP write error.");
                            }
                        }
                    }
                    else {
                        if (err = Dataset_In (DSA, 16, &loworder)) {
                            data[i]=-11;
                            sprintf (bufdata.msg,
                                      "Dataset_In returned %d\n", err);
                            SetCtrlVal (panelHandle, PANEL_HISTORY,
                                        bufdata.msg);
                            if (TCPConnect) {
                                bufdata.msglen = strlen (bufdata.msg) + 2;
                                bufdata.msgHD = MSGHD;
                                if (ServerTCPWrite (TCPHandle, &bufdata,
                                    bufdata.msglen, 1000)<0) {
                                    MessagePopup ("ERROR", "TCP write error.");
                                }
                            }
                        }
                        else {
                            intdata = (intdata * 256)
                                      + (loworder & 0x00ff)+1;
                            if (intdata > 8388352) intdata -= 16777216;
                            /* scale to volts */
                            data[i]=(double)intdata/(838860.8);
                        }
                    }
                }
            }
            PlotStripChart (panelHandle, PANEL_INPUTS, &data[0], 16, 0, 0,
                            VAL_DOUBLE);
            head = Add_To_List (head, curr_time, curr_sec, &data[0]);
        }
    }
    return 0;
}


/***************************************************************************
    CLOSE_ABOUT
        Shuts down the about window.
***************************************************************************/

int CVICALLBACK CLOSE_ABOUT (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    switch (event) {
        case EVENT_COMMIT:
            HidePanel (aboutHandle);
            break;
    }
```

```
    return 0;
}


/*****************************************************************************
    CHART_SCALE
        Changes the scaling for the stripchart.
*****************************************************************************/

int CVICALLBACK CHART_SCALE (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    double min, max, temp;
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panel, PANEL_CHART_MIN, &min);
            GetCtrlVal (panel, PANEL_CHART_MAX, &max);
            if (min > max) {
                SetCtrlVal (panel, PANEL_CHART_MIN, max);
                SetCtrlVal (panel, PANEL_CHART_MAX, min);
                temp = min;
                min = max;
                max = temp;
            }
            SetAxisScalingMode (panel, PANEL_INPUTS, VAL_LEFT_YAXIS,
                                VAL_MANUAL, min, max);
            break;
    }
    return 0;
}


/*****************************************************************************
    DEVICE
        Input device change.
*****************************************************************************/

int CVICALLBACK DEVICE (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    int device;
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panel, PANEL_DEVICE, &device);
            if (device == 0) {
                SetCtrlVal (panel, PANEL_AVERAGE, 1);
                SetCtrlAttribute (panel, PANEL_AVERAGE, ATTR_DIMMED, 1);
                SetCtrlAttribute (panel, PANEL_INTERVAL, ATTR_MIN_VALUE, 0.2);
            }
            else {
                SetCtrlAttribute (panel, PANEL_AVERAGE, ATTR_DIMMED, 0);
                SetCtrlAttribute (panel, PANEL_INTERVAL, ATTR_MIN_VALUE, 0.05);
```

```
            }
            break;
    }
    return 0;
}


/*************************************************************************
    START
        Initialises inputs when we start recording.
*************************************************************************/

int CVICALLBACK START (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    int device, go, i;
    double interval;
    list *curr;
    struct msg_type bufdata;

    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panel, PANEL_DEVICE, &device);
            GetCtrlVal (panel, PANEL_GO, &go);
            GetCtrlVal (panelHandle, PANEL_INTERVAL, &interval);
            SetCtrlAttribute (panelHandle, PANEL_TIMER, ATTR_INTERVAL,
                              interval);
            SetCtrlAttribute (panelHandle, PANEL_TIMER, ATTR_ENABLED, go);
            if (!device && go) {
                Initialise_Dataset(DSA, port, BAUD);
                SetComTime (port, 0.5);
            }
            if (!go) {
                if (logfile = fopen (logpath, "a")) {
                    /* Offload the buffer to the file. */
                    while (head) {
                        fprintf (logfile, "%s, %f, ",head->time,
                                 head->sec);
                        for (i=0;i<8;i++)
                            fprintf (logfile, "%f, ",head->data[i]);
                        fprintf (logfile, "\n");
                        curr = head->next;
                        free (head);
                        head = curr;
                    }
                    /* close up behind us. */
                    fclose (logfile);
                    /* re-initialise file and list lengths */
                    file_len = 99999999;
                    list_len = 0;
                    SetCtrlVal (panelHandle, PANEL_HISTORY,
                                "Data Collection Stopped\n");
```

```
                    sprintf (bufdata.msg,"Data Collection Stopped\n");
                }
            }
            else {
                SetCtrlVal (panelHandle, PANEL_HISTORY,
                            "Data Collection Started\n");
                sprintf (bufdata.msg,"Data Collection Started\n");
            }
            if (TCPConnect) {
                bufdata.msglen = strlen (bufdata.msg) + 2;
                bufdata.msgHD = MSGHD;
                if (ServerTCPWrite (TCPHandle, &bufdata,
                    bufdata.msglen, 1000)<0) {
                    MessagePopup ("ERROR", "TCP write error.");
                }
            }
            break;
    }
    return 0;
}
```

G.4 Water Vapour Radiometer Client

G.4.1 wvr client.c

```
/****************************************************************************

Water Vapour Radiometer Client

Allows us to remotely control the water vapour radiometer data
collection process..

Version:        0.1
Commenced:      5 Sep 2000
Last Revised:   5 Sep 2000
Author:         Suzy Jackson <sjackson@atnf.csiro.au>

****************************************************************************/

#include <ansi_c.h>
#include <cvirte.h>
#include <userint.h>
#include <tcpsupp.h>
#include "client.h"

#define MSGHD 71


/****************************************************************************
    A data structure to hold request/control data.
****************************************************************************/
```

```
struct msg_type
    {
        char msgHD;
        unsigned char msglen;
        char msg[512];
    };


/****************************************************************************
    Prototypes.
****************************************************************************/

int CVICALLBACK Shutdown (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2);
int CVICALLBACK SEND (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2);
int CVICALLBACK TCPClientCallback (unsigned handle, int event, int error,
        void *callbackData);
int CVICALLBACK CONNECT (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2);
void CVICALLBACK Menu (int menuBar, int menuItem, void *callbackData,
        int panel);
int CVICALLBACK CLOSE_ABOUT (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2);


/****************************************************************************
    Global Variables.
****************************************************************************/

int panelHandle, aboutHandle, menuHandle, TCPHandle;


/****************************************************************************
    Main program.
****************************************************************************/

int main (int argc, char *argv[])
{
    char miscstring [300];

    if (InitCVIRTE (0, argv, 0) == 0)
        /* Needed if linking in external compiler */
        return -1;  /* out of memory */
    if ((panelHandle = LoadPanel (0, "client.uir", PANEL)) < 0)
        return -1;
    if ((aboutHandle = LoadPanel (0, "client.uir", ABOUT)) < 0)
        return -1;
    if ((menuHandle = LoadMenuBar (panelHandle, "client.uir", MENU)) < 0)
        return -1;
    if (argc == 2) {
```

```
        SetWaitCursor (1);
        if (ConnectToTCPServer (&TCPHandle, 4321, argv[1],
            TCPClientCallback, NULL, 5000) <0) {
            MessagePopup ("ERROR", "Unable to connect to server.");
            SetCtrlVal (panelHandle, PANEL_CONNECT, 0);
        }
        else {
            SetCtrlVal (panelHandle, PANEL_HISTORY, "Connected to ");
            SetCtrlVal (panelHandle, PANEL_HISTORY, argv[1]);
            SetCtrlVal (panelHandle, PANEL_HISTORY, "\n");
            sprintf (miscstring,"WVR Client - %s",argv[1]);
            SetPanelAttribute (panelHandle, ATTR_TITLE, miscstring);
            SetCtrlVal (panelHandle, PANEL_CONNECT, 1);
        }
        SetWaitCursor (0);
    }
    DisplayPanel (panelHandle);
    RunUserInterface ();
    return 0;
}


/*****************************************************************************
    Shutdown - called when user hits the quit button.
*****************************************************************************/

int CVICALLBACK Shutdown (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)

{
    switch (event) {
        case EVENT_COMMIT:
            QuitUserInterface (0);
            break;
    }
    return 0;
}


/*****************************************************************************
    Send - sends data to the server.
*****************************************************************************/

int CVICALLBACK SEND (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)

{
    int intdata;
    double doubledata;
    struct msg_type message;

    switch (event) {
```

```
        case EVENT_COMMIT:
            switch (control) {
                case PANEL_INTERVAL:
                    GetCtrlVal (panelHandle, PANEL_INTERVAL, &doubledata);
                    sprintf (message.msg,"INTERVAL %f",doubledata);
                    break;
                case PANEL_FILE_SIZE:
                    GetCtrlVal (panelHandle, PANEL_FILE_SIZE, &intdata);
                    sprintf (message.msg,"FILE_SIZE %d",intdata);
                    break;
                case PANEL_DEVICE:
                    GetCtrlVal (panelHandle, PANEL_DEVICE, &intdata);
                    if (intdata == 0) {
                        SetCtrlVal (panel, PANEL_AVERAGE, 1);
                        SetCtrlAttribute (panel, PANEL_AVERAGE,
                                          ATTR_DIMMED, 1);
                        SetCtrlAttribute (panel, PANEL_INTERVAL,
                                          ATTR_MIN_VALUE, 0.2);
                    }
                    else {
                        SetCtrlAttribute (panel, PANEL_AVERAGE,
                                          ATTR_DIMMED, 0);
                        SetCtrlAttribute (panel, PANEL_INTERVAL,
                                          ATTR_MIN_VALUE, 0.05);
                    }
                    sprintf (message.msg,"DEVICE %d",intdata);
                    break;
                case PANEL_TIMESOURCE:
                    GetCtrlVal (panelHandle, PANEL_TIMESOURCE,
                                &intdata);
                    sprintf (message.msg,"TIMESOURCE %d",intdata);
                    break;
                case PANEL_GO:
                    GetCtrlVal (panelHandle, PANEL_GO, &intdata);
                    sprintf (message.msg,"GO %d",intdata);
                    break;
                case PANEL_AVERAGE:
                    GetCtrlVal (panelHandle, PANEL_AVERAGE, &intdata);
                    sprintf (message.msg,"AVERAGE %d",intdata);
                    break;
            }
            /* create the message */
            message.msglen = strlen (message.msg) + 2;
            message.msgHD = MSGHD;
            /* send it */
            if (ClientTCPWrite (TCPHandle, &message, message.msglen,
                1000)<0) {
                MessagePopup ("ERROR", "TCP write error.");
            }
            break;
    }
    return 0;
```

```
}


/*************************************************************************
    TCPClientCallback - deciphers server responses.
*************************************************************************/

int CVICALLBACK TCPClientCallback (unsigned handle, int event, int error,
                                   void *callbackData)

{
    struct msg_type bufdata;
    int dataSize = 512, n;
    char *temp, inbuf[256], linebuf[256];

    TCPHandle = handle;
    switch (event) {
        case TCP_DATAREADY:
            if ((dataSize = ClientTCPRead(TCPHandle, &bufdata, dataSize,
                1000)) == -kTCP_ConnectionClosed) {
                SetCtrlVal (panelHandle, PANEL_HISTORY,
                            "TCP Read Error - connection closed\n");
            }
            /* extract the message from the buffer */
            memcpy (inbuf, &bufdata.msg, bufdata.msglen-2);
            /* append a NULL so it looks like a string */
            inbuf[bufdata.msglen-2]=0;
            GetNumTextBoxLines(panelHandle, PANEL_HISTORY, &n);
            if (n>1) {
                GetTextBoxLine(panelHandle, PANEL_HISTORY, n-2, linebuf);
                if (!strncmp(inbuf,"Logged",6) && !strncmp(linebuf,"Logged",6))
                    DeleteTextBoxLine(panelHandle,PANEL_HISTORY,n-2);
                SetCtrlVal (panelHandle, PANEL_HISTORY, inbuf);
            }
            if (n>100) DeleteTextBoxLine(panelHandle, PANEL_HISTORY, 0);
            break;
        case TCP_DISCONNECT:
            MessagePopup ("NOTE", "Server has closed connection.");
            SetCtrlVal (panelHandle, PANEL_CONNECT, 0);
            break;
    }
    return (0);
}


/*************************************************************************
    Connect - manages connections to the server.
*************************************************************************/

int CVICALLBACK CONNECT (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
```

```
{
    int connect;
    char hoststring [256], miscstring [300];
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panelHandle, PANEL_CONNECT, &connect);
            if (connect == 1) {
                PromptPopup("Enter Host Name",
                  "Please type the name of the host machine.\n(eg wvrca03-cj)",
                  hoststring, 255);
                SetWaitCursor (1);
                if (ConnectToTCPServer (&TCPHandle, 4321, hoststring,
                    TCPClientCallback, NULL, 5000) <0) {
                    MessagePopup ("ERROR", "Unable to connect to server.");
                    SetCtrlVal (panelHandle, PANEL_CONNECT, 0);
                }
                else {
                    SetCtrlVal (panelHandle, PANEL_HISTORY, "Connected to ");
                    SetCtrlVal (panelHandle, PANEL_HISTORY, hoststring);
                    SetCtrlVal (panelHandle, PANEL_HISTORY, "\n");
                    sprintf (miscstring,"WVR Client - %s",hoststring);
                    SetPanelAttribute (panelHandle, ATTR_TITLE, miscstring);
                }
                SetWaitCursor (0);
            }
            else {
                DisconnectFromTCPServer (TCPHandle);
                SetPanelAttribute (panelHandle, ATTR_TITLE, "WVR Client");
                SetCtrlVal (panelHandle, PANEL_HISTORY,
                            "Disconnected from server\n");
            }
    }
    return 0;
}


/*************************************************************************
    Menu - deals with the pulldown menus.
*************************************************************************/

void CVICALLBACK Menu (int menuBar, int menuItem, void *callbackData,
        int panel)

{
    int err;
    char logbase [512];
    struct msg_type message;
    switch (menuItem) {
        case MENU_FILE_QUIT:
            QuitUserInterface (0);
            break;
        case MENU_FILE_LOGFILE:
```

```
            PromptPopup("Enter Filename",
                  "Please enter the filename prefix.\nThe system will
                  append a date and time field to this for each file.\n
                  eg: wvrdata-ca03", logbase, 256);
            sprintf (message.msg,"LOGBASE %s",logbase);
            message.msglen = strlen (message.msg) + 2;
            message.msgHD = MSGHD;
            if (ClientTCPWrite (TCPHandle, &message, message.msglen,
                1000)<0) {
                MessagePopup ("ERROR", "TCP write error.");
            }
            break;
        case MENU_HELP_ABOUT:
            DisplayPanel (aboutHandle);
        default:
            break;
    }
}


/*****************************************************************************
    CLOSE_ABOUT
        Shuts down the about window.
*****************************************************************************/

int CVICALLBACK CLOSE_ABOUT (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)

{
    switch (event) {
        case EVENT_COMMIT:
            HidePanel (aboutHandle);
            break;
    }
    return 0;
}
```