Design of a hardware platform for narrow-band Software Defined Radio applications

Kalen Watermeyer

A dissertation submitted to the Department of Electrical Engineering
University of Cape Town, in fulfilment of the requirements
for the degree of Masters in Electrical Engineering.

Cape Town, January 2007

Abstract

This thesis describes the design and implementation of a hardware platform for Software Defined Radio (SDR) applications.

The Software Defined Radio Forum describes Software Defined Radio as:

> *a collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multi-mode, multi-band, multi-functional wireless devices that can be adapted, upgraded or enhanced by using software upgrades.* [1]

SDR applications perform demodulation, modulation and other signal-processing of a digitised source signal using software modules running on a PC or other capable device.

SDR systems typically consist of a transceiver, with a wide-band ADC for signal reception and a DAC for signal generation implemented in hardware, with signal-processing software running on a host PC. In many cases pre- or post-processing of the data occurs in FPGA's or other programmable devices.

This project was aimed at creating an ADC-based system that could capture samples for processing in software on a host PC, while providing a framework for functionality enhancements through system extensions. The system was designed to integrate with the existing *GnuRadio* open-source SDR toolkit, with the hardware design was based on the existing USRP system [4].

In the context of the Radar and Remote Sensing Research Group, this system can be implemented in sample capture applications, as well as software RADAR systems through system extension.

The USRP system was developed by the Free Software Foundation *GnuRadio* project to complement their open-source SDR software. It is a single board housing a large FPGA which accepts input signals from add-on ADC daughter-boards [5], and feeds output signals to DAC boards. The USB data transfer to and from the host is carried by a Cypress FX2 microcontroller.

This system is split into two modules; an ADC board responsible for capturing analogue signals, and a USB board which transfers the data to the host PC via USB, also using an FX2.

The ADC module implements a 12-bit 65MSPS dual-channel ADC [8] to allow for wide-band sampling of analogue inputs. The sample stream is transported to the USB module via LVDS, where it is buffered before transfer to the host PC.

FPGAs are housed on each of the modules, which provide the logic for buffering of data and flow control. Firmware developed for the FPGAs handles the data flow and debug signals, while the FX2 firmware handles the USB transfers to the host PC.

The FX2 firmware used in this project is a modified version of the open-source *C* code, as are the low-level libraries which interface with the application-level *GnuRadio* software. As the *GnuRadio* toolkit is *Linux*-based, this system was developed on an *Ubuntu 6.06 Linux* platform.

The open nature of the *Linux* development platform meant that existing code (both firmware and software) could be readliy re-used and modified, which sped up the development cycle considerably. Several proprietry applications and sources are available for *Microsoft Windows* development using the Cypress FX2 microcontroller, but these were found to have lower levels of

support and documentation, which would have led to a longer development cycle (alo considering that a large number of freely-available SDR modules have already been developed by *GnuRadio*).

The system was assembled and tested. Due to time constraints however, several limitations were imposed on the effecitve system performance, namely; the sample resolution was reduced to 8 bits to ensure data frame integrity, and the sample rate was reduced considerably to ensure that no discontinuities were introduced in the captured sample streams.

These two limitations introduced compromises in the overall system performance, but recommendations were presented to recover the losses.

Declaration

I declare that this dissertation is my own unaided work. It is being submitted for the degree of Masters in Electrical Engineering at the University of Cape Town. It has not been submitted before for any degree or examination at any other university.

....
Signature of author

Cape Town, January 2007

Acknowledgements

Firstly, I'd like to thank both my parents for all the encouragement they showed me throughout the duration of this masters, as well as putting up with my vague descriptions of the project (both real and imagined!). Shana, I hear the warm climes of Spain calling you – go for it! Sascha, Kiara – all the best guys!

I'd like also to thank Professor Inggs for his guidance, supervision, financial support and extreme patience (particularly towards the completion phase of the masters). You were certainly right about the working / studying juggling act!

Thanks also to Alan Langman for his supervision of the project, and for guidance during the design phase. A big thanks to Richard Lord for his mentorship and advice throughout.

Thanks must be made to the UCT Postgraduate Funding Office for their financial support, and in particular to Fundiswa Sayo for all her assistance.

To Jonathan Hitchcock for putting up with (and answering) all my Linux questions – thanks Shaggy!

Diglets – thanks (I guess) for giving 'New York, New York' a new meaning. To the denizens of the safehouse, thanks for all the laughs and for bearing with the 'awesome foursome' hypothesis...

To Kelly, for the sweet distractions.

Finally, thanks to everyone in the Radar & Remote Sensing Group, to my family and friends for helping make this masters experience a memorable one.

Contents

List of Figures

List of Tables

Nomenclature

ADC	-	Analogue to Digital Converter.
FIFO	-	First In First Out. This memory structure mimics a queue in that the first data item to be written to it is the first to read out.
FPGA	-	Field Programmable Gate Array. FPGA's are reconfigurable devices that allow for the synthesis of logic circuitry.
GUI	-	Graphical User Interface.
LVDS	-	Low-voltage Differential Signalling. A differential signalling standard developed by National Semiconductor.
Python	-	An open-source object-oriented interpreted programming language.
TTL	-	Transistor-transistor Logic.
USB	-	Universal Serial Bus.

1. Introduction

This thesis describes the design and implementation of a hardware platform for Software Defined Radio (SDR) applications. The system is designed to integrate with the existing open-source SDR software toolkit, *GnuRadio 2.x* (developed by the Free Software Foundation *GnuRadio* project).

SDR applications perform demodulation, modulation and other signal-processing of a digitised source signal using software modules running on a PC or other capable device. With signal-processing placed in the software domain, highly configurable and flexible DSP systems can be implemented

SDR systems typically consist of a transceiver, with a wide-band ADC for signal reception and a DAC for signal generation implemented in hardware, with signal-processing software running on a host PC.

In many cases pre- or post-processing of the data occurs in FPGA's, providing yet more flexibility to the system. In these cases, possibilities are opened up for firmware upgrades and modifications being received wirelessly allowing for self-configuring, intelligent networks.

This project aims to create a suitable platform for narrow-band SDR applications as shown in Figure 1.1.

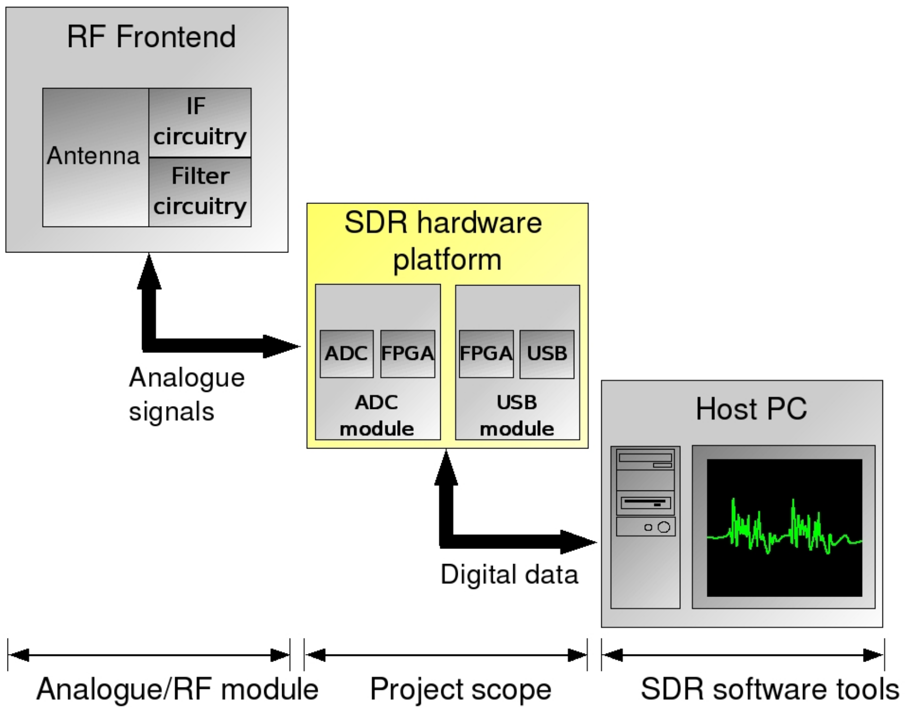


Figure 1.1. Top-level system diagram

The project scope encompassed two modules, an ADC module which is responsible for capturing analogue data and a USB module which controls the USB transactions with the host PC.

The system interfaces with a host PC via the Universal Serial Bus (USB 2.0) protocol. USB was chosen over other protocols because of its ease of use and with a view to simplify the overall system design.

While the bandwidth of USB is limited compared with other protocols (PCI for example), it is suitable for narrow-band applications.

The ADC module houses a dual ADC which samples two external signal inputs continuously. On both modules FPGA's provide the logic resources for IO and data flow control. The firmware for these FPGA's can be downloaded to the system via JTAG ports on both modules.

The system is designed to be constructed in a stackable manner as shown in Figure 1.2. This gives the design a modular architecture, which allows for changes or extensions to the system functionality to be implemented in a simple manner.

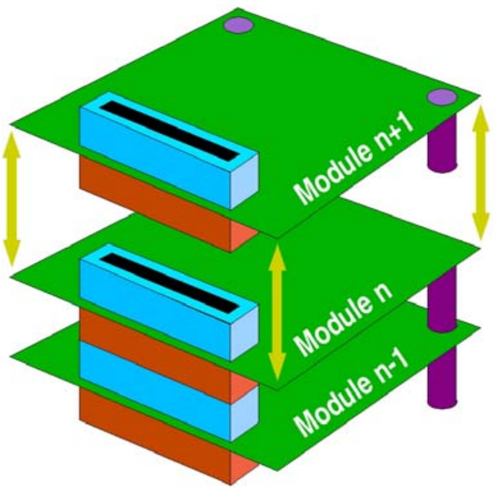


Figure 1.2. Stacked system.

A demonstration of the system's abilities to capture and display narrow-band data in real-time is shown, and a framework for future development of the project is described.

1.2. Overview

An overview of the thesis document is presented next, discussing each chapter briefly.

1.2.1. Project background

This Chapter defines what is meant by Software Defined Radio, and gives the reader some background to the field.

Traditional radio applications consist of systems which operate within fixed frequency bands, and that have pre-determined functionalities and communication protocols. This is a consequence of the fact that in general these systems are built entirely with fixed value components, with few configurable elements.

With the advent of ever-faster clock speeds of personal computers, it is becoming increasingly possible to migrate the processing of digitized signals to the software domain.

In the broadest terms, Software Defined Radio (SDR) refers to the technologies whereby a single hardware unit can receive multiple signals over a large frequency band and process these in software. SDR systems sometimes allow for software-generated signals to be transmitted on a suitable hardware platform.

The main advantages of SDR over traditional radio communications are:

- A single SDR device can perform multiple functions simply by changing software modules.
- System updates can be implemented in software to be downloaded via the transmission network. These include updates to both the software application and to any soft-configurable hardware (e.g. FPGA firmware).
- Highly configurable signal processing systems can be developed with modifications and upgrades made far simpler to implement than more traditional DSP systems
- More flexible communications protocols can be developed that adapt to their environment transparently to the system user (e.g. searching for and operating in locally available bands)

This project aims to create a hardware platform that can facilitate simple narrow-band SDR signal reception applications.

1.2.2. User Requirements

This Chapter presents a product specification for the project and lists the user requirements.

The product specification provides the design, digital, analogue and mechanical specifications that the project needs to meet.The user requirements define both the specifications of the project and any constraints that the design must fall within.

The user requirements include both functional and structural requirements as defined at the project outset.

Broadly, these include :-

- **Signal capture**
  The system must be able to digitize a signal at base-band (down-conversion and band-limiting hardware will be required for IF reception)
- **Transmission of data to host PC**
  The system must transmit narrow-band sampled data to a host PC via a USB 2.0 link in real-time.
- **FPGA configuration**
  The system must be fully configurable via the USB 2.0 link.
- **Integration of system with GnuRadio**
  The system must integrate with the existing *GnuRadio* software framework to speed the development of signal-processing applications.

1.2.3. GnuRadio

This Chapter provides an overview of the *GnuRadio* project, covering both the software framework and the Universal Software Radio Peripheral (USRP) board also developed by the *GnuRadio* team.

1.2.4. System Design

This Chapter discusses the design process of the project and takes a look at the system architecture and each of its design components.

Specifically the Chapter covers :-

- Overview of system architecture
- USB module
- ADC module
- Firmware development
- Inter-module design

1.2.5. System Implementation

This Chapter describes the implementation process of the system. This covers the following areas of the system implementation:

- **Hardware implementation**
  This Section mainly covers the PCB design process, and discusses various issues encountered during this stage of the project.

- **Software integration**
  This Section gives an overview of the *GnuRadio* toolkit framework, followed by a description of the integration of the project into this framework.

  The integration is discussed with reference to the following areas of the project:

  - FPGA firmware (both modules)
  - FX2 firmware
  - C interface with FX2 library

1.2.6. System Tests & Results

This Chapter presents the results of various tests run on the final system. These tests fall into two categories, namely;

- **Functional tests**
  These test the system to check that it meets the functional requirements specified at the project outset. In other words, they test whether the system does what is expected of it.

- **Performance tests**
  These test the analogue performance of the system.

1.2.7. Conclusions and Recommendations

This Chapter presents the conclusions drawn up and gives recommendations for future work.

Briefly the conclusions are:

- The system was demonstrated to meet its functional requirements.

- Performance tests were carried out, and while the analogue performance was inadequate for typical signal processing applications, recommendations were put forward to improve the syetm in future design revisions.

- The GnuRadio software was modified at a low level to allow for simple integration with the system hardware. While all the software in the system is developed in an open-source environment with direct access to the source code, there was a steep learning curve encountered in integrating the hardware with the software.

Based on these conclusions, the following recommendations were made:

- A list of design changes are put forward which will improve the performance of the system.

  Most notably,
    - Introducing gain to the input signal by altering the existing ADC front end circuitry
    - Using larger FPGA's to allow for more sophisticated firmware
    - Increasing the data bus width to allow for larger data rates between the modules

- Possible future projects are suggested based on the existing work.

1.2.8. Appendix A - Schematic diagrams

This appendix presents a list of the project schematic diagrams.

1.2.9. Appendix B - PCB layouts

This appendix presents a list of the layouts for both the project PCBs.

1.2.10. Appendix C - Source code listings

This appendix presents listings of the main source code elements of the system, namely :-

- FPGA firmware
- FX2 firmware
- Host PC application code

A full listing of all the source code from the system can be found in the attached CD.

1.2.11. Appendix D – Installing GnuRadio

This appendix describes how to set up *GnuRadio* on an *Ubuntu*-based Linux system.

2. Project background

This Chapter presents a background to the project and discusses the motivation behind it. Several concepts are introduced, and discussed within the context of this project.

2.1. Background

The Software Defined Radio Forum describes Software Defined Radio as:

a collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multi-mode, multi-band, multi-functional wireless devices that can be adapted, upgraded or enhanced by using software upgrades. [1]

These concepts embrace a shift in wireless systems away from predominantly hardware-based fixed-mode, fixed-band systems, to a scenario in which wireless devices can operate in multiple bands (possibly using multiple transmission protocols), possibly with multiple functions.

The devices themselves could be soft-configurable, allowing for simple changes in functionality through the changing of software modules.

For example, a European GSM SDR device may load a codec to process American GSM signals on detecting a change in the underlying service network, continuing to demodulate GSM signalling with no change in the physical hardware. This process may even be transparent to the user.[2]

One application of the above ideas is to develop a wide-band transceiver unit that can act as a bridge between analogue signals and signal-processing software running on a host PC. This application would allow for experimentation with various signal modulation techniques in software, as well as allow for the simple setup of basic RADAR applications.

Providing a platform on which to implement this or similar SDR applications is the end-goal of this project.

2.2. Project motivation

The Free Software Foundation (FSF) is maintaining a Software Defined Radio project called *GnuRadio* [3]. This GNU project is a collection of software tools that can be used to implement signal processing and radio applications on a PC using external USB-based hardware also developed as part of *GnuRadio* [3].

Maintaining GNU's free and open-source software philosophy, the *GnuRadio* toolkit is freely available, along with the source code. This was one of the main motivating factors of the master's project.

Another motivation for the project was the fact that if successful it could be implemented in various of the RADAR research applications currently being carried out by the Radar Remote Sensing Group.

2.2.1. GnuRadio toolkit

The *GnuRadio* software provides a framework with which to construct SDR applications. The framework has been designed to allow the simple implementation of signal-processing chains in graph-like structures.

The graphs can be constructed using *Python*, with the underlying processing nodes being written in *C++*. A typical processing block may be a filter, a signal source or an output stream (output is typically sent to a PC sound card).

Once the graph has been constructed and suitable hardware is up and running, the SDR application can be executed on the PC. In this way many different signal processing systems can be implemented.

2.2.2. Universal Software Radio Peripheral

The Universal Software Radio Peripheral (USRP), was developed by Matt Ettus, one of the core developers for the *GnuRadio* project [4].

It is a board containing a large FPGA, with ports for add-on RF front-ends via extension daughter-boards. The daughter-boards contain the desired signal mixing and filtering components to condition the signals entering and leaving the USRP. A fully populated URSP board houses four ADC’s and four DAC’s.

The USRP streams digitised data to a host PC via a USB 2.0 link. A DDC and other signal processing is implemented within the FPGA allowing for an effective spectral bandwidth of 6 MHz to be sustained when samples are streamed to the host PC [5]. The USRP provided a design model on which to base the current project.

The next Chapter discusses the user requirements of the master's project.

3. User Requirements

This Chapter presents a product specification, which provides electrical and mechanical specifications for the project. This is followed by a discussion of the user requirements of the project.

3.1. Product specification

A product specification for the project is shown in Table 1.

Table 1. Product specification

System requirement	**Specification category**	**Description**
Signal capture	System design	• The system must be able to capture analogue inputs and stream the samples to a host PC via USB.
Integration with existing software	System design	• The system must integrate as seamlessly as possible with existing open-source Sofrtware Defined Radio software. This will reduce firmware and host software development time as a solid base can be built upon. • To this end, the project should aim to be compliant where possible with the *GnuRadio* framework.
Modular architecture	System design	• The system should consist of module that can be combined with a common interconnection scheme allowing for future system extension. • To this end, the system must be split into two entities, a USB bridge board and an ADC capture board.
System dynamic range	Electrical (analogue)	• The system should aim to achieve a Spurious Free Dynamic Range (SFDR) of 72dBc, which is the maximum possible with a 12-bit ADC: $SFDR_{MAX,\ 12\text{-bit ADC}} = -20\log(\frac{1}{2^{12}}) \approx 72dBc$ • To this end any recommendations made by the ADC manufacturer regarding ADC front-end circuitry and layout should be followed as closely as possible.
System configuration	System design	• The system must allow the downloading of firmware via the USB port (in-system programming), as well as JTAG ports for system debugging.

Narrow band signal analysis	Electrical (digital)	• The system should implement a Digital Down Converter (DDC) in FPGA firmware to extract a signal band of interest that can safely meet the available bandwidth constraint while still guaranteeing a continuous sample stream.
System data rate	Electrical (digital)	• The system must aim to use as much of the bandwidth available to the USB 2.0 protocol (48 MB/s theoretical [12]) • To this end, a bridge device capable of supporting the USB 2.0 protocol must be implemented to provide an interface between the host and the system.
PCB form factor	Mechanical	• The system must conform to the industry-standard PC-104 form factor (95.89 mm x 90.17mm) [13]. • This provides the opportunity for use of standard PC/104 mechanical housings, and presents a compact solution.
Module height	Mechanical	• The maximum component height of each of the system modules must be minimised (not to exceed 20mm). • This keeps the system compact, and lessens the constraint on board interconnect components.

The product specifications listed above can be divided into two categories, namely :-

- **Functional requirements**
  These define what the system must do. i.e. what functions the system is able to carry out.

- **Structural requirements**
  These define any constraints that must be met during the construction of the system.
  i.e. any constraints to the physical structure and architecture of the system.

The functional requirements of the system are listed overleaf.

3.2. Functional requirements

In order to facilitate SDR functionality, the system must be capable of the following functions:

- **Analogue signal capture**
  The system must digitise an analogue input signal with a suitable bandwidth and bit resolution. The system's dynamic range must match the ADC's listed performance as closely as possible.

- **Transmission of data to a host PC**
  The captured data must be sent to a host PC at a suitable data rate for the application.

- **Interaction with existing SDR software tools**
  The system must interface with existing SDR software tools.

- **Inter-module communications**
  The system is to have a modular architecture. A suitable backplane and data transport protocol need to be selected to support the data rate requirements.

- **Application flexibility**
  Where possible, the system must be made as flexible as possible in terms of which applications may be executed. The various system configurations must be easily selected.

3.3. Structural requirements

Two main structural requirements were defined at the outset of the project, namely:

- **PCB form factor**
  If possible an industry standard PCB form factor should be used for the system, with an emphasis in reducing the size of the final board layout. This requirement is driven by the need to keep manufacturing costs down.

- **System modularity**
  The system must be constructed of functional modules, which can be easily upgraded or replaced.

3.4. GnuRadio

A fundamental requirement of this project is that it must integrate with the existing *GnuRadio* software framework.

Before the system design is discussed, an overview of the *GnuRadio* open-source SDR project is presented.

GnuRadio is an open-source project developed using free tools under the *Linux* operating system. The source code is freely available, with support available through forums and mailing lists.

This made it an ideal choice for research work into software-radio applications.

3.4.1. GnuRadio structure

As shown in Figure 3.1, the *GnuRadio* framework consists of a graph structure made up of signal processing blocks linking signal flow paths.

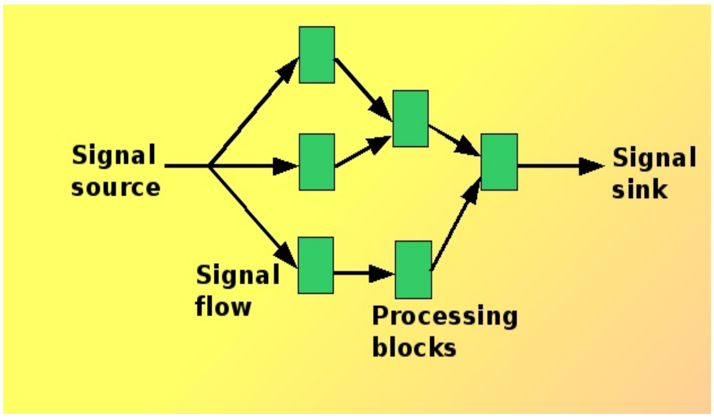


Figure 3.1. GnuRadio abstraction

The signal source can have various implementations, but is usually a USB-based hardware device (specifically USRP or SDR-Module) or a file containing sampled data. The signal stream then flows through a graph made up of processing nodes.

Many processing blocks currently exist within the *GnuRadio* framework, and include filters, demodulators and other signal manipulation elements. The processing blocks are written in *C++*, and are each defined to have input or output ports (or both). The blocks are linked together with *Python*, which also controls the data flow rates between blocks.

The final output of the processing graph terminates in a signal sink, which is typically a graphical representation of the FFT of some sampled data (real-time oscilloscope), or an output file.

This framework allows for simple implementation of powerful signal processing systems.

3.4.2. Universal Software Radio Peripheral

The USRP was designed by the *GnuRadio* group specifically to interface with the *GnuRadio* software. It is a hardware platform which provides the capability to stream data to or from a host PC via USB 2.0.

It houses a large FPGA which coordinates data transfers from or to on-board ADC's and DAC's. Daughter-boards can be attached to the USRP which implement RF frontends.

In addition, the FPGA can perform down-conversion and decimation of wide-band sampled data to enable extraction of a narrow-band for practical transmission over USB.

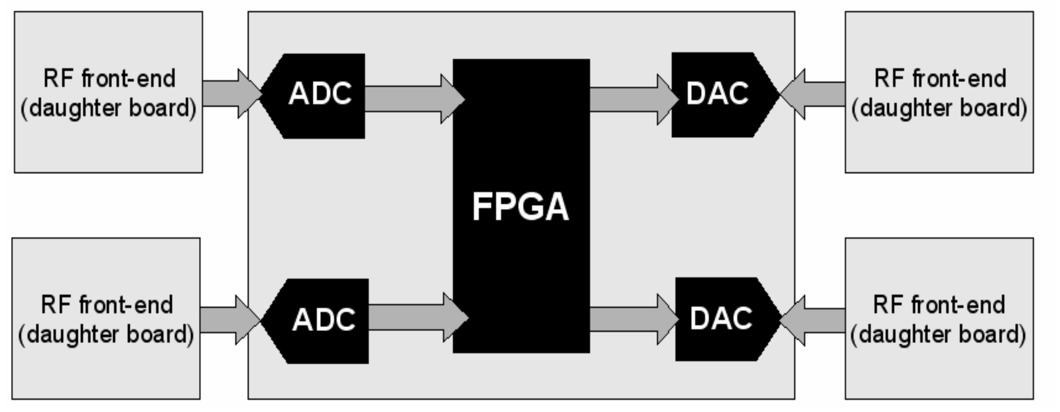

RF front-end (daughter board)
ADC
FPGA
DAC
RF front-end (daughter board)
RF front-end (daughter board)
ADC
DAC
RF front-end (daughter board)

Figure 3.2. USRP block diagram

The functional blocks of the USRP are shown in Figure 3.2. The key blocks are discussed in the Section 3.3.3.1.

3.4.2.1. USRP Hardware

The central control of the system is handled by a large FPGA housed on the main USRP board. This FPGA responds to data transfer requests passed from the FX2 USB microcontroller. It streams down- converted data received from one or more ADC's, and sends sampled data for transmission via the DAC's.

The FX2 microcontroller contains an embedded USB 2.0 transceiver and handles all USB transfers with the upstream USB host. It presents a data bus to the outside world (in this case the FPGA), with generic control signals which can be programmed to behave in a custom manner. This interface is called the GPIF (General Purpose Interface), and is covered in more detail in Section 3.3.3.4.

The FX2 also handles all USB control requests (via *endpoint 0*), which all USB-enabled devices must support to fully comply with the USB standard. These include responses to device capability interrogations and standard setup requests.

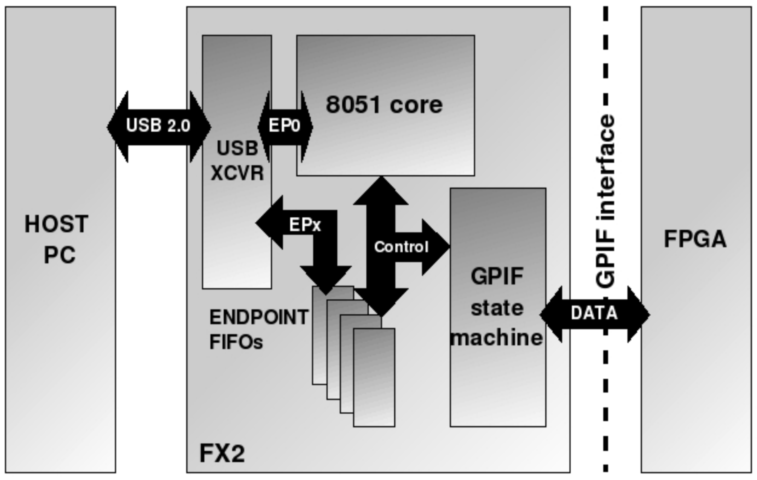


Figure 3.3. FX2 internal block diagram.

A simplified view of the FX2 is shown in Figure 3.3. It houses an industry-standard 8051 micro-controller core (with several extensions and enhancements), which handles all internal control. The 8051 initialises the transceiver, which handles the actual USB transactions. It is also responsible for configuring the FX2's general-purpose I/O ports (not shown) and the GPIF state machine. [6]

Data is transferred in a USB system via endpoints, which are similar to *Ethernet* network socket. Each endpoint must have a specified data-flow direction, either IN or OUT (USB-centric), except the control *endpoint 0* which is bidirectional. The endpoints, must also have defined data transaction types which indicate their bandwidth requirements.

Data entering or leaving the FX2 on the USB host side is stored in the endpoint FIFO's, which can be configured to have various sizes and levels of buffering. The GPIF has direct access to these FIFO's, allowing for seamless data transfer between an external device and the USB host through a series of buffered FIFO's.

3.4.2.2. FX2 firmware

As well as general house-keeping and configuration, the FX2 firmware is responsible for the following areas:

- **GPIF initialization**
  Rather than attempt to handle data transfers directly at USB 2.0 high-speed (480 MBits/s), data transfers are carried out by the GPIF. The FX2's 8051 core initialises this interface.

- **USB control request handling**
  The 8051 core responds to control requests sent by the USB host over *endpoint 0*. All USB-compliant devices must return responses in a pre-determined format to standard requests about the device (e.g. *GetDeviceName* which returns the device's description).

- **USB transfer requests**
  These are implemented as a polling loop that loads GPIF setup registers with the transfer parameters as and when requests are received from the host.

The FX2 firmware is written in *C*, and compiled with the open-source compiler, *SDCC* (Small Device Cross Compiler). The embedded functions are invoked using the *usrp_basic* and *ursp_prims* libraries, as discussed in Section 3.3.4.

3.4.2.3. FPGA firmware

Figure 3.4 shows an overview of the USRP's FPGA internal block diagram.

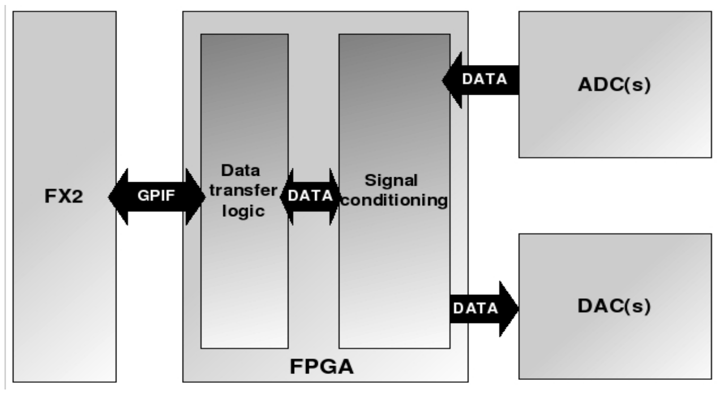


Figure 3.4. USRP FPGA firmware block diagram.

The firmware handles the digital down conversion of the ADC input by mixing, filtering and decimating the sample stream. It also handles the pre-processing of DAC samples before they are sent for transmission to the DAC daughter-boards. Clock domain crossing and alignment are also handled by the FPGA.

A key functionality of the FPGA is to interface with FX2. This interface is covered in the next Section.

3.4.2.4. General Purpose Interface

The GPIF is a mechanism implemented by the FX2 to allow for simple interfacing with many different types of devices. Essentially it is a bi-directional data bus with a set of generic control signals which are governed by a configurable state machine.

The GPIF can be configured to be the interface bus master or can be driven as a slave by the device itself. In both the USRP and this masters project, the GPIF is the bus master.

Six states can be defined for a particular waveform, with decision points that dictate the state transitions depending on the values of the generic control lines.

There are four bus cycle waveforms available which can be configured. These are :-

- **Single Write**
  This bus cycle writes a single data word from the USB to the device (all transactions are defined to be USB-centric)

- **Single Read**
  This cycle reads a single word from the device.

- **FIFO Write**
  This cycle writes a block of data based on some previously setup parameters. During the transfer state, data flow can be throttled by the receiving device.

- **FIFO Read**
  This cycle reads a block of data from the device, and is used to stream ADC data up to the host.

An example of a simple GPIF FIFO read waveform is shown in Figure 3.5.

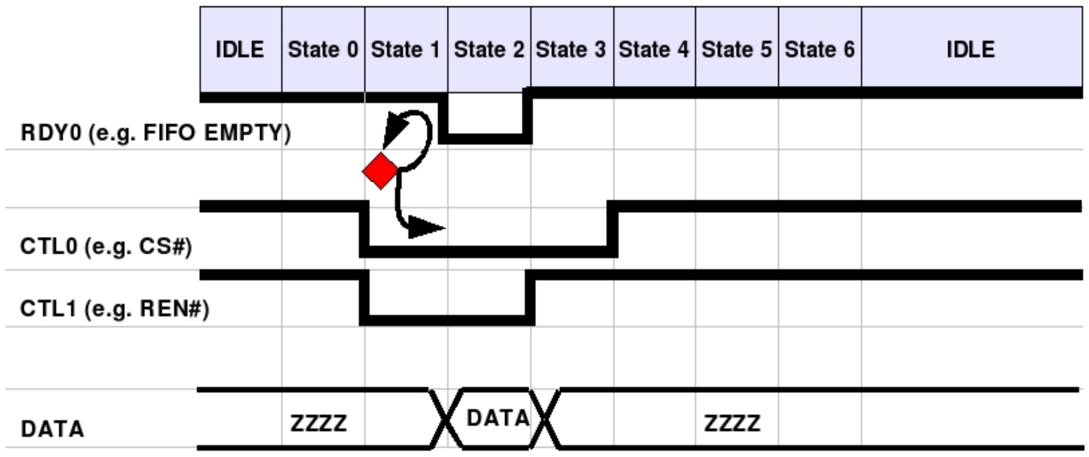


Figure 3.5. Example GPIF waveform

The GPIF has six control inputs *RDY5:0*, and six control outputs *CTL5:0*. These can be used to implement a great variety of standard and proprietary bus control cycles.

In Figure 3.5, the decision point in state 1 will hold the cycle in a stalled state until *RDY0* (in this case representing *FIFO Empty Flag*) is deasserted when the GPIF state machine will advance to state 2 and latch in the data presented on the *DATA* bus.

The FX2 implements the USB endpoints as internal endpoints *EP0, EP1, EP2, EP4, EP6* and *EP8*, which can be configured in various ways to suit the application's buffering requirements. *EP0* is always a 64 byte *CONTROL* endpoint and *EP1* may only be of the *BULK* or *INTERRUPT* types (also 64 bytes).

The remaining endpoint FIFO's can be configured to be double-, triple- or quad-buffered, with a maximum total buffer size of 4K. In this manner, data can be continually streamed to or from the host, provided adequate flow control is in place.

Cypress Semiconductor has developed *GPIF Designer*, a freely available application which can be used to design GPIF state machines with a graphical interface. The application allows a user to customise the state machines, after which the appropriate configuration and initialization source code is generated which can be included with the main FX2 firmware. This simplifies the process of configuring the GPIF waveform registers which would otherwise have to be coded manually.

In the USRP's case, the developers chose to parse the *GPIF Designer's* output code, and to insert their own initialization routines. This was done in part because the output code generated by *GPIF Designer* is somewhat ambiguous (bugs have been encountered by the *GnuRadio* team [3]).

3.4.3. USRP interface

As mentioned previously, the FX2 firmware handles standard USB control requests, sets up the GPIF and performs general housekeeping. It also sets up interrupt handling routines which service the USB requests from the host.

These requests are made by application software through calls to the USB driver, and include such methods as USB initialization, renumeration and basic input and output routines.

Bus renumeration is the process whereby the FX2 boots up with a hard-coded USB product and vendor ID which is recognized by the host PC during its USB bus enumeration. This USB identity is limited to simple diagnostic functions, and the ability to download firmware to the device. Once the firmware has been downloaded to the FX2, it sets an internal register and reboots itself, this time presenting custom product and vendor ID's, which the host detects as the FX2 disconnecting followed by the connection of a custom USB device.

Internally, the GPIF acts autonomously from the actual FX2 8051 core and provides data to and from the internal endpoint FIFO's for streaming up to and down from the USB host.

The USRP interface at the application level is defined and implemented in the *usrp_prims* and *usrp_basic* source and header files.

Usrp_prims defines a set of functions implemented on the FX2 in response to control requests via USB *endpoint 0* that extend the generic USB function set. These function primitives typically write to or read from various FX2 registers, or initiate data transfers via the GPIF.

Usrp_basic provides a simple API to the *usrp_prims* functions in the form of an object-oriented structure. The interface to the system provides higher-level functionality, hiding most of the underlying interface mechanisms.

3.4.4. Application-level software

The data streaming test application is *usrper* which can load FPGA configuration data (fed serially into the configuration lines of the FPGA bitwise), assert debug LEDs, read and write data to the USRP.

3.4.4.1. USRPER test application

On start up, *usrper* first attempts to create an interface object that can communicate with a valid USRP device. This creates a handle to a generic USB device, and then traverses all the devices found attached to the USB, searching for the FX2's vendor and product ID numbers (*0x04B4* and *0x8613* respectively).

On successfully locating a powered, unconfigured FX2 device, *usrper* then downloads the *GnuRadio* firmware to the FX2, which sets up the system. After system setup, the firmware loads custom values into the Vendor and Product ID fields and initiates a software reset, causing the FX2 device to disconnect and then reconnect to the USB bus.

This process is called renumeration, and the *Linux* kernel now sees the USRP board as a USB device with the Product ID and Vendor ID *0xFFFE* and *0x0002*, which correspond to a configured USRP device developed by the Free Software Foundation.

Once this has been achieved, calls are made to USRP-specific methods which can be reads, writes or control requests.

- **Control requests**
  These enable or configure various aspects of the USRP system, and are made to USB endpoint 0. The firmware intercepts these calls (interrupts are generated on all USB control requests that are received from the host) and services them.

  They form vendor-extension functions, and carry out various data transfers to and from internal registers based on the host's control request parameters.

  For example to download the FPGA's firmware, the configuration bitstream is sent via endpoint 0 with the appropriate control parameters indicating a request to configure the FPGA. The FX2 intercepts each byte in this stream, and sends it bitwise over the FPGA's configuration lines (the Altera FPGA only supports serial configuration modes). While servicing the control request, the FX2 firmware sends back the appropriate handshaking signals to the host to indicate that the control request is being handled correctly.

- **Read requests**
  The USB requests data from the USRP by making read requests over endpoint 2 (set up for 512 byte bulk IN transfers). The FX2 intercepts these requests and primes the GPIF transfer state machine with the transfer parameters before relinquishing control to the GPIF. The GPIF enters the FIFO read state machine and remains in this configuration until the transfer is complete, or an error has occurred. The FX2 handles all acknowledge and handshaking signals with the USB host transparently from the firmware.

- **Write requests**
  As with read requests, the host PC makes generic USB write requests, this time to endpoint 6 (configured for 512 byte bulk OUT transfers). The GPIF is this time set up for the FIFO write state machine, and transfers data until the transaction is complete, or an error is encountered.

This masters project modified various aspects of the software framework to allow for the new hardware to be accessed.

Briefly, the modifications were:

- The FX2 firmware was modified to handle custom product and vendor IDs, flow-controlled streaming of data to the host.

- The low-level *C* libraries were modified to recognize the custom hardware, and to handle the incoming data.

- The test application was modified to store the captured data in a custom format for off-line analysis.

These modifications are discussed in Section 5.2.

This Chapter provided a brief background to the *GnuRadio* project, and discussed the USRP and the FX2 USB microcontroller. The next Chapter covers the masters project system design.

4. System Design

The system was designed to meet the product specification and user requirements set out in Chapter 3. This Chapter first gives an overview of the system architecture followed by a discussion of each of the design components.

4.1. System architecture

The first requirement of any SDR system is to be able to convert signals between analogue and digital domains.

Recalling Figure 1.1, the system provides a bridge between an analogue signal source and a host PC running SDR software tools. This allows for a suitably band-limited signal to be processed on the host PC.

The system architecture is modelled on the USRP system.

4.1.1. Architecture overview

A photograph of the assembled system is presented in Figure 4.1, showing the ADC module (the USB module is beneath this).

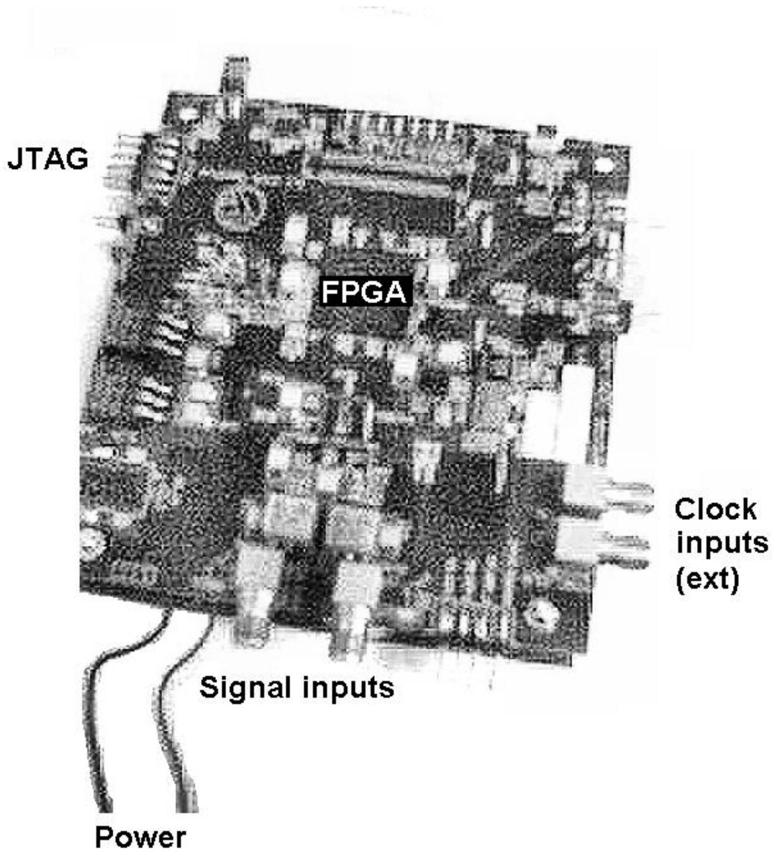


Figure 4.1. Assembled system

The system is divided into two modules, namely the ADC module and the USB module, which is shown in Figure 4.2.

The ADC module samples two analogue input channels and interfaces with an FPGA. External timing signals may also be applied to the ADC to allow for various system timing modes.

The USB module consists of an FPGA and an FX2. The two devices communicate via the 16-bit GPIF interface.

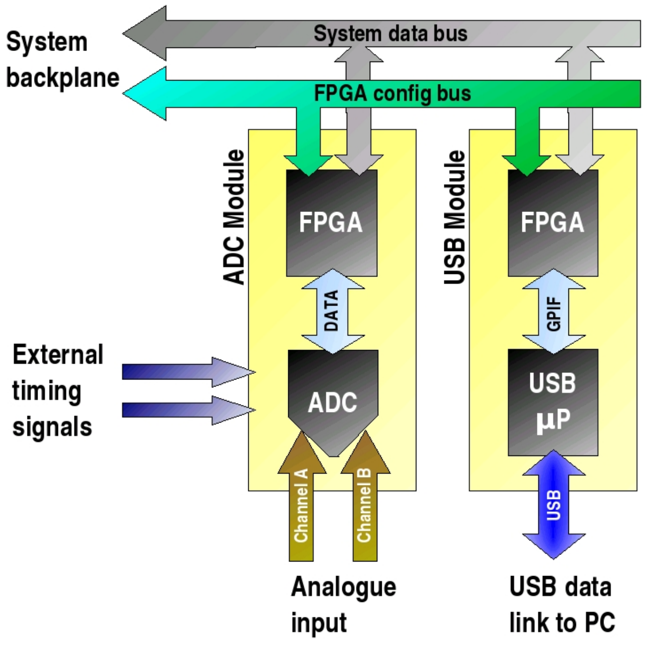


Figure 4.2. Basic system architecture

The two modules are connected via a system-wide backplane. The backplane consists of a data bus, FPGA configuration lines and various other signals which are discussed in detail in Section 4.3.2.

4.1.2. System modularity

One of the user requirements of the system was a modular construction. Each module of the system is divided into a functional unit as shown in Figure 4.3.

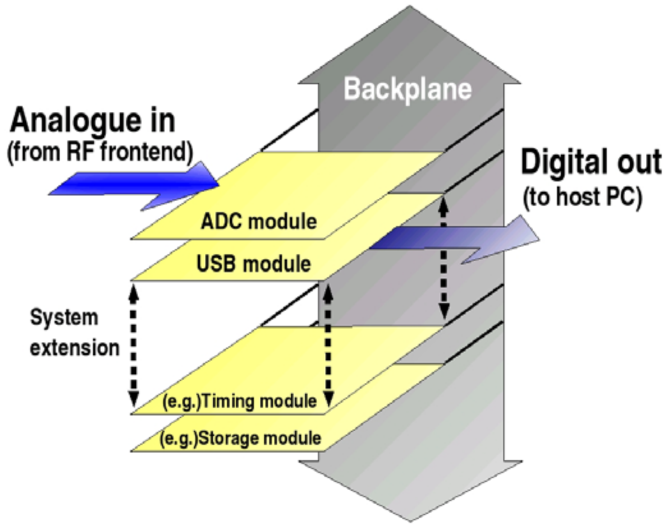


Figure 4.3. System modularity

The project scope encompassed an ADC module to capture analogue signals, and a USB module to transmit the captured data via a USB link to a host PC. Examples of future possible extension modules are also shown in the Figure.

4.2. FPGA configuration

For the system to function, both FPGA's need to be configured. The system was designed to support two methods of downloading the FPGA firmware, via the USB link through the FX2 in *Slave Parallel* mode and independently through each FPGA's JTAG port.

4.2.1. Slave Parallel configuration mode

The Spartan-3 FPGA supports various configuration schemes, broadly split into serial and parallel data transfer modes [7]. For optimum configuration speeds, the *Slave Parallel* configuration mode is implemented in this design.

This mode reads in a configuration byte every clock cycle, where the clock is generated by an external master (the FX2 micro-controller).

The configuration scheme is shown in Figure 4.4, with a description of each of the signals given in Table 2.

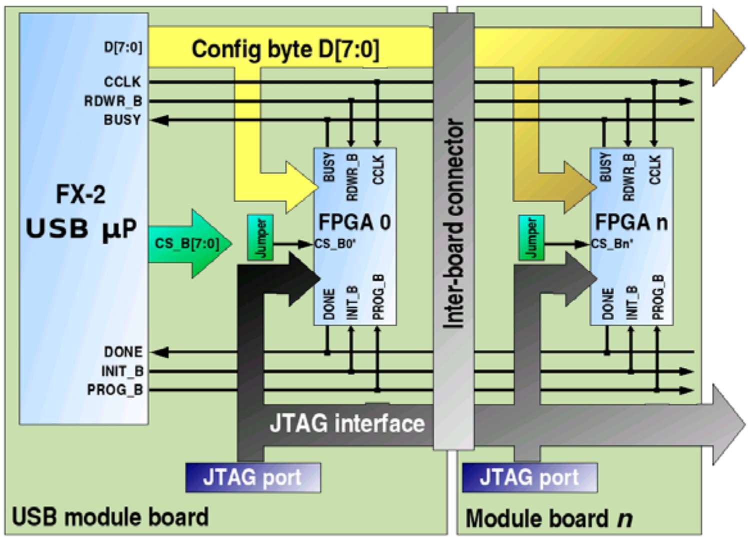


Figure 4.4. FPGA configuration scheme

Table 2. FPGA Configuration signals

Signal Name	**Signal Description**
D[7:0]	Configuration data byte
CCLK	Configuration clock (data transferred on rising edge)
RDWR_B	Read Enable (Write on signal assertion)
BUSY	Data throttle (FPGA throttles data when asserted)
CS_Bx	Chip Select
PROG_B	Low-going pulse resets/initialises the configuration process
DONE	Low-to-high transition indicates completion of the configuration process.
INIT_B	Low-to-high transition indicates that the configuration memory is clear and the configuration process can begin.
M[2:0]	Configuration mode selectors JTAG - 101 Slave Parallel - 110

A low-going pulse on *PROG_B* initiates the configuration process. Following this, any chained FPGA's in the system clear their configuration memories. The end of the clearing process is signalled with a low-to-high transition on the *INIT_B* line.

INIT_B is tied together for all FPGA's in the system, meaning that the process is stalled until all FPGA's have released the line and are ready to receive configuration data.

The next step is to download the configuration bitstreams to each of the FPGA's. This is synchronous with the rising edge of *CCLK*, with the *RDWR_B* and *CS_Bx* selecting the destination FPGA.

The FX2 micro-controller can address eight FPGA's directly with the *CS_B[7:0]* signals. These traverse the backplane and are tapped off on each board via a system jumper.

If any of the FPGA's needs to throttle the configuration data rate, the shared *BUSY* line is asserted which results in the FX2 waiting until the line is released.

Once the configuration process is complete, the DONE line undergoes a low-to-high transition. This line is tied to all FPGA's in the chain and is therefore only released when all FPGA's have been configured.

4.2.2. JTAG configuration mode

Recalling Figure 4.4, there is a JTAG interface to all the FPGA's which traverses the backplane. JTAG allows an alternative means of configuring the FPGA's in the system and is useful for debugging purposes. The configuration mode is selected with system jumpers.

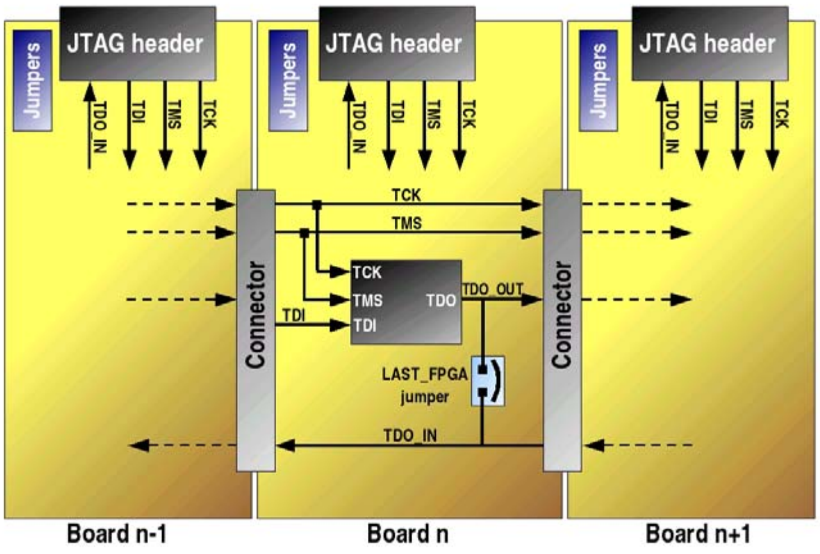


Figure 4.5. JTAG Configuration

In Figure 4.5, the JTAG signal layout is shown in more detail. *TCK* is the clock signal for all JTAG operations, and bits are read in serially on the rising edge of this signal. Based on the value of *TMS* and *TDI*, different operations can be carried out.

Table 3. JTAG signals

Signal Name	**Signal Description**
TCK	Test Clock – This signal synchronizes all JTAG operations
TDI	Test Data Input – JTAG serial data input
TMS	Test Mode Select – This pin selects the JTAG operation
TDO	Test Data Output – JTAG serial data output

Once an FPGA has been configured, all data presented on its *TDI* input is passed through to its *TDO* output. This means that multiple FPGA's can be daisy-chained together to be configured sequentially.

The JTAG protocol requires a returning signal *TDO* to complete the serial datapath. As the system topology can be extended, a loop-back jumper (*LAST_FPGA*) must be in place on the final board of the chain.

The next Sections discuss each module in some detail.

4.3. ADC Module

The ADC module is responsible for digitising analogue input signals and transmitting them across the backplane to the relevant communications module (in this case the USB module).

It consists of an ADC, an FPGA and supporting devices and circuitry, as shown in Figure 4.6.

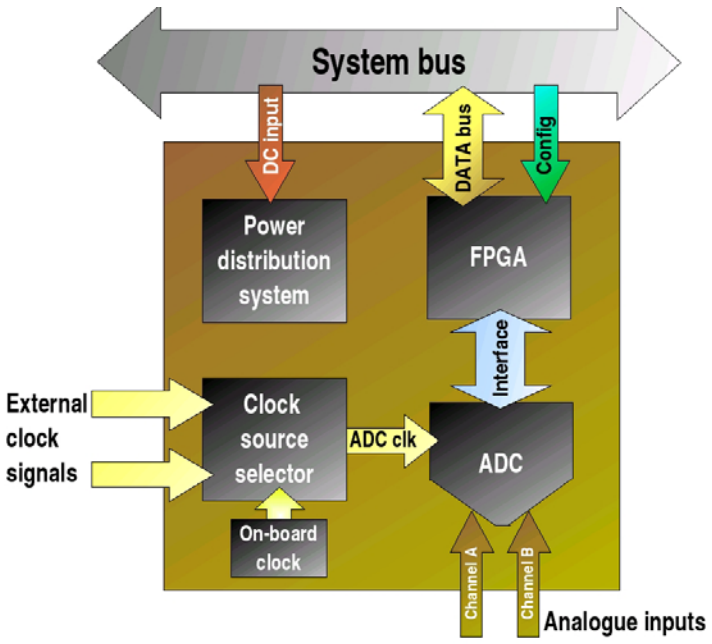


Figure 4.6. ADC Module

4.3.1. ADC

The ADC implemented on the ADC module is the AD9238 from Analog Devices. It is a dual channel ADC with differential inputs. As the two ADC cores are embedded within the IC itself, there is superior cross-talk isolation between the channels.

There are three speed-grades of the AD9238; 20, 40 and 65 MSamples/second. The 65MSPS version is implemented in this design to achieve wideband sampling capability.

4.3.1.1. ADC inputs

The AD9238 has differential inputs which improve noise reduction due to the common-mode rejection capabilities of the ADC [8].

As shown in Figure 4.7, a transformer is used on each channel convert single-ended analogue inputs to a differential signalling. The input impedance to each of the transformers is 50Ω, and the single-ended inputs enter the system via SMA connectors.

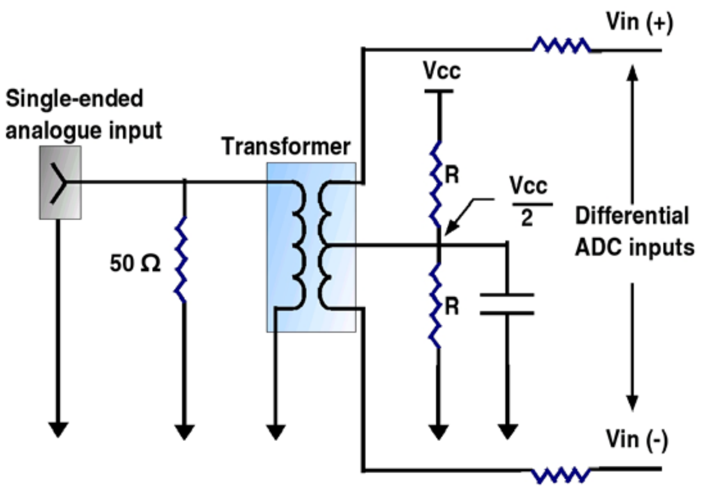


Figure 4.7. ADC inputs

This system uses decimation to reduce the sampling rate by a user-specified factor (discussed in Section 5.3). Thus the bandwidth of the system is reduced. In the test case, the decimation factor was set to 8, resulting in an effective bandwidth of 4MHz.

It is the responsibility of the provider of the signal source to band-limit the signal appropriately (4MHz cut-off for a decimation factor of 8), as no filtering circuitry is in place on the ADC module board. Provided the input signal is band-limited to prevent aliasing, Nyquist sampling could capture signals at other intermediate frequencies (IF's) though this has not been tested.

4.3.1.2. ADC clock source selection

The 65MSPS version of the AD9238 family was chosen to provide optimum system bandwidth. A fixed oscillator running at 64MHz is installed on the ADC module board. Some applications may require slower sampling rates, or user-controllable sampling signals, so a clock multiplexer arrangement has been implemented as shown in Figure 4.8.

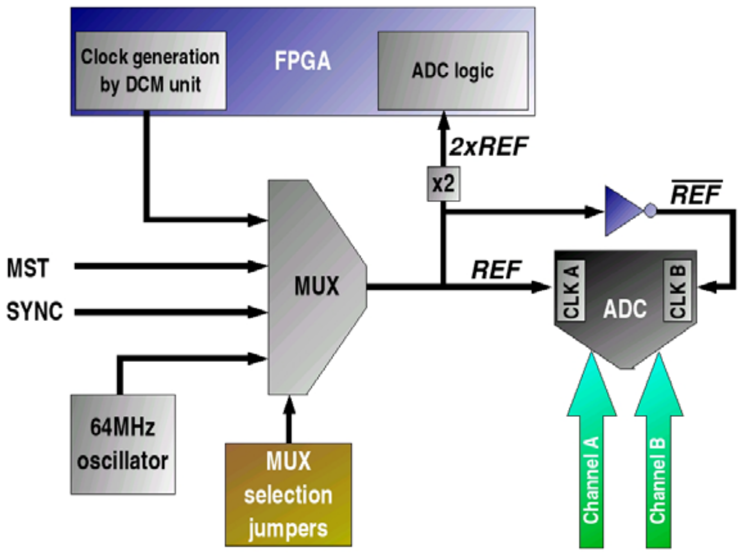


Figure 4.8. ADC sample clock sources

The multiplexer accepts four clock sources:

- **FPGA-generated clock signal**
  This signal is generated by a Digital Clock Management (DCM) unit within the FPGA. The frequency and phase of this signal can be altered in the FPGA firmware, allowing for a custom sampling clock signal to be generated.

- **MST clock source**
  This is an externally provided master clock source.

- **SYNC clock source**
  This is an externally provided synchronisation clock source generated elsewhere in the system. One application of SYNC may be as a system event trigger.

- **On-board 64MHz oscillator**
  This is the default clock source generated by the on-board oscillator.

All the clock signals must be in the range of 1 to 65MHz as per the AD9238 datasheet.[8]

The output of the multiplexer *REF*, is fed into a clock multiplier IC (not shown) which generates *2xREF* and *nREF* signals. The *2xREF* signal drives the FPGA interface with the ADC and means that the ADC logic can optionally have two clock cycles for every ADC sampling cycle. The FPGA-ADC interface is discussed in Section 4.3.2.

In the routing phase of the project, the net lengths of the various clock signals were equalised to within the nearest millimetre to reduce phase differences. In addition, all corners on all the clock nets were rounded to reduce signal reflections and echoes.

4.3.2. FPGA-ADC interface

The AD9238 is a 12-bit dual ADC with parallel output and therefore requires two 12-bit data buses. The design was created with the 14-bit AD9248 in mind, so two 14-bit data buses have been implemented (the AD9238 only supports a 12-bit data bus, but is pin-compatible with the AD9248).

In addition to the data lines, the ADC has several other signals that are input to the FPGA. The ADC-FPGA interface signals are shown in Table 4.

Table 4. ADC-FPGA interface signals

Signal name	**Signal description**
ADC_A[13:0]	14-bit data bus for ADC channel A
ADC_B[13:0]	14-bit data bus for ADC channel B
PDWN_A	Channel A power down
PDWN_B	Channel B power down
OEB_A	Channel A output enable
OEB_B	Channel B output enable
OTR_A	Out of range indicator for channel A
OTR_B	Out of range indicator for channel B

The internal structure of the FPGA firmware implemented on the ADC module is shown in Figure 4.9.

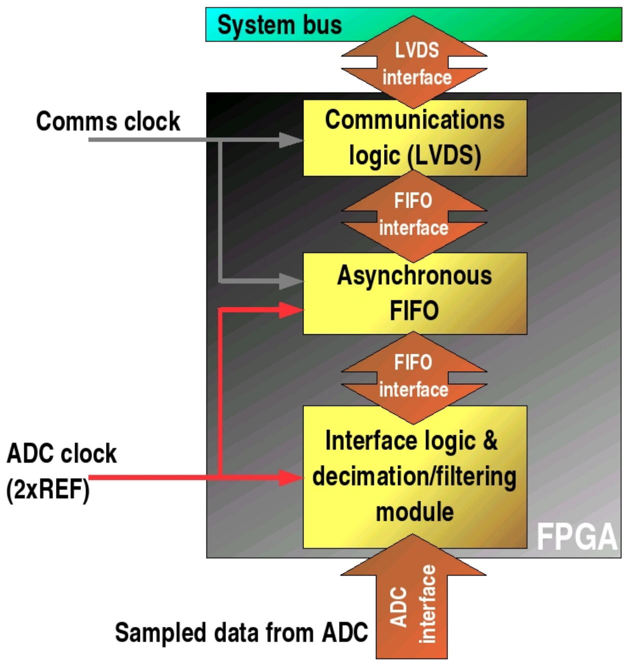


Figure 4.9. ADC Module FPGA firmware structure

As the ADC interface logic can be driven at variable clock speeds (depending on the value of the ADC clock as in Section 4.3.1.2), an asynchronous FIFO is required to cross the data into the communications clock domain.

With a sample rate of 64MHz and dual channels capturing data with 12-bit resolution, the maximum output data rate of the ADC module is 256 MB/s (12 bit samples zero-padded to 16-bit words). The greatly exceeds the theoretical maximum USB 2.0 data rate of 48MB/s (disregarding control overhead, the USB rate is less than this).

Thus, the raw sample rate must be reduced to ensure continuous signal capture. It was originally intended to implement a simple DDC within the FPGA which would have allowed for narrow-band analysis of the wide-band input and reduced the module's output rate, but there were insufficient logic resources to put this in place. Instead, sample decimation is used to reduce the data rate.

Table 5 indicates some of the achievable data rates with the system. Note that when sampling with both channels, the samples are interleaved and thus the bandwidth per channel is halved.

Table 5. System data rates

ADC clock rate	Decimation factor	Sample size	Maximum data rate to host	Comment
64 MHz	None	12-bit padded (16-bits)	48MB/s (theoretical)	DDR 8-bit transfers to USB module (internal logic running twice sampling rate) Transfer rate is limited by USB 2.0
32 MHz (effective)	None	12-bit padded (16-bits)	32MB/s	8-bit transfers every clock cycle at 64MHz (effective ADC sample rate 32MHz) ADC module is limiting data rate
64 MHz	2	12-bit padded (16-bits)	16MB/s	- As above but every second sample is discarded, and two clock cycles per sample
64 MHz	2	8-bit samples	32MB/s	- As above but only 8 MSBs of each sample are transferred (loss of sample resolution). One sample transferred per clock cycle
64 MHz	8	8-bit samples	8MB/s	- As above but the effective sample rate is reduced to 8 MHz

This system was tested with the final option in Table 5 as discussed in Section 5.2.

4.3.3. Inter-module communications logic

The inter-module communication is handled by the communications logic unit within the FPGA. The project has been designed to use Low Voltage Differential Signalling (LVDS) between the modules, which has good noise-rejection properties and can support high data rates.

4.3.4. USB Module

The USB module is responsible for controlling all communications with the host PC via a USB 2.0 link. It also transfers FPGA configuration data to all the FPGA's at system start up.

A functional block diagram of the USB module is shown in Figure 4.10.

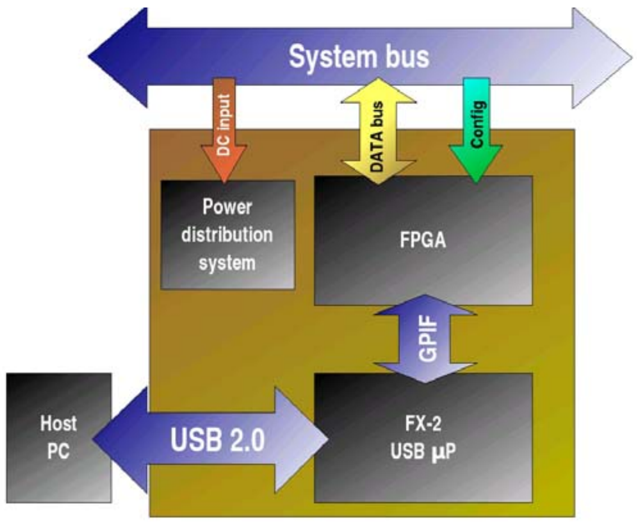


Figure 4.10. USB Module block diagram

4.3.5. USB Micro-controller

On-board the USB module is an FX2 micro-controller with an embedded USB 2.0 transceiver from Cypress Semiconductor. The FX2 handles all communication with the USB host, and was discussed in detail in Section 3.3.3.1.

The firmware executed on the FX2 is a modified version of the USRP's firmware code. The differences are:

Custom vendor ID is loaded. The vendor ID used to identify this master's project is *0x0099*.

FPGA reset function uses a different FX2 pin, and is called after the FX2 has initialised itself.

The USB module was routed to be able to support both GPIF modes, master and slave. The master mode is implemented in this project (as per USRP setup). The GPIF slave mode is discussed next briefly for reference.

4.3.5.1. GPIF Slave mode

The USB module is responsible for controlling all communications with the host PC via a USB 2.0 link. It also transfers FPGA configuration data to all the FPGA's at system start up.

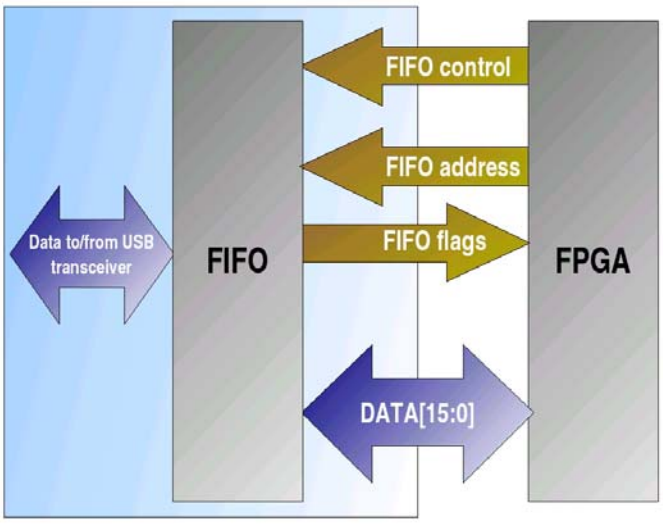


Figure 4.11. GPIF slave mode interface

In the GPIF slave mode, the control of the FIFO is assigned to an external device (in this case the FPGA).

As shown in Figure 4.11, the FPGA supplies control and addressing signals to the FIFO, which optionally returns flags. The signals are listed in Table 6 (the flag signals are shared with the *RDY* lines of the GPIF interface).

Table 6. GPIF slave interface signals

Signal name	Signal description
nSLCS	Slave Chip Select
SLRD	Slave Read
SLWR	Slave Write
SLOE	Slave Output Enable
PKTEND	Packet End (forces a non-standard number of words to be committed as a USB packet)
FIFOADR[1:0]	FIFO address (FIFO select)

The FPGA treats the FIFO as a slave device and by asserting the control signals appropriately can initiate data transfers to and from the FIFO.

4.3.5.2. FX2-FPGA interface firmware

The internal firmware for the FPGA on the USB module is shown in Figure 4.12.

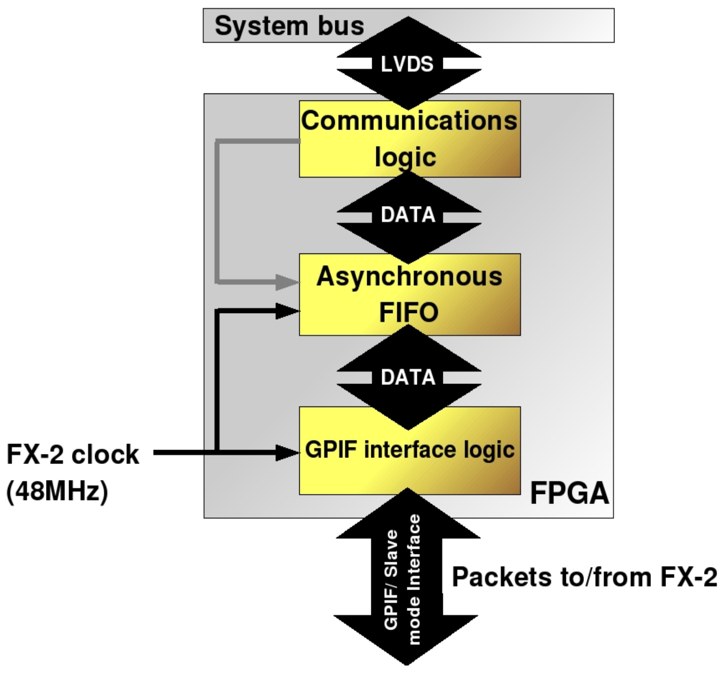


Figure 4.12. USB module firmware

The USB module's FPGA firmware is similar to the ADC module firmware in that there is an asynchronous FIFO linking the LVDS communications unit with the rest of the datapath.

4.4. Power distribution system

Several voltage supplies are required for the Spartan-3 FPGA. In addition to these supply voltage levels, analogue 3.3V supplies are required for both the ADC (ADC module) and for the *FX2*'s USB transceiver (USB module).

The DC voltage input to the system is carried across the backplane, and tapped off on each module as required. The power bus can carry a maximum of 4A of current.

The DC voltage is regulated on each module to provide the required voltage levels. These supplies are listed below:

- **1.2V**
  - FPGA Core voltage (VCCINT).

- **2.5V**
  - Supply voltage for auxiliary FPGA units like DCM's (VCCAUX).
  - Supply for LVDS output drivers on selected FPGA IO banks.

- **3.3V (digital)**
  - Supply voltage for LVTTL output drivers on selected FPGA IO banks
  - Supply voltage for all remaining IC's on the boards.

- **3.3V (analogue)**
  - Supply voltage for the ADC's analogue components (ADC module)
  - Supply voltage for the FX2's embedded USB transceiver (USB module).

The analogue supply planes are isolated from their corresponding 3.3V digital planes via balun transformers to reduce signal cross-talk between the digital and analogue signal domains.

All the boards' power supplies are implemented as split supply planes, as detailed in Section 5.1.3.

4.5. System backplane

The backplane is the set of signals shared across the system. These include data, control and timing signals as well as power signals.

Efforts were made during the routing process to ensure that there is minimal cross-talk between each of the signals, and that high speed nets were of the same track length which reduces relative delay between adjacent signals.

4.5.1. Backplane architecture

To facilitate the modularity requirement in the system structure, a backplane needed to be designed so as to have the following properties:

The modules must be freely interchangeable with no effect on the overall system operation or performance.

The backplane must be capable of achieving high data rates between the modules. In order to meet these two requirements, a point-to-point bus topology was chosen as shown in Figure 4.13.

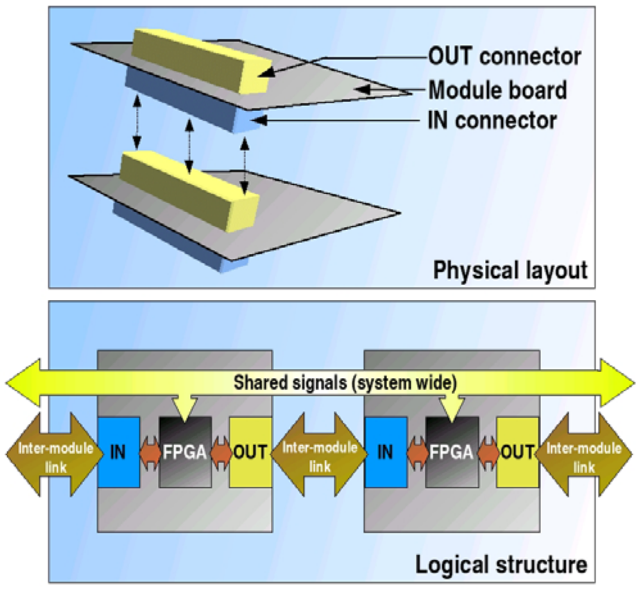


Figure 4.13. Backplane architecture

The inter-module links are divided into IN and OUT groupings and are physically separated on each module. Other signals on the backplane are shared across the system (for example the FPGA configuration clock, CCLK).

4.5.2. Backplane signal assignments

The backplane needs to transport four signal categories:

- FPGA configuration signals
- Inter-module data signals
- DC power signals
- Debugging / IO signals

The backplane signal assignments are shown in Figure 4.14.

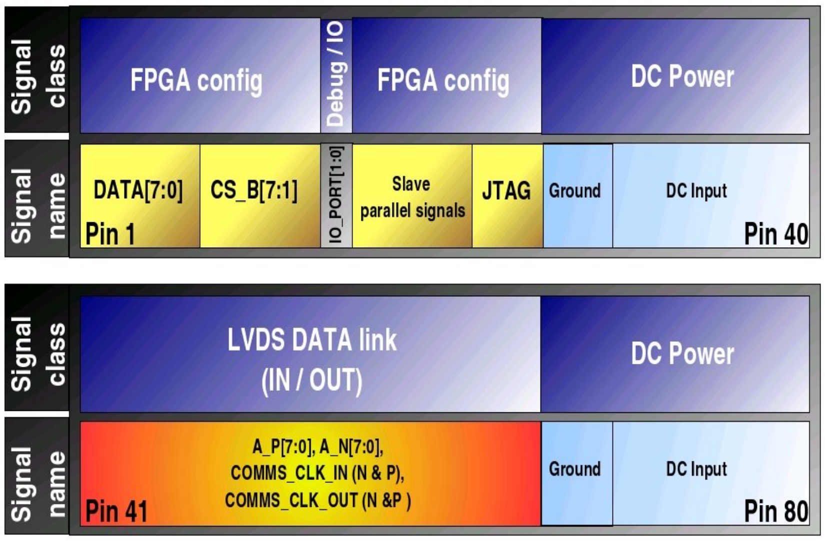


Figure 4.14. Backplane signals

The signal classes are each discussed in the following Sections.

4.5.2.1. FPGA configuration signals

The FPGA's are configured using shared signals across the backplane. The configuration signals are:

- **Slave parallel configuration signals**
  These are the signals required for slave parallel FPGA configuration.

- **JTAG**
  These are the four JTAG configuration signals.

- **DATA[7:0]**
  FPGA configuration data.

- **CS_B[7:1]**
  Configuration Chip Select lines. Individual Chip Select lines are tapped off on each module as required as shown in Figure 4.15.

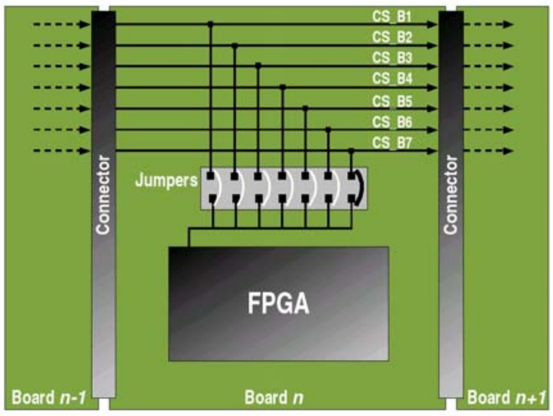


Figure 4.15. Chip select signal tapping

This scheme allows for a maximum of eight FPGA's to be configured in a chain (including the FPGA on the USB module board). It also allows for individual FPGA's to be addressed during the configuration process.

The Xilinx *Slave Serial* configuration signalling is very similar to that used by Altera FPGA's [9], allowing for modules with alternative FPGA architectures to be configured with minimal system modification.

4.5.2.2. Inter-module data signals

The inter-module data signals are shown in Figure 4.16.

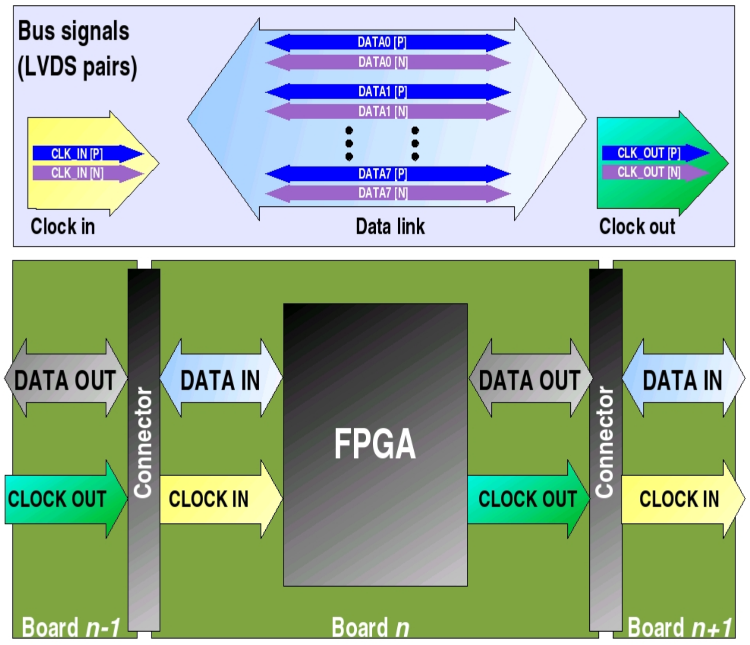


Figure 4.16. Inter-module data signals

There is an 8-bit point-to-point link between each module and both the previous module and the next. The communications are synchronous, and the clock signal traverses the bus along an LVDS link (repeated internally by the FPGA).

4.5.2.3. DC power signals

To facilitate the three power supply levels required by the system, a power bus is implemented across the backplane.

The power bus carries unregulated DC and is tapped off on each board as necessary. The boards then regulate the voltage to their required values as shown in Figure 4.17.

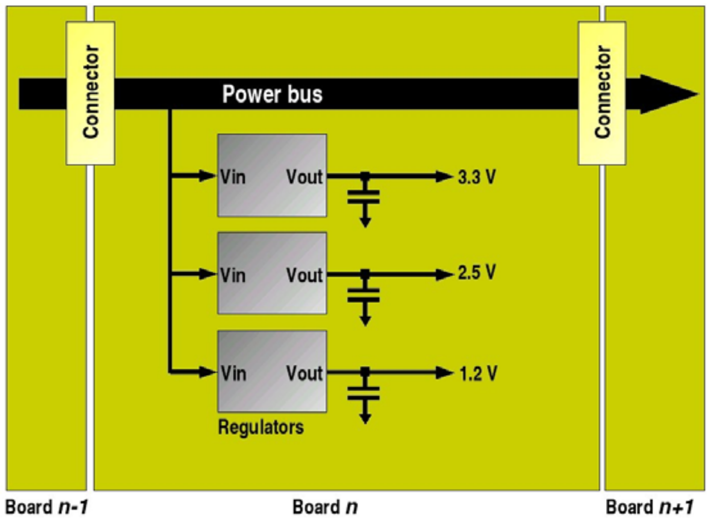


Figure 4.17. Power bus

The FPGA's require 1.2V for their internal core supplies, the 2.5V is required for the FPGA auxiliary logic components and the LVDS drivers, while the 3.3V is used by all the remaining circuitry.

4.5.2.4. Debugging / IO signals

To allow for simple debugging and system testing (particularly testing of inter-module communications), two general purpose signals are provided on the system backplane, *IO_PORT0* and *IO_PORT1*.

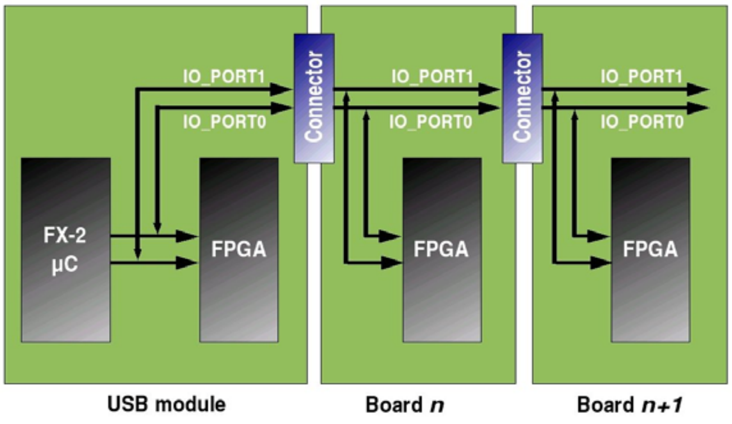


Figure 4.18. IO_PORT[1:0] debug signals

These originate on the USB module and are hard-wired to two of the general I/O pins of the FX2 micro-controller.

The signals are shared across all modules allowing them to be driven by an alternative source, provided the particular FX2 I/O pins are placed in high-impedance mode to avoid signal contention.

4.5.3. Backplane connectors

While most of the backplane signals are shared as they are in a multi-drop topology, the inter-module communications signals are grouped into *IN* and *OUT* links, based on a point-to-point topology.

As the system is designed to be stackable with modules directly above and below each other, the backplane connectors are surface-mounted as shown in Figure 4.19.

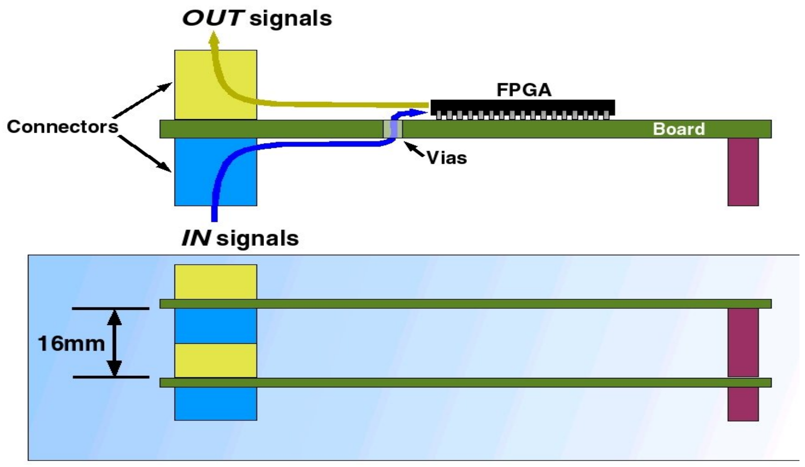


Figure 4.19. Surface-mount connectors.

In using surface-mounted connectors, a height constraint is introduced to each of the system modules of 16 mm between the boards.

4.6. Inter-module communications

With a modular architecture, this system requires a mechanism for inter-module communication. This mechanism needs to support high data rates, and must have a high noise immunity to preserve signal integrity.

4.6.1. LVDS

LVDS (Low Voltage Differential Signalling) is a differential serial signalling protocol ideally suited to high data rates with good common-mode noise rejection properties [10]. It is a form of current mode logic whereby alternating transistors drive current through a differential trace terminated with a 100Ω resistor. An example of a simplified point to point LVDS circuit is shown in Figure 4.20.

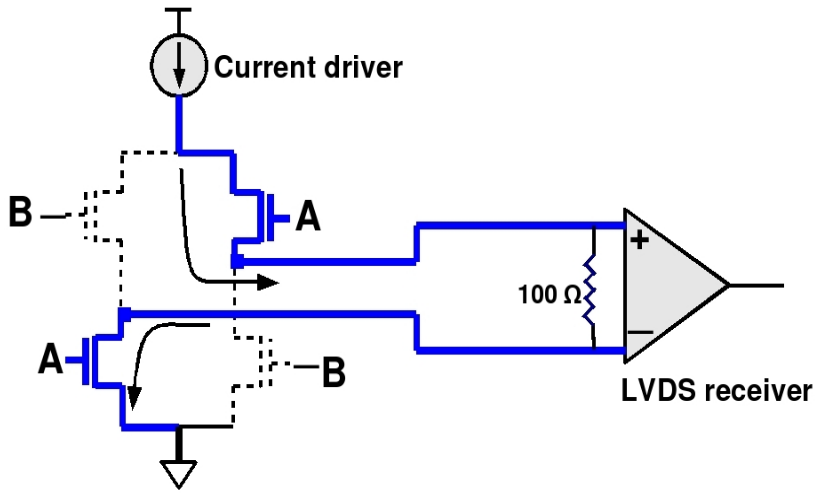


Figure 4.20. LVDS transceiver

The current path alternates through path A followed by path B. As the LVDS receiver is differential, common-mode signals picked up along the trace lengths is rejected, allowing for good noise rejection. As a result of the noise-rejection properties of the LVDS scheme, lower level and faster signalling can be implemented than traditional single-ended logic.

This Chapter described the system design, and covered the system's hardware architecture discussing the ADC and USB modules in some detail. The interconnect scheme was outlined, with an introduction of LVDS signalling provided.

FPGA configuration was also discussed, as well as system debugging ports.

The next Chapter discusses the implementation of the system.

5. System Implementation

This Chapter deals with the implementation phase of the project.

5.1. Printed Circuit Boards

As a predominantly hardware-based master's thesis, a major deliverable of this project is the assembly of two printed circuit boards.

The first stage in the routing process is the placement of the components.

5.1.1. Component placement

To ensure the simplest routing topology, and to minimise net lengths, a lot of care was taken during component placement.

The larger IC's were placed in the centre region of the PCB, while connectors and headers were placed on the edges of the board to enable external signal access. Decoupling capacitors were placed as close as possible to IC's to reduce ground current loops.

Efforts were made to isolate high speed signals from lower speed signals to reduce cross-talk. In addition, analogue signals were isolated from digital signals wherever possible.

5.1.2. Board layers

To ensure the integrity of high speed signals, separate ground and power planes were implemented [10][11]. Both of the boards in this project have four layers as shown in the layer stack below.

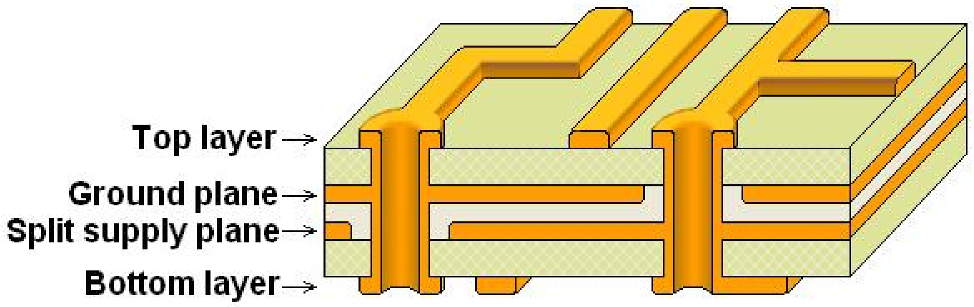


Figure 5.1. PCB layer stack

The top and bottom layers carry signals, with a solid ground plane and a split supply plane sandwiched between them. To minimise production costs, all vias were of through-hole type as opposed to buried vias.

5.1.3. Split supply planes

As there are multiple power supplies required for this system, the supply plane needed to be divided into physically separate areas, each carrying different voltage levels.
An example of a split supply plane is shown in Figure 5.2.

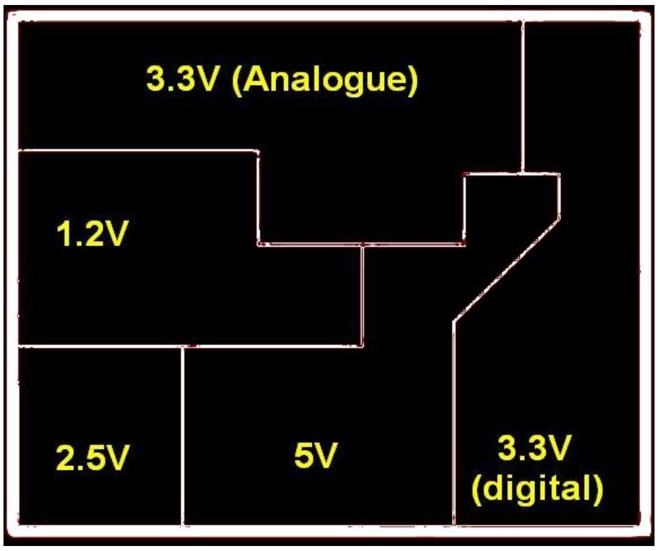


Figure 5.2. Example of a split supply plane

As shown in the Figure, each of the planes is physically separated from the others and these boundaries have to be adequately large to minimise signal cross-talk between the planes (16mil in the case of this design).

This constraint made the design of the supply planes somewhat difficult, especially underneath the FPGA's, as they have multiple supply pins within a small area.

In addition, there are many vias required underneath the FPGA's in order to access the BGA pins, which decreases the effective area of the planes and can introduce 'pinch-points'. This is shown in Figure 5.3.

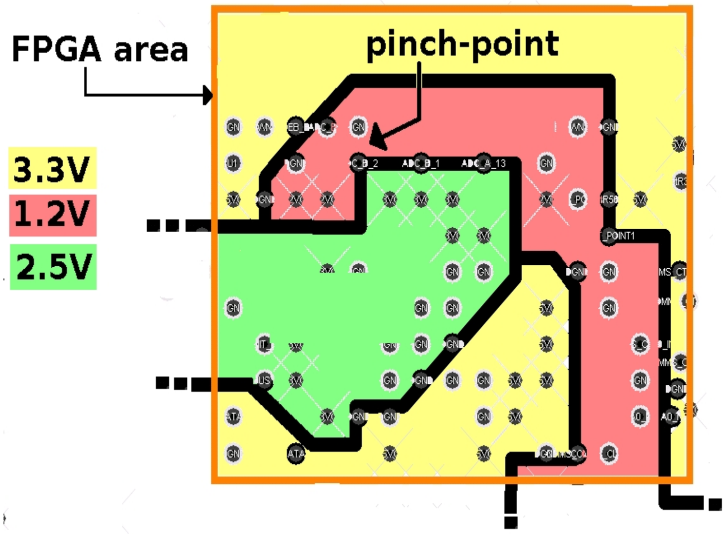


Figure 5.3. Difficulties faced in split supply design

5.1.4. Differential impedances

Both the LVDS and USB signalling standards are differential, and specify a differential impedance value for each signal pair. Each impedance value needs to be maintained along the entire length of the signal pair, and can be calculated based on the physical dimensions of the tracks.

5.1.4.1. Differential impedance calculation

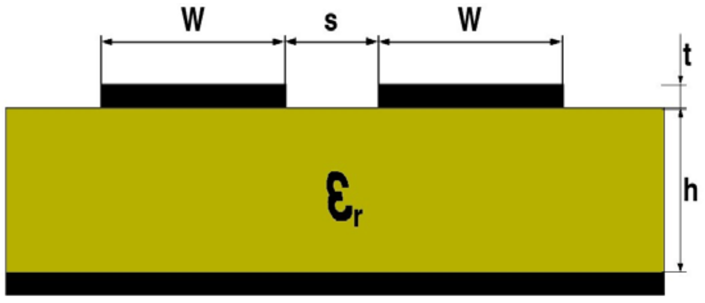


Figure 5.4. Co-planar microstrip

$$Z_0 = \frac{60}{\sqrt{0.475\,\varepsilon_0 + 0.67}} * \ln\left(\frac{4h}{0.67(0.8W + t)}\right)$$

$$Z_{diff} \approx 2 * Z_0\left(1 - 0.48\,e^{-0.96\frac{s}{h}}\right)$$

Table 7. Differential impedance calculation

Symbol	Description
W	Width of track
s	Spacing between tracks
h	Height of tracks above ground plane
t	Thickness of tracks
ε_r	Dielectric of board medium
Z_0	Intermediate impedance value (for clarity in calculation)
Z_{diff}	Differential impedance

The above equations are used to calculate the differential impedance of a coplanar microstrip, which is the structure adopted for all the differential signals in this project.

As recommended by [10], the spacing s between all differential tracks was kept constant during routing, set to the minimum spacing allowed by the board manufacturer which is 0.127mm . Based on standard values for ε_r (approximately 4.5 for FR-4 material), *h* (0.24 mm) and *t* (30 microns), the width *W* was varied in each instance to produce the required differential impedance for the specific tracks.

The main advantage of differential signalling schemes is their common-mode noise rejection. In order to maximise this property, the net lengths were kept to similar lengths and impedance values (within the specified tolerances).

Proximity with the ground pour areas was also balanced where possible, as a ground plane next to only one of the nets would cause an impedance imbalance. Vias introduce impedance mismatches in the signal traces and were avoided where possible.

5.1.4.2. USB signals

In addition to the general routing guidelines for differential signals, the USB standard requires that the signal tracks have a differential impedance of 90Ω within a tolerance of 10% [12].

The tracks must be less than 75mm long (preferred length is 25-30mm), and must be at least 6.5mm from all other non-static nets [11].

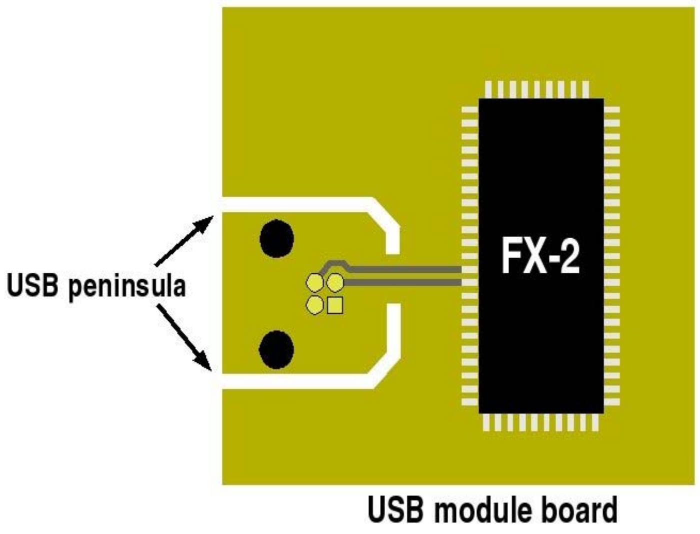


Figure 5.5. USB peninsula

As shown in Figure 5.5, the USB connector is placed on a 'peninsula', which has solid ground and supply planes beneath it, surrounded by a cut-out moat of no tracks [11].

This helps to isolate the USB signals from noise and high-speed signals generated on the rest of the board.

5.1.4.3. LVDS signals

As with the USB differential signals, several considerations had to be made with the routing of the LVDS tracks.

Firstly, care was taken to keep all other nets away from the LVDS signal pairs, while maximising the distance between the LVDS pairs themselves.

Secondly, as the LVDS signals collectively form a bus, efforts were made to ensure that all the net lengths were within 2.54 mm of each other to reduce signal skew. This was achieved by routing the LVDS pair with the furthest pin distance first, and compensating for the additional length in all the other pairs by adding kinks and loop-backs as shown in Figure 5.6.

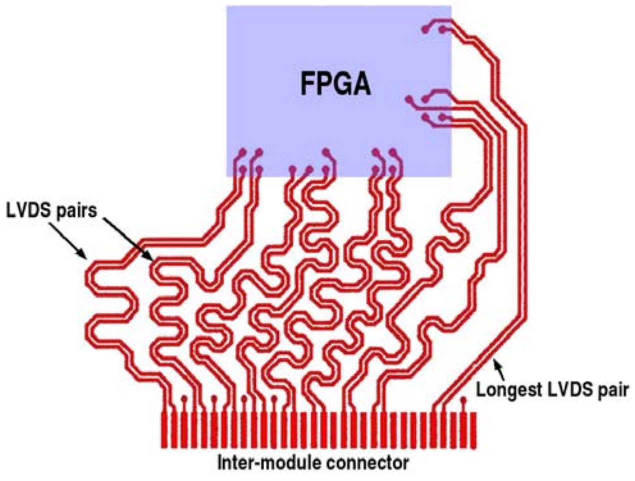


Figure 5.6. LVDS pairs

5.2. Integration with the GnuRadio software

As an open-source project, the *GnuRadio* software and its source code is freely available. The code allows DSP process chains to be constructed which can act on various signal inputs and outputs.

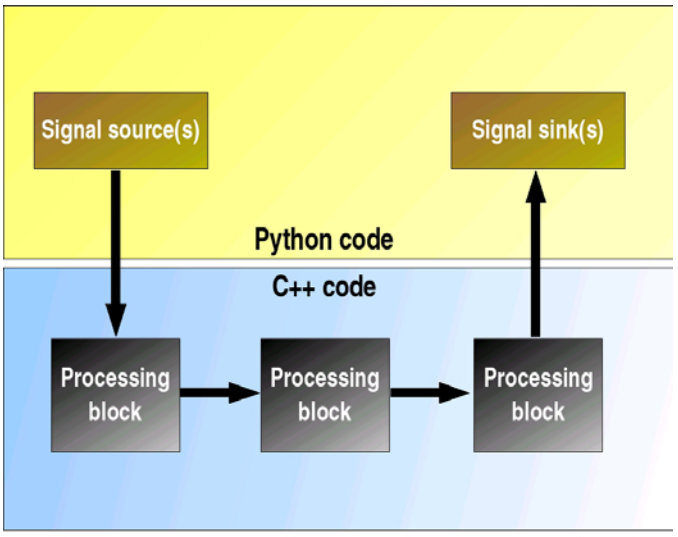


Figure 5.7. GnuRadio abstraction

A variety of sources have been designed by the *GnuRadio* team, including generic sound cards (microphone input) and audio-format files. Signal sinks include audio output, files and visual oscilloscopes which are updated in real-time.

The performance-critical processing blocks are written in *C++*, while the description of the processing graph structure for a given application is written in the *Python* scripting language.[3] After installing the requisite *GnuRadio* modules, *test_basic_rx*, a customized testing module was run on an *Ubuntu Linux* system.

5.2.1. Testing the SDR platform

To test the project, the modules need to be connected and powered up with a DC supply of +5V.

A Windows platform is required to configure the FPGA's via JTAG, using the *Xilinx ISE iMPACT* configuration application. Once the FPGA's have been configured, The FX2 needs to be programmed (using the open-source *fxload* application).

5.2.1.1. Programming the FPGA's

To program the FPGA's, the following steps must be carried out :-

- Ensure that the modules are connected, and that the boards are secure (alignment bolts have been tightened).
- Power up the board.
- Connect the JTAG programming cable to the JTAG port of the USB module.
- Start up the Xilinx *iMPACT* application, and select 'Configure devices'.
- Select 'Autodetect cable'. If there is an error at this point check that the cable is connected correctly, and that the board has been powered to the correct supply voltage (+5V).
- Once a generic Spartan-3 device has been detected, a configuration file needs to be associated with the project. Open the USB module's *.bit* file.
- Right-click the Spartan-3 device and select 'Program'.
- The application will indicate successful FPGA configuration. Note that if the FPGA fails its configuration sequence, simply exit and try again (erroneous errors are sometimes encountered).
- Repeat the above steps with the JTAG cable plugged into the ADC module's JTAG port (note that you will have to relaunch *iMPACT*). This time select the ADC module's *.bit* file.

When both FPGA's are configured, the current firmware asserts the board LED's in the following sequence:

- Both of the ADC module's LED's are de-asserted on start up. *LED[0]* flashes on briefly at the start of a data transfer sequence. If *LED[1]* is asserted at any stage, there is an error in the sample datapath.
- Both of the USB module's LED's are repeatedly asserted then deasserted in the MHz range when both of the boards have been connected and configured (the LED's are not fully on in this state). If either of the LED's exit this state, then there is an error in the datapath.

5.2.1.2. Programming the FX2

Once the FPGA's have been configured, the FX2 needs to be programmed so that the host PC can access the system.

This is done from a shell console on a Linux-based system, in the following steps :-

- Connect the powered, connected system to a high-speed USB 2.0 port using a USB cable.

- Check that the FX2 device has been recognized correctly by the operating system by typing *lsusb*. This command lists the devices currently connected to the USB hierarchy. You should see something like the following:

  ```
  kalen@3man:~$ lsusb
  Bus 005 Device 003: ID 04b4:8613 Cypress Semiconductor Corp. CY7C68013 EZ-USB
  FX2 USB 2.0 Development Kit
  Bus 005 Device 001: ID 0000:0000
  Bus 003 Device 001: ID 0000:0000
  Bus 002 Device 001: ID 0000:0000
  Bus 001 Device 001: ID 0000:0000
  Bus 004 Device 001: ID 0000:0000
  ```

  Note that device 003 on the USB bus 005 has been identified as the FX2 device.

- Now the FX2 needs to be configured with the appropriate firmware. This is done using the fxload application with the following command:

  ```
  fxload -t fx2 -D /dev/bus/usb/<bus>/<device> -I <firmware_file.ihx>
  ```

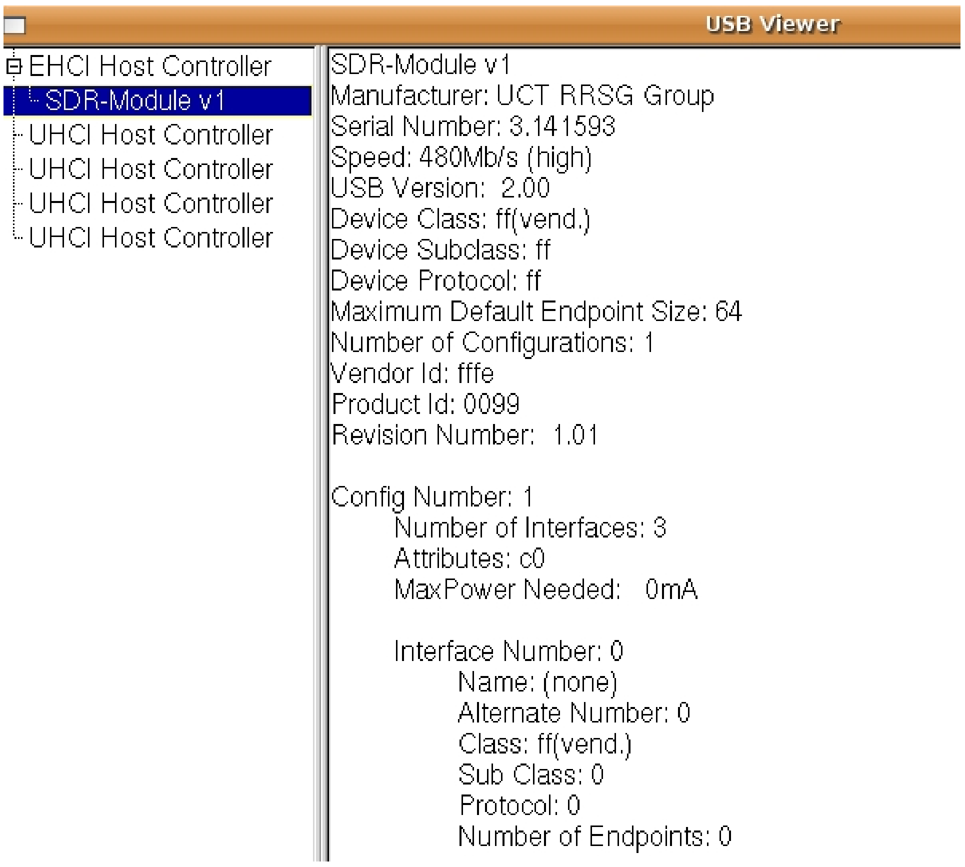
  Where <*dev*> is the bus number and <*device*> is the USB device number (005 and 003 respectively in this instance). <*firmware_file.ihx*> is the full path and filename of the FX2 firmware (*usrp_main.ihx*).

  Note that the firmware forces a USB renumeration with a new product/vendor ID combination on start up, so when you type lsusb after downloading the firmware, a different USB topology will be presented. It will look something like this:

  ```
  kalen@3man:~$ lsusb
  Bus 005 Device 004: ID fffe:0099
  Bus 005 Device 001: ID 0000:0000
  Bus 003 Device 001: ID 0000:0000
  Bus 002 Device 001: ID 0000:0000
  Bus 001 Device 001: ID 0000:0000
  Bus 004 Device 001: ID 0000:0000
  ```

  In the above shell output, the original FX2 device has now renumerated (device number incremented to 004), and that it has the vendor ID 0xFFFE (Free Software Foundation) and product ID 0x0099 (custom selected ID for this project).

- You can establish that the system has been configured correctly by running *usbview* which will present the following screen on successful system configuration:



```
SDR-Module v1
Manufacturer: UCT RRSG Group
Serial Number: 3.141593
Speed: 480Mb/s (high)
USB Version:  2.00
Device Class: ff(vend.)
Device Subclass: ff
Device Protocol: ff
Maximum Default Endpoint Size: 64
Number of Configurations: 1
Vendor Id: fffe
Product Id: 0099
Revision Number:  1.01

Config Number: 1
        Number of Interfaces: 3
        Attributes: c0
        MaxPower Needed:   0mA

        Interface Number: 0
                Name: (none)
                Alternate Number: 0
                Class: ff(vend.)
                Sub Class: 0
                Protocol: 0
                Number of Endpoints: 0
```

At this point, the system has been configured correctly, and testing can begin with the *test_basic_rx* application which is discussed in the next Chapter.

5.3. System limitations

Various limitations were imposed on the system during its implementation, departing from the original design's specifications. These are discussed next.

5.3.1. ADC sampling resolution

The ADC sampling resolution has been reduced to 8 bits, as opposed to the full resolution of 12 bits.

The system implements an 8-bit LVDS bus between the modules, and the original design intended for the ADC samples to be padded to 16 bits words, split into two byte portions, then recombined in the correct sequence by the USB module.

Unfortunately, a data flow control track was damaged on the USB module and could not be repaired due to time limitations. This limited the flow control to a single line, which is required for a system-wide reset.

The current implementation therefore has no way of indicating to upstream devices when transmitted data is valid or not. This leads to the problem of synchronising the reconstruction of the sample bytes into their original 16-bit word values, where frame slippage would cause extreme corruption of the resultant sample values. The only way to guarantee the validity of the data therefore was to reduce it to atomic units of 8-bits each.

This obviously reduces the dynamic range of the system, but fortunately the problem is limited to a single board and future board assemblies can make use of robust data flow control. This will also limit sample discontinuities (a recommended solution is to hold the capture sub-system in a reset state until the host initiates a data request, whereby signal capture begins. The host will then read its required data allotment with no discontinuities as the internal buffering in each FPGA is 16K, which is far larger than the average USB packet size).

5.3.2. Effective sampling rate

A consequence of the above flow control limitation is that the sample source cannot indicate to the upstream board whether or not the data presented on the inter-module bus is valid or not. To ensure that continuous blocks of data were captured during the performance testing stage, decimation was used to throttle the sample rate (the FX2 stalls while waiting for data).

The system was designed with the view to implement some form of DDC in firmware to allow narrow-band analysis of a wide-band sample stream, reducing the data rate in any case.

5.3.3. FPGA configuration

The original design planned for FPGA configuration via the USB connection. The schematic was designed in such a way that the configuration and sample data buses are shared. In addition, the flow control and configuration control lines share the same dual-purpose FPGA pins.

This implementation can be made to work provided the FX2's GPIF can be programmed to handle both FPGA configuration bus cycles and regular data transfers. In other words, on intercepting an FPGA configuration request from the host, the GPIF would be setup to handle Xilinx's configuration protocol. On completion, the FX2 would then setup the GPIF to handle bulk USB transfers with the FPGA (its configuration data lines are made available as general-purpose I/O internally).

Unfortunately, the FX2 turned out to be complex device to configure correctly. After much discussion with the *GnuRadio* forum, and reverse-engineering of their firmware (they keep the sample and FPGA configuration lines separate from each other, sending configuration data bitwise over a single line), it was decided that the best use of limited time would be to carry out various hardware modifications to match the USRP's GPIF control interface.

As this prevented the configuration of the FPGA’s via USB, the JTAG signals across the backplane were also isolated to allow for independent FPGA configuration. While this is a slightly more cumbersome solution for system configuration, the overall result is the same.

Future revisions of the system may look more closely at configuring the GPIF to handle all four waveform modes of data transfer, but this is a non-trivial problem.

This Chapter covered the implementation of the system design, first discussing the component placement and layout of the circuit boards, followed by the integration of the system with the *GnuRadio* software.

Finally, Section 5.3 discussed various system limitations.

The next Chapter looks at system testing.

6. System Tests & Results

This Chapter discusses the various tests carried out on the system, and the results obtained. It has been divided into two halves, covering both functional and performance tests.

6.1. Functional tests

Functional tests determine the system's ability to carry out its specified functional requirements. That is, they determine whether or not the system does what it was designed to do. These tests determine whether the system meets its most basic design requirements.

6.1.1. General system

Initially, functional tests were carried out on the common aspects of both boards, namely the power supplies and the FPGA’s.

6.1.1.1. Power supplies

Tests were carried out on the power supplies on both of the boards to ensure that the correct voltage levels were distributed across the systems. The ripple voltage on the supplies was also measured, and found to be within +/-10% of the supply level.

6.1.1.2. JTAG FPGA configuration

The FPGA’s need to support the JTAG configuration mode, and this was tested with a pre-compiled bitstream on both of the FPGA’s. The bitstream implemented a simple circuit which read in an input clock frequency and output various fractions of this frequency. In addition, LEDs were toggled based on switch inputs.

These basic tests were carried out successfully, confirming that the JTAG ports on both of the modules were working correctly.

6.1.2. USB module functionality

These tests were carried out on the USB module to ensure that it was operating as specified. They basically tested the module's sub-systems.

6.1.2.1. USB connection

The first test that was carried out was of the USB connection itself. Without loading any FX2 firmware, the USB host must recognize the FX2 device as a valid USB 2.0 device. In effect, this tests that the FX2 has been correctly implemented, including its power supply and signal connections.

Initially, this test was done using *CyConsole* and other free diagnostic tools for Windows developed by Cypress Semiconductor. The device was recognized as a USB 2.0 device, and basic configuration information was retrieved from the device successfully.

After this, more substantial tests on the FX2 microcontroller were carried out. As these used the *GnuRadio* USRP modules, they were carried out on a Linux platform.

On power up, the Linux shell command *lsusb* listed the FX2 device as being attached to the system. After downloading of the FX2 firmware (and internal bus renumeration with different product and vendor ID's), the system was identified with the following characteristics:

- Vendor ID – 0xFFFE
- Product ID – 0x0099

The shell displays were the same as those discussed in Section 5.2.1.2. Once it had been verified that the system was being recognized as a valid USB device, the USB connection with the system was confirmed to be working.

6.1.2.2. USB data stream

To test that the system was delivering valid data timeously and in response to host requests over the IN endpoint 2, the *test_usrp_standard_rx* application was modified to interface with the system.

This application was written by Larry Doolittle for the Avnet Virtex-4 evaluation board (LBNL UXO), and released under the same GPL license as *GnuRadio* itself.

The application accepts various command line parameters and requests a block of data of a pre-specified size from the attached device, writing the data to file. On completion of the test, it calculates and displays the USB data rate sustained during the transfer.

As part of the test, the USB module's FPGA was configured to output a cycling 16-bit ramp signal when requested for data. The recorded file generated by the test application was then examined and checked against the expected results. Examination of the captured data indicated that the USB module's data streaming capabilities were working correctly.

6.1.3. ADC module functionality

These tests were carried out on the ADC Module to ensure that it was transferring data to the USB module and then on to the host correctly.

Once again, the 16-bit cycling test pattern was used to verify the data transfer. This time, the ramp was generated in the ADC module firmware and sent across the LVDS bus. The USB firmware captured the data, and buffered it before issuing to the host.

The captured data was again examined to ensure that the test pattern was transferred all the way through the system to the host.

After this test was carried out successfully, the ADC was tested along with the system in general.

6.2. System performance tests

The final testing was carried out on the system as a whole, with the run-time firmware loaded on both FPGA's. Analogue signals were captured, with the samples transmitted up to the host PC for offline analysis to determine the system's performance.

6.2.1. Test setup

Figure 6.1 shows the test setup used to carry out these tests. It consists of a PC running *Ubuntu 6.06 Linux*, the project hardware and a signal generator.

A JTAG cable provides a programming interface for the module FPGA's. A standard USB cable connects the USB module to the PC, while a shielded BNC to SMA cable feeds the signal generator's output to the ADC module.

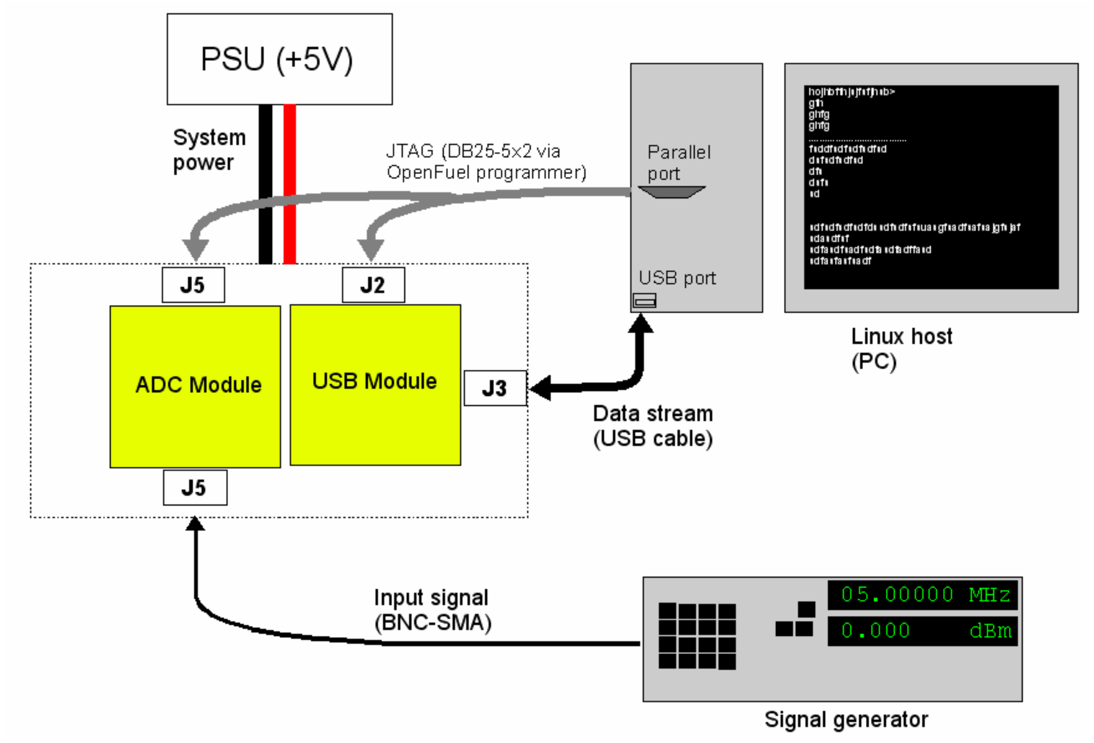


Figure 6.1. Test setup

A series of single tone references were fed into the analogue input of the ADC module, which sampled them at 64MHz, streaming the data continuously to the USB module.

As discussed in Section 5.3, the full 12 bits of ADC resolution could not be realised, and the samples captured in this test were 8-bit bytes sign-extended to 16-bit words.

The results were examined using *MatLab 7.0*.

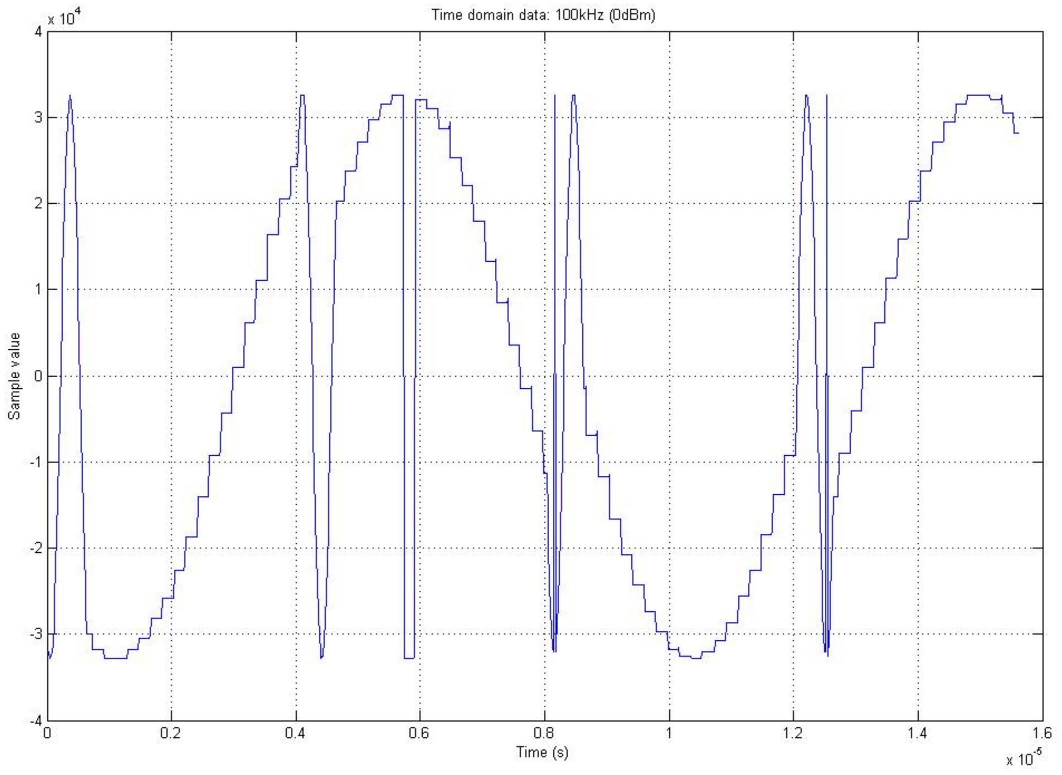


Figure 6.2. Time domain capture of a 100kHz signal at 0dBm

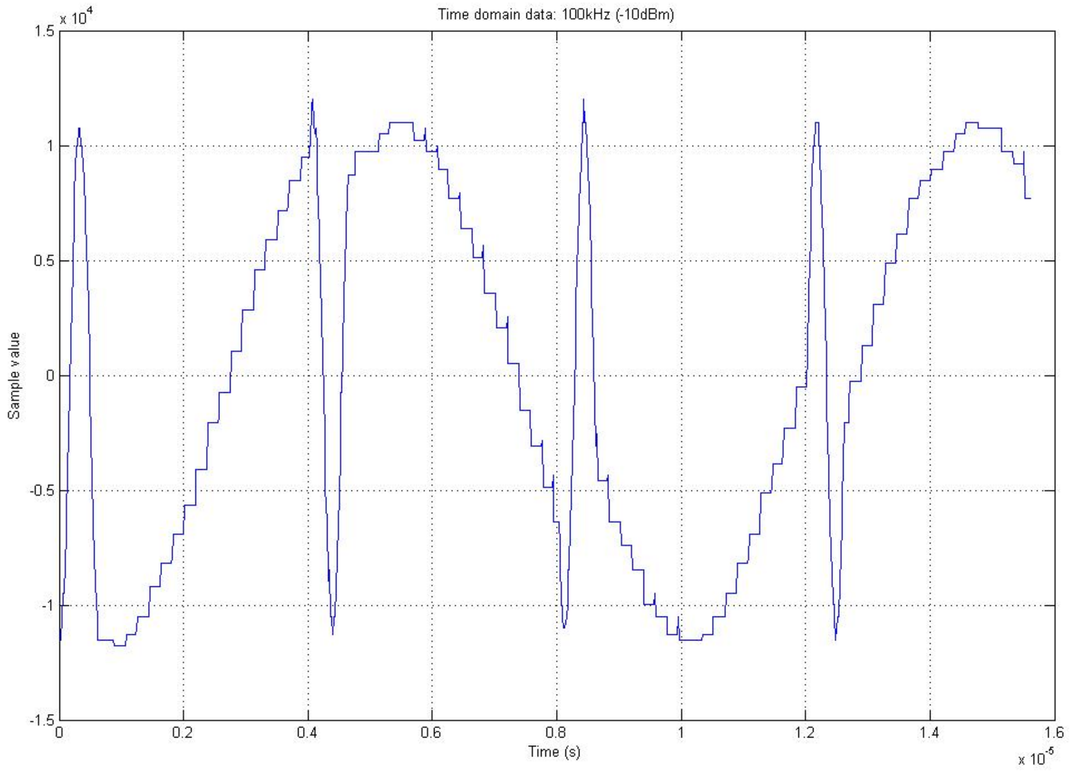


Figure 6.3. Time domain capture of a 100kHz signal at -10dBm

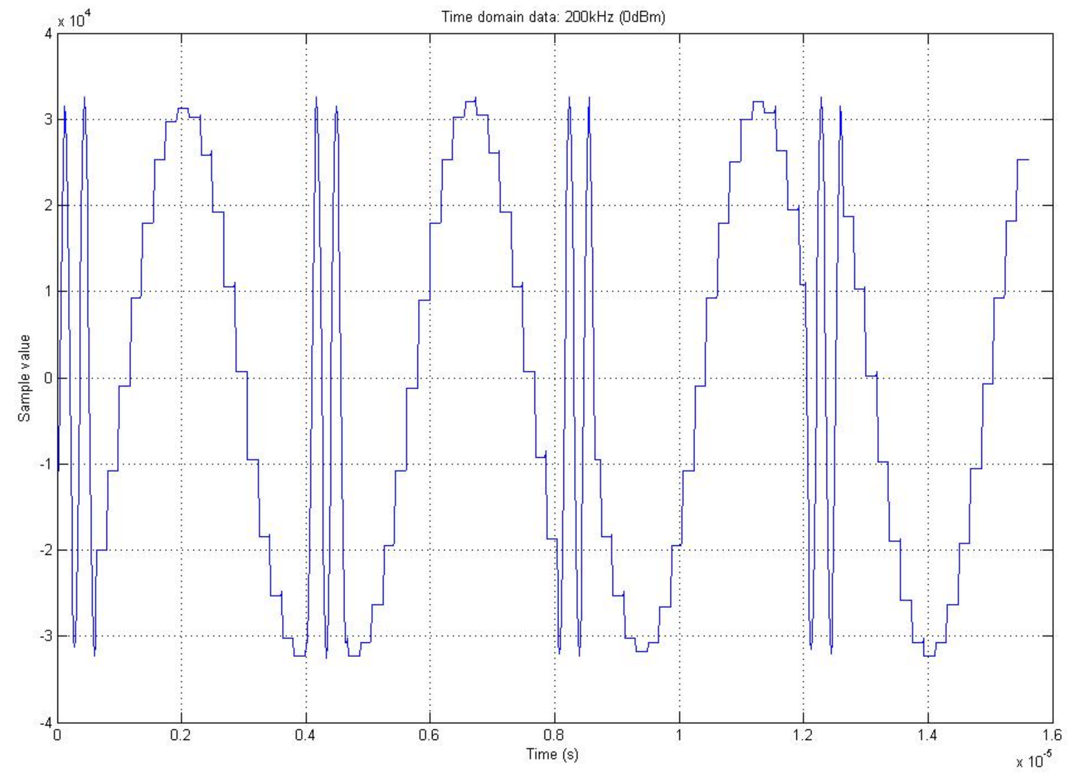


Figure 6.4. Time domain capture of a 200kHz signal at 0dBm

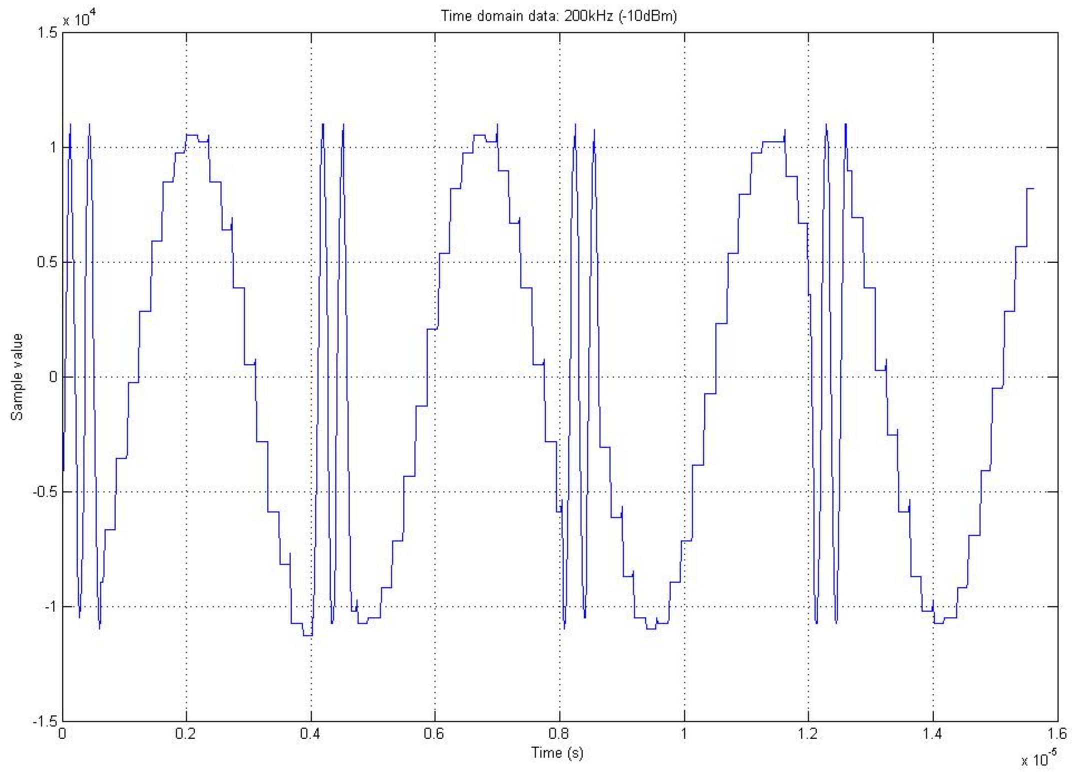


Figure 6.5. Time domain capture of a 200kHz signal at -10dBm

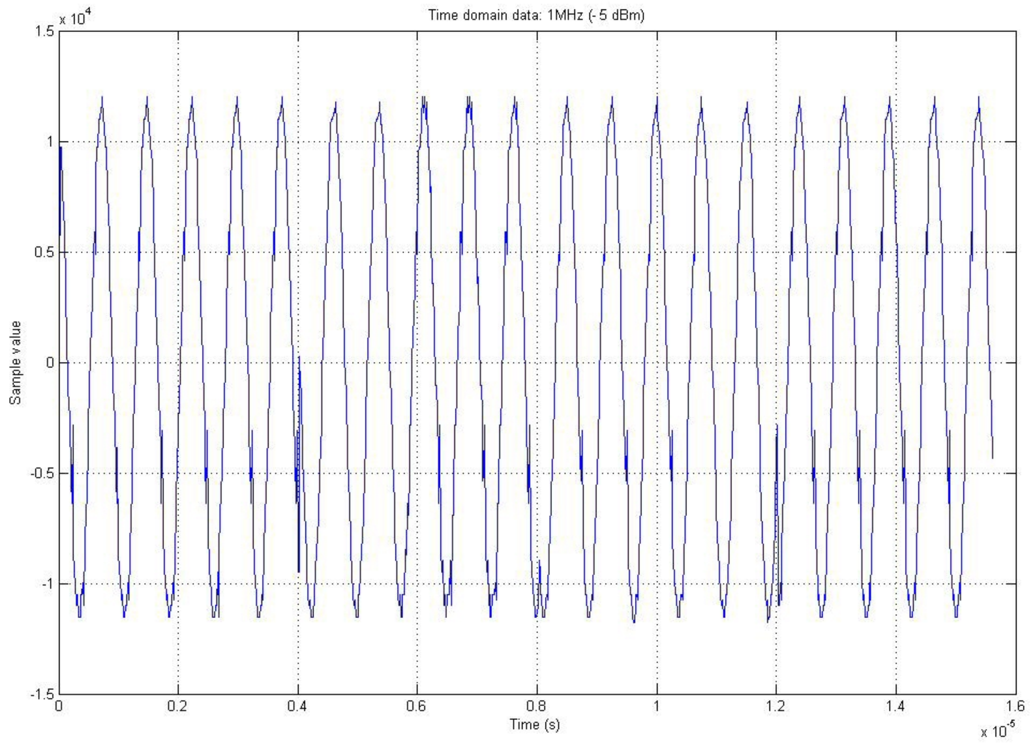


Figure 6.6. Time domain capture of a 1MHz signal at -5dBm

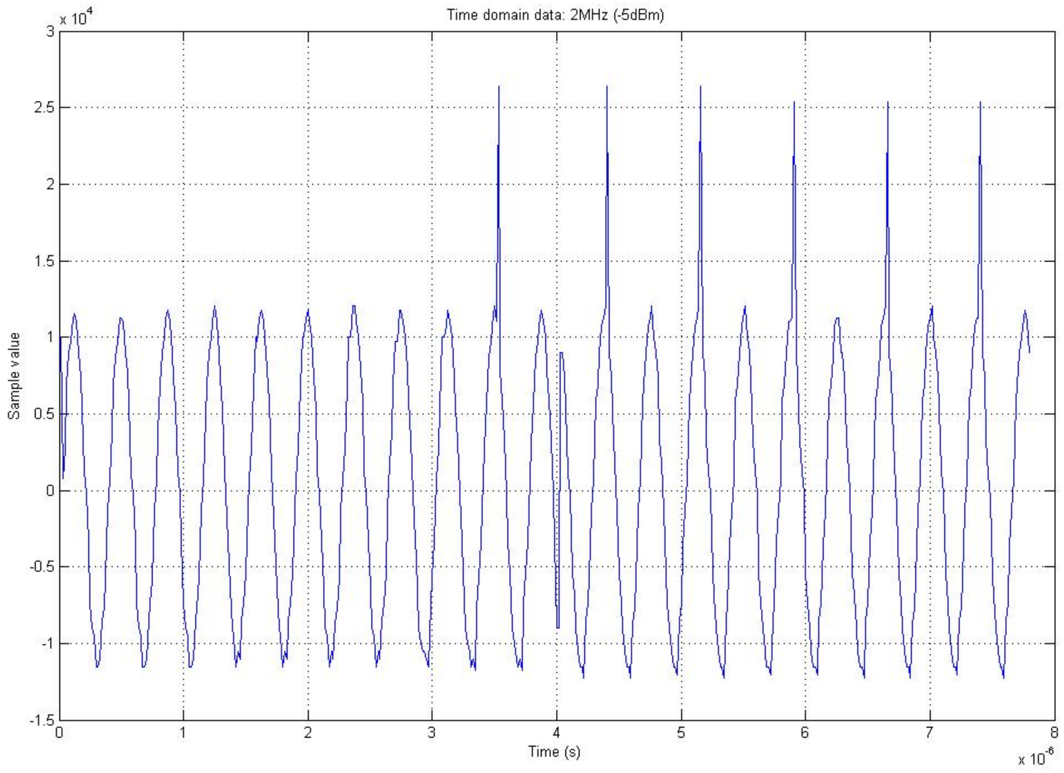


Figure 6.7. Time domain capture of a 2MHz signal at -5dBm

The above results indicated that the system was capturing an analogue signal and transmitting it through to the USB host, albeit with erroneous values introduced into the sample stream, which are discussed further at the end of the Chapter.

The next section looks at the the datasets in the frequency domain.

6.2.2. Determination of system SFDR

Once the system had passed its functional testing, the analogue performance needed to be determined. This was carried by examining the SFDR on the datasets captured in the previous Section.

The following Figures present 4096-point FFT's done on the previous datasets.

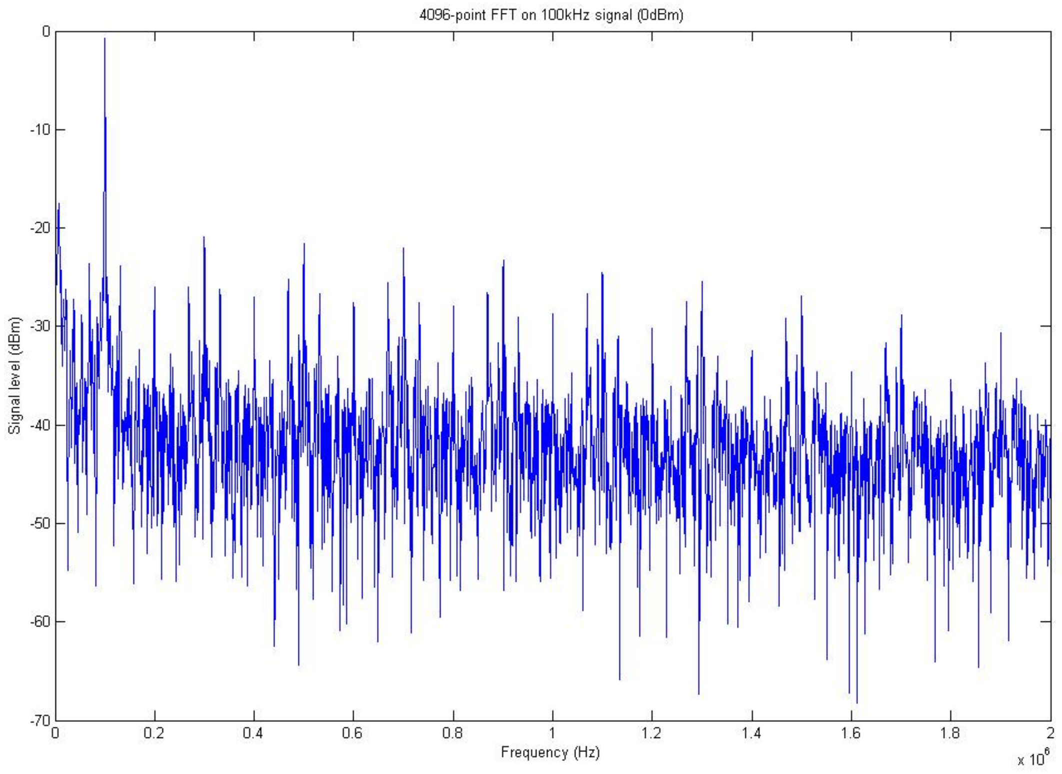


Figure 6.8. 4096-point FFT of 100kHz signal (0dBm)

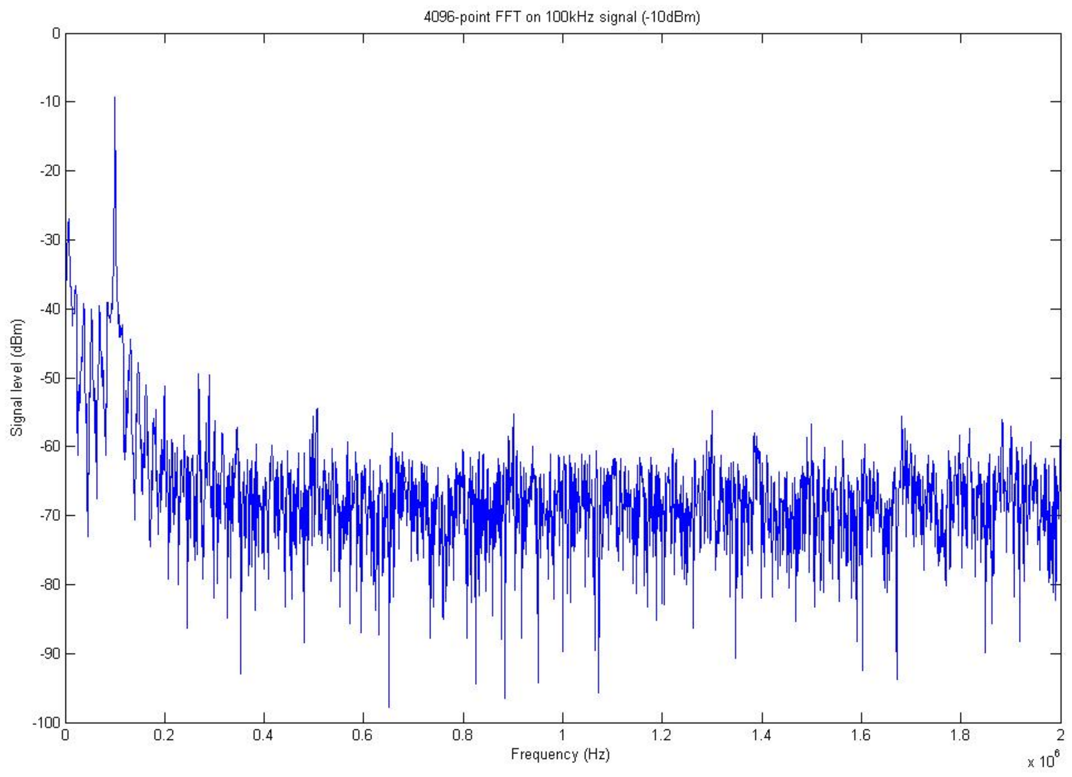


Figure 6.9. 4096-point FFT of 100kHz signal (-10dBm)

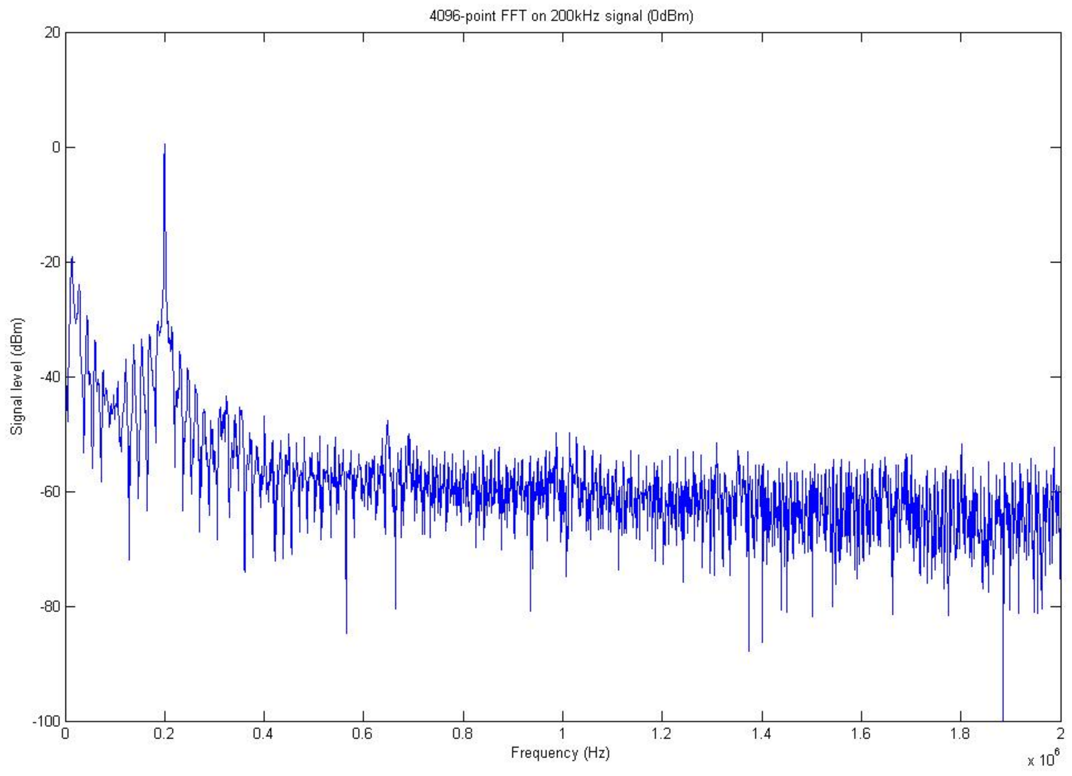


Figure 6.10. 4096-point FFT of 200kHz signal (0dBm)

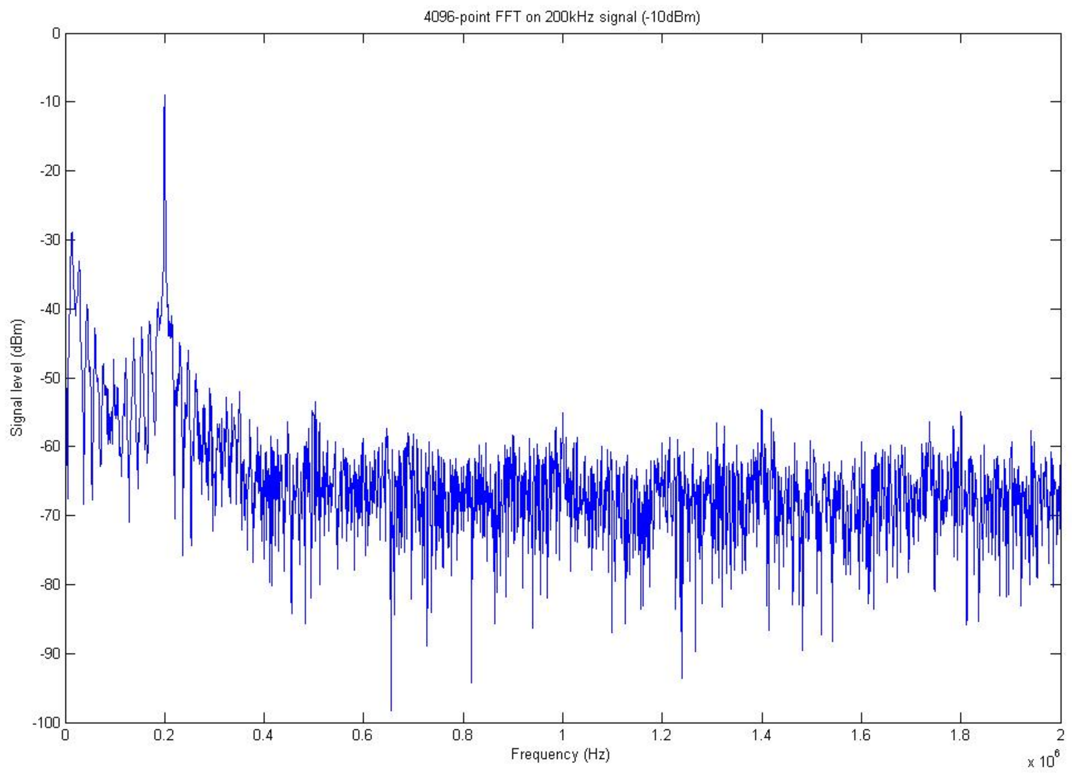


Figure 6.11. 4096-point FFT of 200kHz signal (-10dBm)

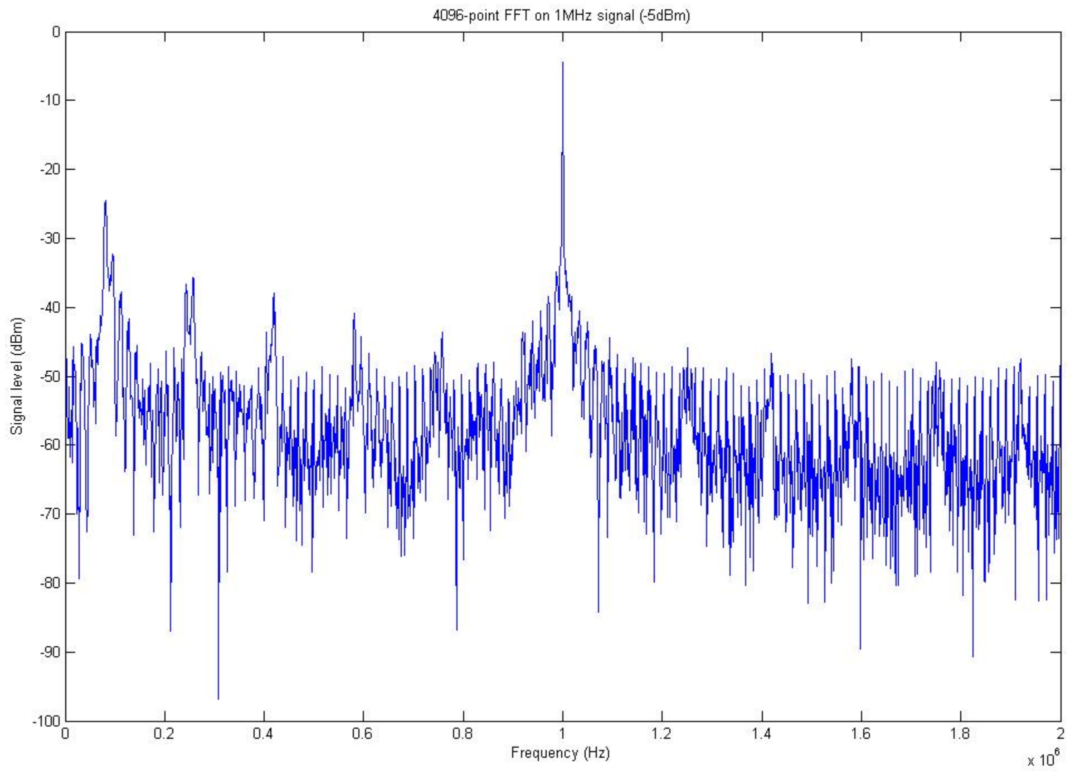


Figure 6.12. 4096-point FFT of 1MHz signal (-5dBm)

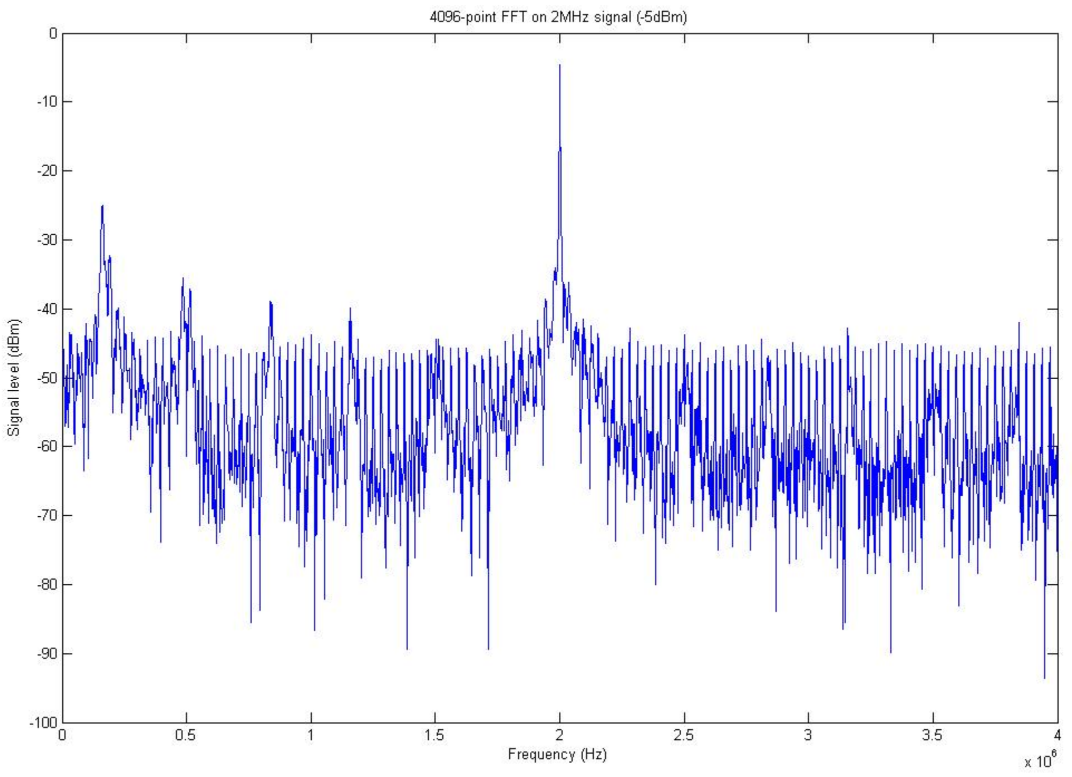


Figure 6.13. 4096-point FFT of 2MHz signal (-5dBm)

A discussion of the test results follow in the next Section.

6.3. Analysis of test results

From the FFT's shown in Figures 6.8 to 6.13, it can be seen that the system SFDR is around 20dBm. This performance level would be unacceptable in most signal analysis applications.

6.3.1. ADC full scale level

As discussed in Section 5.3.1, the ADC resolution had to be reduced to 8 bits. The smallest signal that can be represented using 8 bits (256 levels) is:

$$20\log\left(\frac{1}{256}\right),$$

which yields a signal level of -48dBFS, or in other words an 8-bit ADC has a theoritical dynamic range of 48 dB.

Of course, this range is relative to the full scale input of the ADC. This system was designed with a full-scale input swing of 2Vp-p, which for a 50Ω load has a power level of 13dBm, or in other words it has a full scale input level of 13dBFS.

A full scale requirement of 13dBFS places a considerable constraint on the signal source, which needs to output a clean signal at levels around 13dBm to make optimal use of the available range.

This point is mentioned to bring to mind that with test levels of 0dBm and lower, up to 13dBm of range has been sacrificed. To ease the full scale input requirement on the ADC, two strategies can be adopted in future designs.

6.3.1.1. Introducing gain to the ADC front-end using a transformer

One method for reducing the signal level requirement is to introduce gain in the ADC front-end through using transformers with step-up ratios (the current transformer has a 1:1 ratio). The step up effect of the transformer amplifies the input voltage to the ADC, allowing lower input signals.

It must be remembered that impedances on either side of the transformer are translated by a factor related to the number of turns squared. This means that a resistor (possibly a shunt) must be placed on the primary side of the transformer so that the system still presents a 50Ω load to the outside world.

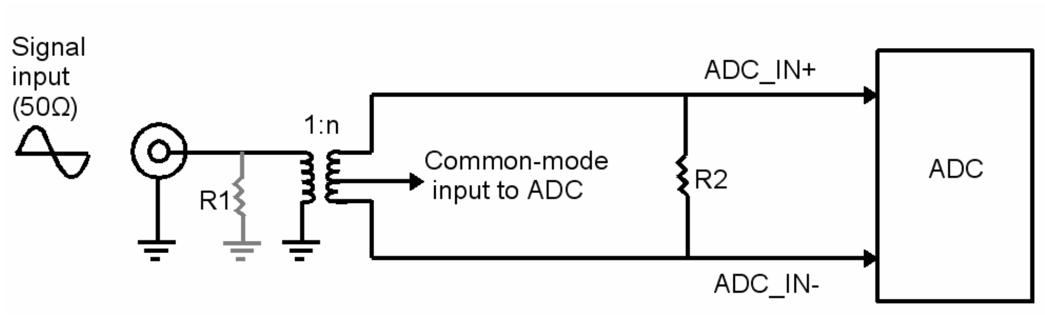


Figure 6.14. ADC front-end using transformer

A simplified diagram of the ADC front-end using transformers is shown in Figure 6.14. The input impedance of a differential ADC is quoted in its datasheet, and is typically around 1k Ω. *R2* can be adjusted empirically until it matches the differential impedance of the ADC. This impedance is reflected back to the transformer input, so the shunt resistor *R1* is required to present the signal input with a 50 Ω impedance. This method will only be valid in the pass-band of the transformer, outside of which non-linearitie will be introduced.

6.3.1.2. Using a differential amplifier to drive an ADC

An alternative method for driving a differential ADC is by using a differential amplifier. This allows direct coupling of the signal to the ADC (offering improved linearity at low frequencies approaching DC). Differential amplifiers are available with fixed gain internal to the IC which is desirable as it reduces the overall component count. The linearity of the amplifier must be studied in the frequency band of interest, because as an active device, the amplifier will introduce noise and non-linearities to the signal.

6.3.2. Discontinuities in the sample stream

Discontinuities in the captured sample streams are evident (most notably in Figures 6.2 to 6.5). These are attributed to the lack of adequate data flow control from the ADC module through to the host PC (discussed in Section 5.3.2). These periodic discontinuitied are visible as sub-harmonic spurs in Figures 6.8 through 6.13.

A more robust scheme of controlling the data flow control is required. The first step would be to ensure that an adequate number of control signals are available between the two modules. Stalling is available between the *FX2* microcontroller and the host PC, so provided that the incoming data rate from the ADC is adjusted so as not to exceed the average USB data rate, no discontinuities will be introduced as a result of internal buffer overruns.

On a similar point, it is recommended that any future work based on this design increases the data bus to 16 bits, so as to maximize the full ADC data rate (in single channel mode). A Digital Down-converter (DDC) implemented in firmware could decimate the sample stream sufficiently to meet the available USB bandwidth, while allowing analysis of selected *IF's* within the sampling band. In order to successfully inmplement a DDC, a larger FPGA will be required.

Potentially, the 14-bit pin-compatible ADC (AD9248-65) could be placed in future assemblies to gain an extra two bits of resolution, though these too would be sacrificed with the current 8-bit data bus.

6.3.3. Signal distortion

Also evident in both the time and frequency domains, there is general signal distortion which can be attributed most likely to a number of sources:

- Possible marginal timing discrepancies between the ADC and the first FPGA. The *User Constraints File* (UCF) for the ADC module's firmware did not include any time constraint specifications to meet logic setup and hold times. This was an oversight and is recommended for any future work on the design.

- Jitter introduced by the FPGA's DCM when used as a clock source for the ADC may be a source of noise. The Spartan-3 introduces DCM jitter of the order of +/-100ps [7] which equates to roughly +/-6% of the sample period when the ADC is sampling at 65MSPS [8]. It is recommended that ADC be driven through a *BUFG* (dedeicated FPGA clock buffer) as opposed to a DCM output.

- A further area for improvement for system performance would be to bettre isolate the digital and analogue power planes in the system. At present they are isolated through balun transformers. A simple low-pass filtering circuit consisting of an inductor and several shunt capacitors could be introduced to limit noise transfer between the planes. In addition, the ADC shares its supply with the analogue section of the *FX2*. These should definitely be isolated to minimise crosstalk between the two *IC's.*

The next Chapter presents conclusions drawn at the end of the project, and gives a list of recommendations for future work.

7. Conclusions and Recommendations

Various conclusions were drawn up after the project development, namely :-

- A hardware platform was developed that consists of two modules, a sample capture board and a USB bridge.

  The development process involved:
  - Schematic design, board layout, board assembly and debugging.
  - FPGA firmware development.
  - Microcontroller firmware understanding and custom modifications.
  - Host library understanding and custom modification.
  - Application software understanding and custom modification.

- The system was found to function correctly in that reference signals were captured and their corresponding sample streams transported to a host PC via USB 2.0. In terms of the product specifications in Table 1, the system can not be programmed via USB and must be configured via the two JTAG ports.

- The analogue performance of the system was found to be inadequate for most typical signal processing applications, with an average SFDR of around 20dBm (the target SFDR was 72dBFS). The possible causes of this disparity in performance were discussed, along with steps to improve the system performance in future designs.

- The *GnuRadio* software was successfully integrated with the custom hardware, providing access to a limited set of the *GnuRadio* functionality.

  It must be noted that while development within the open-source framework offers more support and a visible code base, a steep learning curve was encountered getting up to speed with the *GnuRadio* framework.

  Ultimately though, the use of open-source software provided a much more robust and stable system than a similar one based on a closed-source paradigm would have.

- Finally, a simpler, more inexpensive system could be designed with the same functionality and specifications by following the recommendations listed below.

Based upon these conclusions, several recommendations for future work can be made, namely:-

- **System design modifications**
  The following suggestions can be implemented in future design revisions to improve the overall system performance:

  - The module FPGA's should be selected so as to contain more logic resources, allowing more sophisticated data flow control, firmware DDC's and improved buffering. The current design is pin compatible with the *Xilinx* XC3S1000 FPGA (FT256 package), which has a fourfold increase in logic resources.

  - The interconnection scheme should be revised to allow for 16-bit transfers from the ADC to the USB module. Ideally this would be a shared bus with adequate bus control to avoid contention. This could possibly make use of the PC104 stack-through connectors by re-assigning the pins to custom functions (not PCI).

    - The FPGA configuration scheme should mirror that of the URSP whereby the *FX2* microcontroller drives out the required configuration signals. The number of backplane pins required for a configuration daisy chain is minimal. The JTAG ports should remain on any future design revisions to allow for debug access.

    - More test access points should be added to the design as this would make the debugging process much simpler.

    - An analogue front end that introduces some fixed gain to the signal input should be implemented. This can either make use of transformers or a differential amplifier as discussed in Sections 6.3.1.1 and 6.3.1.2.

    - Basic passive filter circuitry should be introduced to isolate the analogue power supplies of both the *FX2* and ADC.

- **Software modifications**
    - A more extensive test application should be written to handle more extensive testing, ideally allowing real-time display of the sample stream.

Over and above the modifications just described, the following areas are recommended for future work :-

- **DAC module**
  A DAC module could be developed for signal transmission. The module architecture would be based on the ADC module. In order for the DAC module to co-exist with the ADC module in the system, and addressing scheme would need to be developed which would be controlled by the USB module as the bus master.

- **Timing module**
  A timing module could be developed to supply system-wide time information, possibly using a GPS-based signal.

Appendix A - Schematic Diagrams

This appendix presents the schematic diagrams of the project's two modules. They are listed in the following order :-

USB Module

- FX2
- FPGA
- FPGA configuration
- Power supplies
- Interconnect

ADC Module

- ADC
- FPGA
- FPGA configuration
- Power supplies
- Interconnect

Appendix B - PCB layouts

This appendix presents the layouts for both the modules in the project.

They are arranged in the following order :-

- USB Module – top side
- USB Module – bottom side
- ADC Module – top side
- ADC Module – bottom side

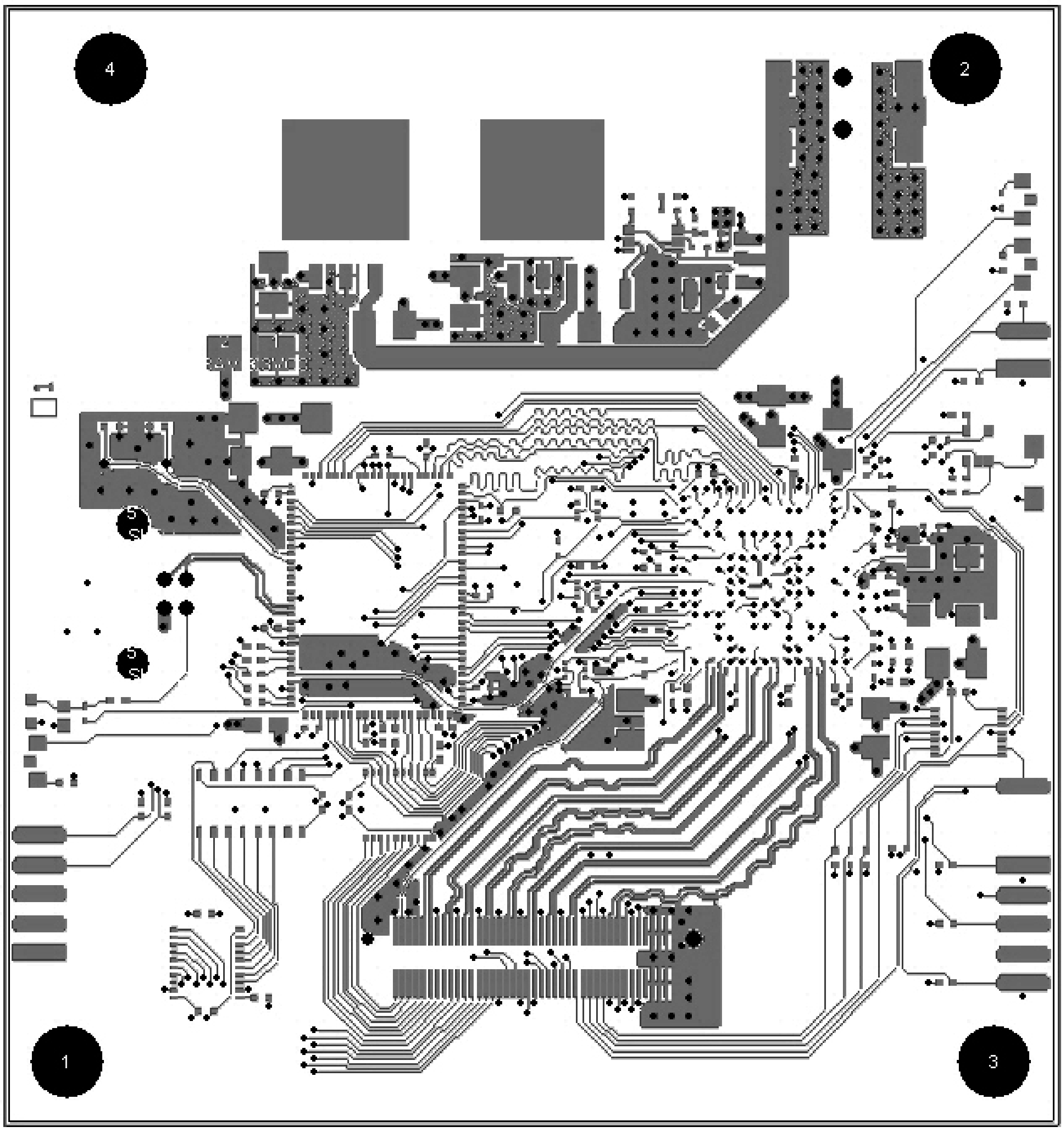


Figure B.1. USB Module top layer

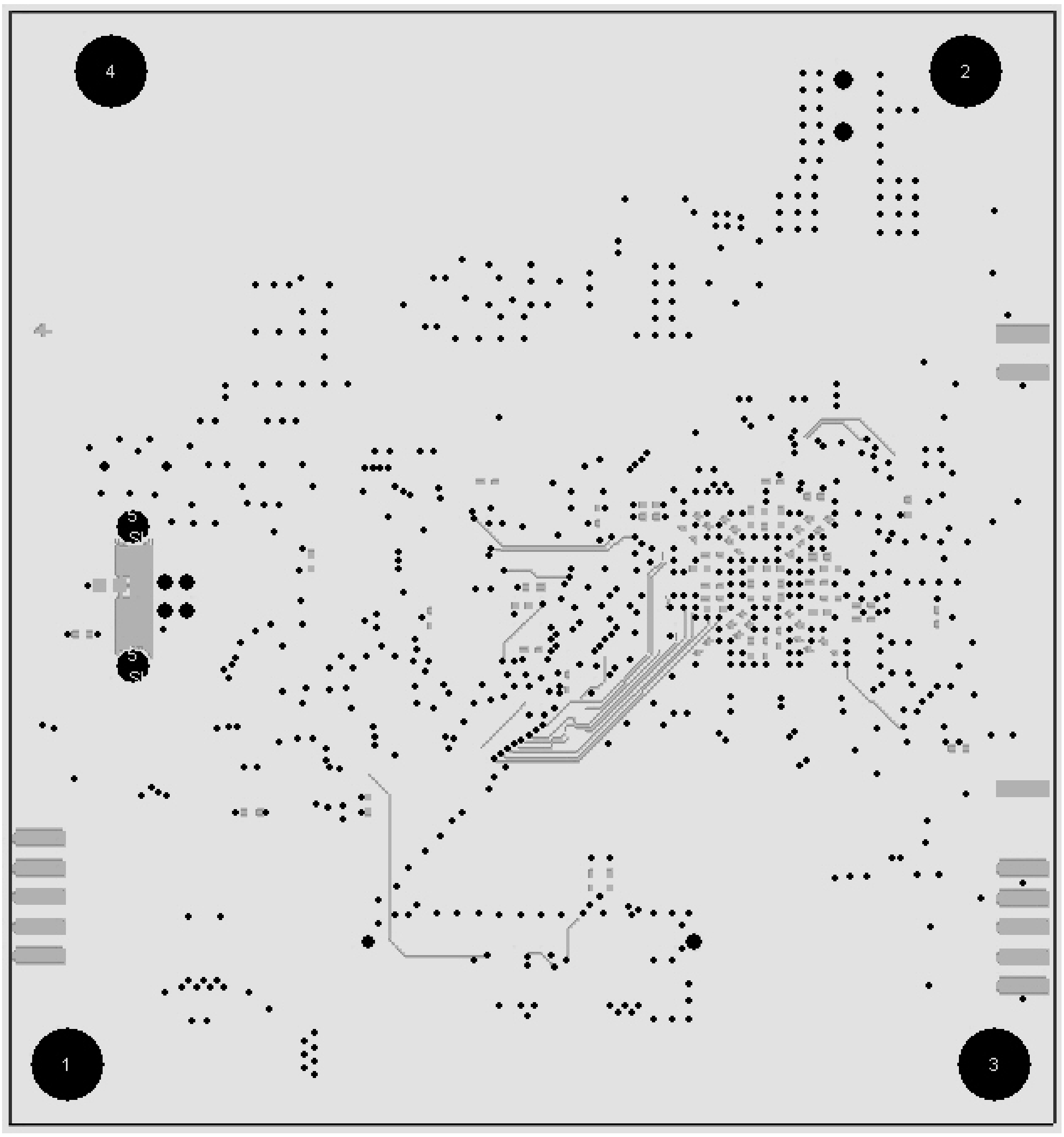


Figure B.2. USB module bottom layer

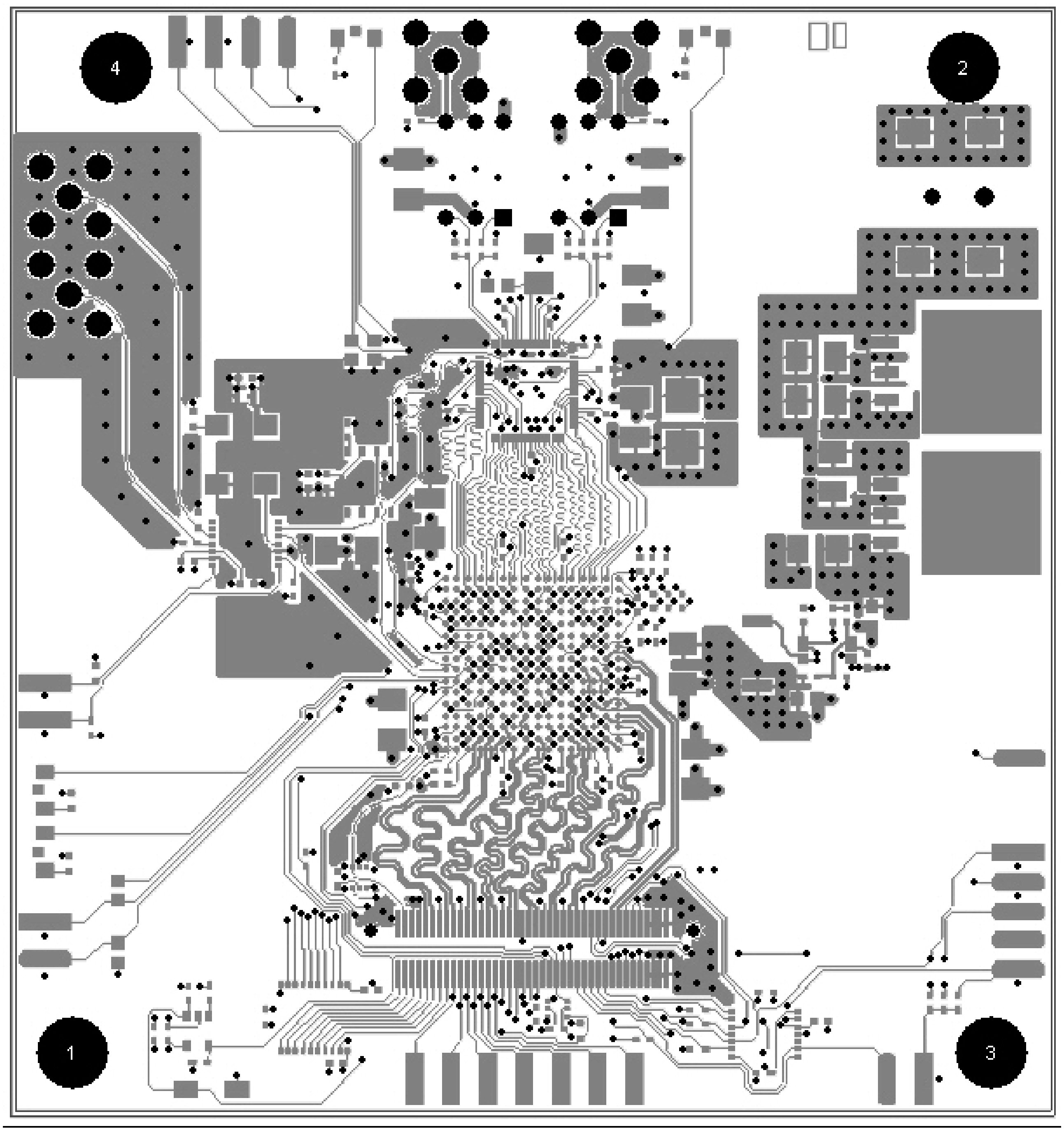


Figure B.3. ADC module top layer

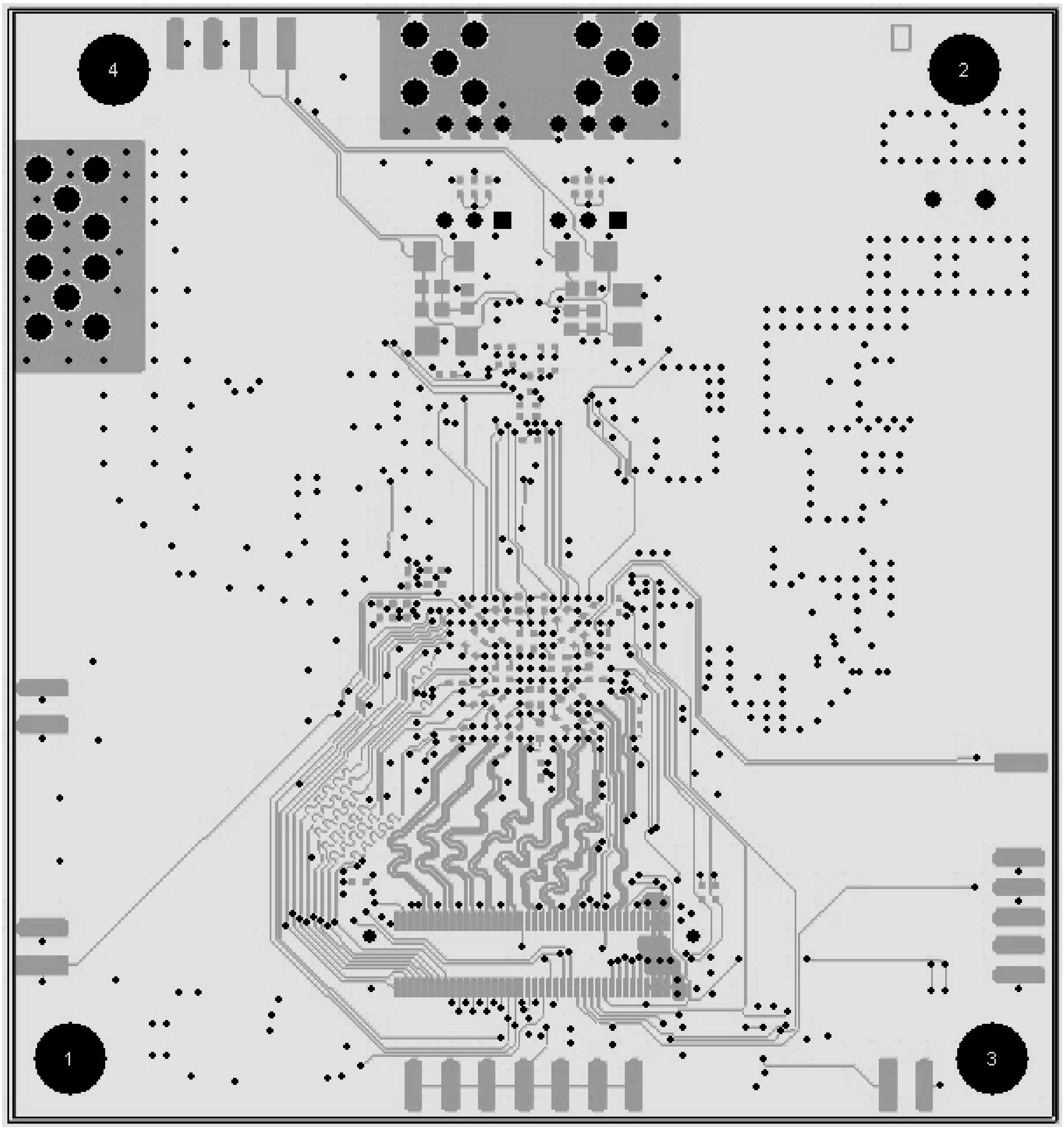


Figure B.4. ADC module bottom layer

Appendix C - Source code listings

This appendix presents source code listings for the main elements of the system.

These are :-

FPGA firmware
- USB Module firmware
- ADC Module firmware

FX2 firmware
- Main FX2 program

Host PC application code
- Host-FX2 interface library code
- Streaming data test code

USB Module firmware (SDR_USB_Core.vhd)

This firmware accepts ADC samples from the ADC module via the LVDS data bus. It then buffers these, crossing them over to the FX2 clock domain (*IFCLK*). The samples are again buffered and fed up to the FX2 on receiving USB data requests.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity SDR_USB_Core is
       port (
               --system clock
               ifclk: in std_logic;
               rst: in std_logic;

               --GPIF control signals
               data: inout std_logic_vector(15 downto 0);
               ctl: in std_logic_vector (5 downto 0);
               rdy: out std_logic_vector(5 downto 0);

               fx2_reset: out std_logic;

               --lvds bus
               B0_P: in std_logic;
               B0_N: in std_logic;
               B1_P: in std_logic;
               B1_N: in std_logic;
               B2_P: in std_logic;
               B2_N: in std_logic;
               B3_P: in std_logic;
               B3_N: in std_logic;
               B4_P: in std_logic;
               B4_N: in std_logic;
               B5_P: in std_logic;
               B5_N: in std_logic;
               B6_P: in std_logic;
               B6_N: in std_logic;
               B7_P: in std_logic;
               B7_N: in std_logic;

               --lvds clock
               COMMS_CLK_P_OUT: in std_logic;
               COMMS_CLK_N_OUT: in std_logic;

               --comms control
               --COMMS_CTL0_OUT: out std_logic;
               COMMS_CTL1_OUT: out std_logic;

               --debug leds
               led: out std_logic_vector(1 downto 0)
       );
end SDR_USB_Core;

architecture Behavioral of SDR_USB_Core is

--DCM component (multiplies clock by 2)
component CLK_2X_MUL_DCM
   port (
       CLKIN_IN        : in    std_logic;
       RST_IN          : in    std_logic;
       CLKIN_IBUFG_OUT : out   std_logic;
       CLK0_OUT        : out   std_logic;
       CLK2X_OUT       : out   std_logic;
       LOCKED_OUT      : out   std_logic
       );
end component;

--lvds bus component
component lvds_bus_in_8 is
       port (
               data_in_p: in std_logic_vector(7 downto 0);
               data_in_n: in std_logic_vector(7 downto 0);
               data_out: out std_logic_vector(7 downto 0)
       );
end component;

--lvds driver
component obufds is
       generic(
                       IOSTANDARD: string := "LVDS_25"
               );
       port (
               I: in std_logic;
               O: out std_logic;
               OB: out std_logic
       );
end component;

--lvds receiver
component ibufds is
       generic(
                       IOSTANDARD: string := "LVDS_25"
               );
       port (
               I: in std_logic;
               IB: in std_logic;
               O: out std_logic
       );
end component;

--this async fifo crosses incoming samples from the ADC_module's clock domain to the USB_Module's
--it also provides a buffering stage
component fifo_x16 IS
       port (
       din: IN std_logic_VECTOR(15 downto 0);
       wr_en: IN std_logic;
       wr_clk: IN std_logic;
       rd_en: IN std_logic;
       rd_clk: IN std_logic;
       ainit: IN std_logic;
       dout: OUT std_logic_VECTOR(15 downto 0);
       full: OUT std_logic;
       empty: OUT std_logic;
       almost_empty: OUT std_logic);
END component;

signal sys_rst: std_logic;    --system-wide reset signal
signal clk_divider: std_logic;

--diagnostic ramp signals
signal counter: std_logic_vector(15 downto 0);             --16 bit counter
signal counter_8bit: std_logic_vector (7 downto 0); --8 bit counter from ADC module (diagnostic)
signal counter_16_bit: std_logic_vector(15 downto 0);

--lvds signals
signal lvds_data_in: std_logic_vector(7 downto 0);
signal sample_data: std_logic_vector(7 downto 0);   --16-bit samples

signal diagnostic_tag: std_logic_vector(7 downto 0);

signal output_enable: std_logic;
signal temp:std_logic;

signal ifclk_1x: std_logic;
signal ifclk_2x: std_logic;   --IFCLK multiplied by 2
signal dcm_lock: std_logic;   --DCM lock indicator

signal data_out: std_logic_vector(15 downto 0);     --output data bus (GPIF)

signal byte_toggle: std_logic;

--sample fifo signals
signal sample_fifo_din:       std_logic_vector(15 downto 0);
signal sample_fifo_wen:       std_logic;
signal adc_module_clk:        std_logic;
signal adc_module_clk_div_2:  std_logic;
signal sample_fifo_ren:       std_logic;
signal sample_fifo_read_toggle: std_logic;
signal sample_fifo_ren_d:     std_logic;
signal sample_fifo_dout:      std_logic_vector(15 downto 0);
signal sample_fifo_almost_empty: std_logic;
signal sample_fifo_full: std_logic;

--reconstrucion signals
signal sample_data_8_bit: std_logic_vector(7 downto 0);
signal sample_data_16_bit: std_logic_vector(15 downto 0);
signal sample_data_16_bit_val: std_logic;

begin

--FX2 GPIF control signals
---------------------------------------
-- GPIF Ctrl Outputs
-- CTL 0    = WEN#
-- CTL 1    = REN#
-- CTL 2    = OE#
-- CTL 3    = CLRST
-- CTL 4    = unused
-- CTL 5    = BOGUS

-- GPIF Rdy Inputs
-- RDY0     = EF#
-- RDY1     = FF#
-- RDY2     = unused
-- RDY3     = unused
-- RDY4     = unused
-- RDY5     = TCXpire (internal)
----------------------------------------

       --system reset will hold logic until DCM has achieved lock
       sys_rst <= '1' when (rst = '1') else
                       '1' when (dcm_lock = '0') else
                       '0';

       --feed through reset signal to ADC module
       COMMS_CTL1_OUT <= sys_rst;

       --multiply IFCLK by 2 to allow for DDR transfers
       IFCLK_DCM: CLK_2X_MUL_DCM
          port map
              (
              CLKIN_IN              => ifclk,
              RST_IN                => rst,
              CLKIN_IBUFG_OUT       => ifclk_1x,
              CLK0_OUT              => open,
              CLK2X_OUT             => ifclk_2x,
              LOCKED_OUT            => dcm_lock
              );



       process(sys_rst, ifclk_1x)
       begin
              if (sys_rst = '1') then
                      data <= (others => 'Z');
              elsif(rising_edge(ifclk_1x)) then
                      if ((ctl(1) = '1')and(ctl(2) = '1')) then
                              data <= sample_fifo_dout;--counter_16_bit;
                      else
                              data <= (others => 'Z');
                      end if;
              end if;
       end process;

       --debug leds
       led(0) <= sample_fifo_wen;
       led(1) <= adc_module_clk;

       --DO NOT ASSERT FX2/SPARTAN3 reset line!!
       fx2_reset <= '1';

---------------------------------------------------------------------------------
--                      LVDS interface
---------------------------------------------------------------------------------


       --LVDS clock recovery
  lvds_clk : IBUFDS
       port map (
       O => adc_module_clk,  -- Clock buffer output
       I => COMMS_CLK_P_OUT,  -- Diff_p clock buffer input (connect to top-level port)
       IB => COMMS_CLK_N_OUT -- Diff_n clock buffer input (connect directly to top-level port)
       );


       --LVDS bus interface
       lvds_interface: lvds_bus_in_8
              port map (
                      data_in_p(0)   => B0_P,
                      data_in_p(1)   => B1_P,
                      data_in_p(2)   => B2_P,
                      data_in_p(3)   => B3_P,
                      data_in_p(4)   => B4_P,
                      data_in_p(5)   => B5_P,
                      data_in_p(6)   => B6_P,
                      data_in_p(7)   => B7_P,

                      data_in_n(0)   => B0_N,
                      data_in_n(1)   => B1_N,
                      data_in_n(2)   => B2_N,
                      data_in_n(3)   => B3_N,
                      data_in_n(4)   => B4_N,
                      data_in_n(5)   => B5_N,
                      data_in_n(6)   => B6_N,
                      data_in_n(7)   => B7_N,

                      data_out              => lvds_data_in
              );

       --This process latches in the incoming lvds data (byte-wide)
       process(adc_module_clk, sys_rst)
       begin
              if (sys_rst = '1') then
                      sample_data_8_bit <= (others => '0');
              elsif(falling_edge(adc_module_clk)) then
                      sample_data_8_bit <= lvds_data_in;
              end if;
       end process;

       store_samples: process(adc_module_clk,sys_rst)--adc_module_clk, sys_rst)
       begin
```

```
		if (sys_rst = '1') then
			sample_data_16_bit <= (others => '0');
			sample_data_16_bit_val <= '0';
			byte_toggle <= '0';
			counter_16_bit <= (others => '0');

		elsif(rising_edge(adc_module_clk)) then--adc_module_clk)) then
			sample_data_16_bit <= sample_data_8_bit & "00000000";
			sample_data_16_bit_val <= '1';
		end if;
	end process;


	sample_fifo_din <= sample_data_16_bit;
	sample_fifo_wen <= sample_data_16_bit_val;

	--this process handles the fifo read enable signals
	process(ifclk_2x, sys_rst)
	begin
		if (sys_rst = '1') then
			sample_fifo_ren <= '0';
			sample_fifo_read_toggle <= '0';

		elsif(rising_edge(ifclk_2x)) then
			if ((ctl(1) = '1') and (ctl(2) = '1')) then
				if (sample_fifo_read_toggle = '0') then
					sample_fifo_ren <= '1';
					sample_fifo_read_toggle <= '1';
				else
					sample_fifo_ren <= '0';
					sample_fifo_read_toggle <= '0';
				end if;
			else
				sample_fifo_ren <= '0';
				sample_fifo_read_toggle <= '0';
			end if;
		end if;
	end process;

	sample_fifo: fifo_x16
	port map (
		din		=> sample_fifo_din,
		wr_en		=> sample_fifo_wen,
		wr_clk		=> adc_module_clk,
		rd_en		=> sample_fifo_ren,
		rd_clk		=> ifclk_2x,
		ainit		=> sys_rst,
		dout		=> sample_fifo_dout,
		full		=> sample_fifo_full,
		empty		=> open,
		almost_empty	=> sample_fifo_almost_empty
		);

	--full flag RDY[0] always deasserted (for now)
	rdy(0) <= '1';		--full flag
	rdy(1) <= not sample_fifo_almost_empty;		--empty flag
	rdy(5 downto 3) <= (others => 'Z');  --RDY[5:2] unused
end Behavioral;
```

ADC Module firmware (ADC_Core.vhd)

This firmware reads in samples directly from the ADC data buses. It decimates the sample stream by a pre-determined factor, and stores the data in a 16K FIFO. All logic runs in the ADC clock domain, and this clock is re-transmitted to the USB module for synchronous data reception. The FIFO is continually emptied with data constantly streaming to the USB module.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

   entity ADC_Core is
       port (
              --debug/status leds
              led: out std_logic_vector(1 downto 0);

              --ADC interface
              --reduced sample resolution to 8-bits - for now
              ADC_A4: in std_logic;
              ADC_A5: in std_logic;
              ADC_A6: in std_logic;
              ADC_A7: in std_logic;
              ADC_A8: in std_logic;
              ADC_A9: in std_logic;
              ADC_A10: in std_logic;
              ADC_A11: in std_logic;
              ADC_A12: in std_logic;

              CLK_A: in std_logic;  --ADC clock
              CLK_A_2X: in std_logic;      --2x ADC clock

              PWDN_A: out std_logic;
              PWDN_B: out std_logic;

              OTR_A: in std_logic;
              OTR_B: in std_logic;

              OEB_A: out std_logic;
              OEB_B: out std_logic;

              --LVDS bus interface
              A0_P: out std_logic;
              A0_N: out std_logic;
              A1_P: out std_logic;
              A1_N: out std_logic;
              A2_P: out std_logic;
              A2_N: out std_logic;
              A3_P: out std_logic;
              A3_N: out std_logic;
              A4_P: out std_logic;
              A4_N: out std_logic;
              A5_P: out std_logic;
              A5_N: out std_logic;
              A6_P: out std_logic;
              A6_N: out std_logic;
              A7_P: out std_logic;
              A7_N: out std_logic;

              --comms control
              --COMMS_CTL0_IN: in std_logic;
              COMMS_CTL1_IN: in std_logic;

              --LVDS clock
              COMMS_CLK_P_IN: out std_logic;
              COMMS_CLK_N_IN: out std_logic

              --generic ports
              --IO_PORT: in std_logic_vector(1 downto 0)
       );
```

```
end ADC_Core;

architecture Behavioral of ADC_Core is

--components
component lvds_bus_8 is
       port (
              data_in: in std_logic_vector(7 downto 0);
              data_out_p: out std_logic_vector(7 downto 0);
              data_out_n: out std_logic_vector(7 downto 0)
       );
end component;

--lvds receiver
component ibufds is
       generic (
              IOSTANDARD: string := "LVDS_25"
       );
       port (
              I: in std_logic;
              IB: in std_logic;
              O: out std_logic
       );
end component;

--lvds driver
component obufds is
       generic (
              IOSTANDARD: string := "LVDS_25"
       );
       port (
              I: in std_logic;
              O: out std_logic;
              OB: out std_logic
       );
end component;

--buffers the samples before splitting into bytes for transmission
component sync_fifo_x16 IS
       port (
       clk: IN std_logic;
       sinit: IN std_logic;
       din: IN std_logic_VECTOR(15 downto 0);
       wr_en: IN std_logic;
       rd_en: IN std_logic;
       dout: OUT std_logic_VECTOR(15 downto 0);
       full: OUT std_logic;
       empty: OUT std_logic;
       data_count: OUT std_logic_VECTOR(1 downto 0));
END component;

--DCM instantiation for doubling a clock signal
component CLK_2X_DCM is
   port ( CLKIN_IN        : in    std_logic;
          RST_IN          : in    std_logic;
          CLKIN_IBUFG_OUT : out   std_logic;
          CLK0_OUT        : out   std_logic;
          CLK2X_OUT       : out   std_logic;
          LOCKED_OUT      : out   std_logic);
end component;

--signals

--clock derived from COMMS_CLK
signal sys_clk: std_logic;
signal sys_clk_buf: std_logic;
signal sys_rst: std_logic;
signal external_rst: std_logic;

signal sys_clk_2x: std_logic;
signal sys_clk_dcm_lock: std_logic;

--data bus signals
signal data_out: std_logic_vector(7 downto 0);

--ramp signals
signal ramp_data: std_logic_vector(7 downto 0);
signal ramp_data_buf: std_logic_vector(15 downto 0);
signal ramp_clk_div: std_logic;

signal counter_toggle: std_logic;
signal counter_16_bit: std_logic_vector(15 downto 0);

signal test_data: std_logic_vector(7 downto 0);
signal test_data_byte_toggle: std_logic;

--adc signals
signal adc_clk: std_logic;
signal adc_clk_2x: std_logic;
signal adc_clk_div_2: std_logic;
signal adc_clk_div_4: std_logic;
signal adc_dcm_lock: std_logic;

--sample buffer signals
signal sample_fifo_din: std_logic_vector(7 downto 0);
signal sample_fifo_wen: std_logic;
signal sample_fifo_ren: std_logic;
signal sample_fifo_ren_d: std_logic;
signal sample_fifo_dout: std_logic_vector(7 downto 0);
signal sample_fifo_full: std_logic;
signal sample_fifo_empty: std_logic;
signal sample_fifo_data_count: std_logic_vector(1 downto 0);

--ADC data input signals
signal adc_a_data: std_logic_vector(7 downto 0);

--decimator signals
signal adc_a_dec: std_logic_vector(7 downto 0);
signal adc_a_dec_val: std_logic;
signal dec_toggle: std_logic;

signal adc_data_8bit: std_logic_vector(7 downto 0);
signal adc_data_8bit_val: std_logic;

signal byte_toggle: std_logic;

type fifo_state_t is (IDLE,
                              READ_SAMPLE,
                              SEND_SAMPLE);

signal fifo_state: fifo_state_t;

begin

       --debug/status leds
       led(1) <= sample_fifo_full;
       led(0) <= sample_fifo_empty;

       --ADC module can be reset externally by the USB module
       external_rst <= COMMS_CTL1_IN;

       --system-wide reset (wait till DCM has locked)
       sys_rst <= '1' when (external_rst = '1') else
                       '1' when (adc_dcm_lock = '0') else
                       '0';

-------------------------------------------------------------------
       --adc interface
-------------------------------------------------------------------

       double_adc_clk: CLK_2X_DCM
              port map (
                     CLKIN_IN        => CLK_A,
                     RST_IN          => external_rst,
                     CLKIN_IBUFG_OUT => adc_clk,
                     CLK0_OUT        => open,
                     CLK2X_OUT       => adc_clk_2x,
                     LOCKED_OUT      => adc_dcm_lock
                     );

       PWDN_A <= '0';         --enable ADC channel A
       PWDN_B <= '1';         --disable ADC channel B

       OEB_A <= '0';          --enable channel A output
```

```
	OEB_B <= '1';		--disable channel B's output

	--latch in ADC data
	process(adc_clk, sys_rst)
	begin
		if (sys_rst = '1') then
			adc_a_data <= (others => '0');

		elsif(rising_edge(adc_clk)) then
			--latch sample into upper 12 bits
			adc_a_data <=  ADC_A11 &
						ADC_A10 &
						ADC_A9 &
						ADC_A8 &
						ADC_A7 &
						ADC_A6 &
						ADC_A5 &
						ADC_A4;
		end if;
	end process;



	--decimate sample stream
	process(adc_clk, sys_rst)
	begin
		if (sys_rst = '1') then
			adc_a_dec <= (others => '0');
			adc_a_dec_val <= '0';
			dec_toggle <= '0';

		elsif(rising_edge(adc_clk)) then
			adc_a_dec <= adc_a_data;
			adc_a_dec_val <= '1';
		end if;
	end process;

	--buffer the decimated sample stream in a fifo (16Kx8)
	sample_fifo: sync_fifo_x8
	port map(
		clk			=> adc_clk,
		sinit		=> sys_rst,
		din			=> sample_fifo_din,
		wr_en		=> sample_fifo_wen,
		rd_en		=> sample_fifo_ren,
		dout			=> sample_fifo_dout,
		full			=> sample_fifo_full,
		empty		=> sample_fifo_empty,
		data_count	=> sample_fifo_data_count
		);

	--buffer write control
	sample_fifo_din <= adc_a_dec;
	sample_fifo_wen <= adc_a_dec_val;

--------------------------------------------------------------------------------
-------------   lvds interface                                --------------------------

	--LVDS clock generation
   lvds_clk : obufds
	port map (
	I => adc_clk,--adc_clk_div_4,                       -- Clock buffer output
	O => COMMS_CLK_P_IN,  -- Diff_p clock buffer input (connect to top-level port)
	OB => COMMS_CLK_N_IN  -- Diff_n clock buffer input (connect directly to top-level port)
	);

	--data output
	--DIAGNOSTIC
	--free running 16-bit counter for testing
	process(adc_clk, sys_rst)
	begin
		if (sys_rst = '1') then
			counter_16_bit <= (others => '0');
			counter_toggle <= '0';

		elsif(rising_edge(adc_clk)) then
			counter_toggle <= not counter_toggle;
```

```
                if (counter_toggle = '1') then
                    counter_16_bit <= counter_16_bit + 1;
                end if;
            end if;
        end process;

        --this process handles reading out of the sample fifo
        process(adc_clk, sys_rst)
        begin
            if (sys_rst = '1') then
                data_out <= (others => '0');
                sample_fifo_ren <= '0';
                sample_fifo_ren_d <= '0';

            elsif(rising_edge(adc_clk)) then
                --check if there's data in the fifo
                if (sample_fifo_empty = '0') then
                    sample_fifo_ren <= '1';
                else
                    sample_fifo_ren <= '0';
                end if;

                --delayed version of read enable (data latent by 1 clock cycle)
                sample_fifo_ren_d <= sample_fifo_ren;

                if (sample_fifo_ren_d = '1') then
                    data_out <= sample_fifo_dout;
                else
                    data_out <= (others => '0');
                end if;
            end if;
        end process;


        --LVDS bus
        data_bus: lvds_bus_8
            port map (
                data_in => data_out,

                data_out_p(7)  => A7_P,
                data_out_p(6)  => A6_P,
                data_out_p(5)  => A5_P,
                data_out_p(4)  => A4_P,
                data_out_p(3)  => A3_P,
                data_out_p(2)  => A2_P,
                data_out_p(1)  => A1_P,
                data_out_p(0)  => A0_P,

                data_out_n(7)  => A7_N,
                data_out_n(6)  => A6_N,
                data_out_n(5)  => A5_N,
                data_out_n(4)  => A4_N,
                data_out_n(3)  => A3_N,
                data_out_n(2)  => A2_N,
                data_out_n(1)  => A1_N,
                data_out_n(0)  => A0_N
            );
end Behavioral;
```

FX2 firmware (USRP_MAIN.C)

This firmware intializes the FX2, and its GPIF circuitry. It primes the GPIF for data requests from the USB module itself. It also handles USB renumeration and all standard USB control requests, providing the host with the required information.

It was originally written by the *GnuRadio* team, followed by several modifications by Larry Doolittle and the author.

```
/*
 * USRP - Universal Software Radio Peripheral
 *
 * Copyright (C) 2003,2004 Free Software Foundation, Inc.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include "usrp_common.h"
#include "usrp_commands.h"
#include "fpga.h"
#include "usrp_gpif_inline.h"
#include "timer.h"
#include "i2c.h"
#include "isr.h"
#include "usb_common.h"
#include "fx2utils.h"
#include "usrp_avnet_regs.h"
#include "usrp_globals.h"
#include <string.h>
#include "spi.h"


#define bRequestType   SETUPDAT[0]
#define bRequest       SETUPDAT[1]
#define wValueL SETUPDAT[2]
#define wValueH SETUPDAT[3]
#define wIndexL SETUPDAT[4]
#define wIndexH SETUPDAT[5]
#define wLengthL       SETUPDAT[6]
#define wLengthH       SETUPDAT[7]


unsigned char g_tx_enable = 0;
unsigned char g_rx_enable = 0;
unsigned char g_rx_overrun = 0;
unsigned char g_tx_underrun = 0;

/*
 * the host side fpga loader code pushes an MD5 hash of the bitstream
 * into hash1.
 */
#define USRP_HASH_SIZE        16
xdata at       USRP_HASH_SLOT_1_ADDR unsigned char hash1[USRP_HASH_SIZE];

static void get_ep0_data (void)
{
       EP0BCL = 0;                   // arm EP0 for OUT xfer.  This sets the busy bit
       while (EP0CS & bmEPBUSY)      // wait for busy to clear
       ;
}
```

```
/*
 * Handle our "Vendor Extension" commands on endpoint 0.
 * If we handle this one, return non-zero.
 */
unsigned char app_vendor_cmd (void)
{
	OEE = 0x0f;
	IOE = 0x04;  IOE = 0x01;  // pulse two I/O pins
	if (bRequestType == VRT_VENDOR_IN)
	{

		/////////////////////////////////
		//    handle the IN requests
		/////////////////////////////////

		switch (bRequest)
		{

			case VRQ_GET_STATUS:
				switch (wIndexL)
				{
					case GS_TX_UNDERRUN:
						EP0BUF[0] = g_tx_underrun;
						g_tx_underrun = 0;
						EP0BCH = 0;
						EP0BCL = 1;
					break;

					case GS_RX_OVERRUN:
						EP0BUF[0] = g_rx_overrun;
						g_rx_overrun = 0;
						EP0BCH = 0;
						EP0BCL = 1;
					break;

				       default:
						return 0;
			       }
			break;

			case VRQ_I2C_READ:
				if (!i2c_read (wValueL, EP0BUF, wLengthL))
					return 0;

			      EP0BCH = 0;
			      EP0BCL = wLengthL;
		       break;

			case VRQ_SPI_READ:
			if (!spi_read (wValueH, wValueL, wIndexH, wIndexL, EP0BUF, wLengthL))
				return 0;

			      EP0BCH = 0;
			      EP0BCL = wLengthL;
		       break;

			default:
			      return 0;
		}
	}
	else if (bRequestType == VRT_VENDOR_OUT)
	{

		/////////////////////////////////
		//    handle the OUT requests
		/////////////////////////////////

		switch (bRequest)
		{
			case VRQ_SET_LED:
				switch (wIndexL)
				{
					case 0:
						set_led_0 (wValueL);
					break;

					case 1:
```

```
					set_led_1 (wValueL);
				break;

				default:
					return 0;
			}
		break;

		case VRQ_FPGA_LOAD:
			switch (wIndexL)
			{			// sub-command
			    case FL_BEGIN:
					return fpga_load_begin ();

				case FL_XFER:
					get_ep0_data ();
					return fpga_load_xfer (EP0BUF, EP0BCL);

				case FL_END:
					return fpga_load_end ();

				default:
					return 0;
			}
		break;

		case VRQ_FPGA_SET_RESET:
			fpga_set_reset (wValueL);
		break;

		case VRQ_FPGA_SET_TX_ENABLE:
			fpga_set_tx_enable (wValueL);
		break;

		case VRQ_FPGA_SET_RX_ENABLE:
			fpga_set_rx_enable (wValueL);
		break;

		case VRQ_FPGA_SET_TX_RESET:
			fpga_set_tx_reset (wValueL);
		break;

		case VRQ_FPGA_SET_RX_RESET:
			fpga_set_rx_reset (wValueL);
		break;

		case VRQ_I2C_WRITE:
			get_ep0_data ();
			if (!i2c_write (wValueL, EP0BUF, EP0BCL))
				return 0;
		break;

		case VRQ_SPI_WRITE:
			get_ep0_data ();
		if (!spi_write (wValueH, wValueL, wIndexH, wIndexL, EP0BUF, EP0BCL))
			return 0;
		break;

		default:
			return 0;
	    }
	}
	else
	    return 0;	// invalid bRequestType

	return 1;
}

static void main_loop (void)
{
	setup_flowstate_common ();

	while (1)
	{
		OEE = 0x0f;
		IOE = 0x02;  IOE = 0x01;  // pulse two I/O pins
		if (usb_setup_packet_avail ())
```

```
                        usb_handle_setup_packet ();

                if (GPIFTRIG & bmGPIF_IDLE)
                {
                        // OK, GPIF is idle.  Let's try to give it some work.
                        // First check for underruns and overruns
#if 1
      //if (UC_BOARD_HAS_FPGA && (USRP_PA & (bmPA_TX_UNDERRUN | bmPA_RX_OVERRUN))){
                        if (1)
                        {
                                // record the under/over run
                                //if (USRP_PA & bmPA_TX_UNDERRUN)
                                //g_tx_underrun = 1;
                                g_tx_underrun = 0;

                                //if (USRP_PA & bmPA_RX_OVERRUN)
                                //g_rx_overrun = 1;
                                g_rx_overrun = 0;

                                // tell the FPGA to clear the flags
                                fpga_clear_flags ();
                        }
#endif
                        // Next see if there are any "OUT" packets waiting for our attention,
                        // and if so, if there's room in the FPGAs FIFO for them.

/* commented by kw - only handling IN packets for now...

                        if (g_tx_enable && !(EP24FIFOFLGS & 0x02))
                        {  // USB end point fifo is not empty...

                                if (fpga_has_room_for_packet ())
                                {         // ... and FPGA has room for packet
                                        GPIFTCB1 = 0x01;      SYNCDELAY;
                                        GPIFTCB0 = 0x00;      SYNCDELAY;
                                        setup_flowstate_write ();
                                        SYNCDELAY;

                                        // start the xfer
                                        GPIFTRIG = bmGPIF_EP2_START | bmGPIF_WRITE;
                                        SYNCDELAY;

                                        while (!(GPIFTRIG & bmGPIF_IDLE))
                                        {
                                                // wait for the transaction to complete
                                        }
                                }
                        }
*/
                        // See if there are any requests for "IN" packets, and if so
                        // whether the FPGAs got any packets for us.

                        // USB end point fifo is not full...
                        if (g_rx_enable && !(EP6CS & bmEPFULL))
                        {
                                // ... and FPGA has packet available
                                if (fpga_has_packet_avail ())
                                {
                                        IOE = 0x08;  // signal start of transfer
                                        GPIFTCB1 = 0x01;      SYNCDELAY;
                                        GPIFTCB0 = 0x00;      SYNCDELAY;
                                        setup_flowstate_read ();
                                        SYNCDELAY;
                                        // start the xfer
                                        GPIFTRIG = bmGPIF_EP6_START | bmGPIF_READ;
                                        SYNCDELAY;

                                        while (!(GPIFTRIG & bmGPIF_IDLE))
                                        {
                                                // wait for the transaction to complete
                                        }
                                        SYNCDELAY;
                                        // tell USB we filled buffer (6 is our endpoint num)
                                        INPKTEND = 6;
                                        IOE = 0x01;  // signal end of transfer
                                }
                        }
```

```
		}
	}
}


/*
 * called at 100 Hz from timer2 interrupt
 *
 * Toggle led 0
 */
void isr_tick (void) interrupt
{
	static unsigned char count = 1;
	if (--count == 0)
	{
		count = 50;
		// USRP_LED_REG ^= bmLED0;
	}
	clear_timer_irq ();
}

void main (void)
{
//added by kw - must reset the FPGA logic before trying to do data transfers (DCM specifically)
	PORTCCFG	= 0x00;
	OEC			= 0x10;	//enable PC4

	IOC			= 0x10;	//assert PC4 to reset logic
	IOC			= 0x10;	//assert PC4 to reset logic
	IOC			= 0x10;	//assert PC4 to reset logic
	IOC			= 0x10;	//assert PC4 to reset logic
	IOC			= 0x10;	//assert PC4 to reset logic
	IOC			= 0x10;	//assert PC4 to reset logic
	IOC			= 0x10;	//assert PC4 to reset logic

	IOC			= 0x00;	//deassert PC4
	OEC			= 0x00;	//PORT output disabled

#if 0
	g_rx_enable = 0;	// FIXME (work around initialization bug)
	g_tx_enable = 0;
	g_rx_overrun = 0;
	g_tx_underrun = 0;
#endif

	PORTECFG = 0x00;
	OEE = 0x0f;
	IOE = 0x0c;

	// zero fpga bitstream hash.  This forces reload
	memset (hash1, 0, USRP_HASH_SIZE);

	init_usrp ();
	init_gpif ();

	// if (UC_START_WITH_GSTATE_OUTPUT_ENABLED)
	// IFCONFIG |= bmGSTATE;			// no conflict, start with it on

	//commented by kw
	//set_led_0 (0);
	//set_led_1 (0);

	EA = 0;		// disable all interrupts

	setup_autovectors ();
	usb_install_handlers ();
	hook_timer_tick ((unsigned short) isr_tick);

	EIEX4 = 1;		// disable INT4 FIXME
	EA = 1;		// global interrupt enable

	IOE = 0x04;
	fx2_renumerate ();	// simulates disconnect / reconnect
	IOE = 0x08;

	main_loop ();
}
```

FX2 low-level interface library (USRP_PRIMS.CC)

This library code forms the direct interface with the FX2 as a set of low-level functions. Data transfers and various control requests are passed to the FX2 via USB *endpoint 0*, while the data received is sent back via *endpoint 2*.

The code was written by the *GnuRadio* team, with several modifications by Larry Doolittle and the author. Note that only those main methods relevant to this project are included here.

```
/* -*- c++ -*- */
/*
 * Copyright 2003,2004 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "usrp_prims.h"
#include "usrp_commands.h"
#include "usrp_ids.h"
#include "usrp_i2c_addr.h"
#include <usb.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>             // FIXME should check with autoconf (nanosleep)
#include <algorithm>
#include <assert.h>

#define VERBOSE 0

void usrp_one_time_init ()
{
       static bool first = true;

       printf("usrp_one_time_init: Entering function\n");
       if (first)
       {
              first = false;
              printf("usrp_one_time_init: Calling usb_init\n");
              usb_init ();                 // usb library init

              printf("usrp_one_time_init: Calling usb_find_busses\n");
              usb_find_busses ();

              printf("usrp_one_time_init: Calling usb_find_devices\n");
              usb_find_devices ();
       }
       printf("usrp_one_time_init: Leaving function\n");
}

// ----------------------------------------------------------------
// Danger, big, fragile KLUDGE.  The problem is that we want to be
// able to get from a usb_dev_handle back to a usb_device, and the
// right way to do this is buried in a non-installed include file.

static struct usb_device *dev_handle_to_dev (usb_dev_handle *udh)
{
	struct usb_dev_handle_kludge
	{
		int	fd;
		struct usb_bus *bus;
		struct usb_device	*device;
	};
	return ((struct usb_dev_handle_kludge *) udh)->device;
}

// ----------------------------------------------------------------

bool usrp_fx2_p (struct usb_device *q)
{
	return (q->descriptor.idVendor == USB_VID_CYPRESS
	  && q->descriptor.idProduct == USB_PID_CYPRESS_FX2);
}

bool usrp_SDR_p(struct usb_device *q)
{
	printf("usrp_SDR_p: Vendor ID [0x%X] Product ID [0x%X]\n", q->descriptor.idVendor, q-
>descriptor.idProduct);
	//search for vendor id = SDR_MODULE (99)
	return ((unsigned)q->descriptor.idVendor == USB_VID_FSF
	  && (unsigned)q->descriptor.idProduct == USB_PID_FSF_SDR_MODULE);
}

// ----------------------------------------------------------------

//Searches for an FX2 device (as opposed to USRP)
struct usb_device *fx2_find_device (int nth)
{
	struct usb_bus *p;
	struct usb_device *q;
	int	 n_found = 0;

	printf("Entering fx2_find_device...\n");

	printf("Calling usrp_one_time_init...\n");
	usrp_one_time_init ();
	printf("Left usrp_one_time_init...\n");

	printf("Calling usb_get_busses\n");
	p = usb_get_busses();

	//kw - diagnostic
	if (p == NULL)
	{
		printf("usb_get_busses returned NULL pointer\n");
	}
	else
	{
		printf("Left usb_get_busses\n");
	}

	printf("Scanning through USB devices...\n");

	while (p != NULL)
	{
		q = p->devices;
		while (q != NULL)
		{
			//if (usrp_fx2_p (q) ){
			if (usrp_SDR_p (q) )
			{
				if (n_found == nth)	// return this one
					//kw - diagnostic
					printf("KW - found SDR-module device!\n");
				return q;
				n_found++;		// keep looking
			}
			q = q->next;
		}
		p = p->next;
	}
	return 0;	// not found
	printf("Leaving fx2_find_device\n");
}

static struct usb_dev_handle *usrp_open_interface (struct usb_device *dev, int interface, int
altinterface)
{
	printf("usrp_open_interface - Entering function with interface [0x%X], alt interface
[0x%X]\n",(unsigned) interface, (unsigned) altinterface);

	struct usb_dev_handle *udh = usb_open (dev);
	if (udh == 0)
	{
		printf("usrp_open_interface - failed to open USB device (usb_open)\n");
		return 0;
	}
	else
	{
		printf("usrp_open_interface - open USB device returned positive
result!(usb_open)\n");
	}

	if (dev != dev_handle_to_dev (udh))
	{
		fprintf (stderr, "%s:%d: internal error!\n", __FILE__, __LINE__);
		abort ();
	}

	printf("usrp_open_interface - skipping set configuration...\n");
/* commented out by kw
	if (usb_set_configuration (udh, 1) < 0)
	{
		printf("usrp_open_interface - failed to set USB configuration\n");
		fprintf (stderr, "%s: usb_claim_interface: failed conf %d\n",__FUNCTION__,interface);
		fprintf (stderr, "%s\n", usb_strerror());
		usb_close (udh);
		return 0;
	}
*/

	printf("usrp_open_interface - attempting to claim interface\n");
	if (usb_claim_interface (udh, interface) < 0)
	{
		printf("usrp_open_interface - claim interface failed\n");
		fprintf (stderr, "%s:usb_claim_interface: failed interface %d\n",
__FUNCTION__,interface);
		fprintf (stderr, "%s\n", usb_strerror());
		usb_close (udh);
	return 0;
	}
	else
	{
		printf("usrp_open_interface - claim interface succeeded!\n");
	}

	printf("usrp_open_interface - attempting to set alt interface\n");
	if (usb_set_altinterface (udh, altinterface) < 0)
	{
		printf("usrp_open_interface - failed to set alt interface\n");
		fprintf (stderr, "%s:usb_set_alt_interface: failed\n", __FUNCTION__);
		fprintf (stderr, "%s\n", usb_strerror());
		usb_release_interface (udh, interface);
		usb_close (udh);
		return 0;
	}
	else
	{
		printf("usrp_open_interface - set alt interface succeeded!\n");
	}
	return udh;
}

struct usb_dev_handle *usrp_open_cmd_interface (struct usb_device *dev)
{
        printf("Calling usrp_open_cmd_interface\n");
        return usrp_open_interface (dev, USRP_CMD_INTERFACE, USRP_CMD_ALTINTERFACE);
}

struct usb_dev_handle *usrp_open_rx_interface (struct usb_device *dev)
{
        printf("Calling usrp_open_rx_interface\n");
        return usrp_open_interface (dev, USRP_RX_INTERFACE, USRP_RX_ALTINTERFACE);
}

bool usrp_close_interface (struct usb_dev_handle *udh)
{
        // we're assuming that closing an interface automatically releases it.
        return usb_close (udh) == 0;
}

// ----------------------------------------------------------------
// write vendor extension command to USRP
static int write_cmd (struct usb_dev_handle *udh,
           int request, int value, int index,
           unsigned char *bytes, int len)
{
        int     requesttype = (request & 0x80) ? VRT_VENDOR_IN : VRT_VENDOR_OUT;

        if (requesttype == VRT_VENDOR_IN)
                printf("write_cmd - VENDOR IN request type\n");
        else
                printf("write_cmd - VENDOR OUT request type\n");

        int r = usb_control_msg (udh, requesttype, request, value, index,
                           (char *) bytes, len, 1000);
        if (r < 0)
        {
                printf("Error encountered during control message write to FX2\n");
                // we get EPIPE if the firmware stalls the endpoint.
                if (errno != EPIPE)
                {
                printf("write_cmd - the following error was encountered during USB control message
write [%s]\n", usb_strerror());
                fprintf (stderr, "usb_control_msg failed: %s\n", usb_strerror ());
                }
        }
        else
        {
                printf("write_cmd - Control message write to FX2 returned positive value\n");
        }
        return r;
}

static bool usrp_set_switch (struct usb_dev_handle *udh, int cmd_byte, bool on)
{
        printf("usrp_set_switch - calling write_cmd (0x%X)\n", (unsigned)cmd_byte);
        return write_cmd (udh, cmd_byte, on, 0, 0, 0) == 0;
}
bool usrp_set_fpga_rx_enable (struct usb_dev_handle *udh, bool on)
{
        printf("usrp_set_fpga_rx_enable - calling usrp_set_switch \n");
        return usrp_set_switch (udh, VRQ_FPGA_SET_RX_ENABLE, on);
}
```

FX2 application-level interface library (USRP_BASIC.CC)

This library presents a higher-level interface to the FX2, in the form of object-oriented methods which call the functions listed above in usrp_prims.cc.

It was written by the *GnuRadio* team, and modified by Larry Doolittle and the author. Only those methods relevant to the project are shown here.

```
/* -*- c++ -*- */
/*
 * Copyright 2003,2004 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "usrp_basic.h"
#include "usrp_prims.h"
#include "usrp_interfaces.h"
#include "fpga_regs_common.h"
#include "fusb.h"
#include <usb.h>
#include <stdexcept>
#include <assert.h>
#include <math.h>

#define NELEM(x) (sizeof (x) / sizeof (x[0]))

// These set the buffer size used for each end point using the fast
// usb interface.  The kernel ends up locking down this much memory.

static const int FUSB_BUFFER_SIZE = 2 * (1L << 20); // 2 MB (was 8 MB)
static const int FUSB_BLOCK_SIZE = fusb_sysconfig::max_block_size();
static const int FUSB_NBLOCKS    = FUSB_BUFFER_SIZE / FUSB_BLOCK_SIZE;
static const double POLLING_INTERVAL = 0.1; // seconds

////////////////////////////////////////////////////////////////

static struct usb_dev_handle *open_rx_interface (struct usb_device *dev)
{
       printf("Entering (usb_dev_handle) open_rx_interface\n");
       struct usb_dev_handle *udh = usrp_open_rx_interface (dev);
       if (udh == 0)
       {
              fprintf (stderr, "usrp_basic_rx: can't open rx interface\n");
              usb_strerror ();
       }
       return udh;
}

static struct usb_dev_handle *open_tx_interface (struct usb_device *dev)
{
       struct usb_dev_handle *udh = usrp_open_tx_interface (dev);
       if (udh == 0)
       {
              fprintf (stderr, "usrp_basic_tx: can't open tx interface\n");
```

```
                usb_strerror ();
        }
        return udh;
}

//////////////////////////////////////////////////////////////////
//
//                      usrp_basic
//
////////////////////////////////////////////////////////////////


usrp_basic::usrp_basic (int which_board,
                        struct usb_dev_handle *
                        open_interface (struct usb_device *dev))
        : d_udh (0),
        d_usb_data_rate (16000000),   // SWAG
        d_bytes_per_poll ((int) (POLLING_INTERVAL * d_usb_data_rate)),
        d_verbose (false)
{
        memset (d_fpga_shadows, 0, sizeof (d_fpga_shadows));
        usrp_one_time_init ();

        // FIXME allow subclasses to load own code
        // if (!usrp_load_standard_bits (which_board, false))
        //   throw std::runtime_error ("usrp_basic/usrp_load_standard_bits");

//kw - only check for FX2
//  struct usb_device *dev = usrp_find_device (which_board);
        printf("usrp_basic::usrp_basic constructor - Looking for fx2 device...\n");

        struct usb_device* dev = fx2_find_device(which_board);
        if (dev == 0)
        {
                fprintf (stderr, "usrp_basic: can't find usrp[%d]\n", which_board);
                throw std::runtime_error ("usrp_basic/usrp_find_device");
        }

/* commented out by kw - doesn't check for usrp anymore - so won't throw this exception
  if (!(usrp_usrp1_p (dev) || usrp_usrp2_p (dev))){
    fprintf (stderr, "usrp_basic: sorry, this code only works with Rev 1 and 2 USRPs\n");
    throw std::runtime_error ("usrp_basic/bad_rev");
  }
*/
        printf("usrp_basic::usrp_basic constructor - Calling open_interface...\n");

        if ((d_udh = open_interface (dev)) == 0)
                throw std::runtime_error ("usrp_basic/open_interface");

        printf("usrp_basic::usrp_basic constructor - open_interface succeeded! (no exceptions
thrown)\n");

/* commented out by kw - will handle the USB vendor extensions a bit later (ie add stuff to
usrp_basic, using the write_cmd method)
  // initialize registers that are common to rx and tx
  _write_fpga_reg (FR_MODE, 0);             // ensure we're in normal mode
*/
        printf("Leaving usrp_basic::usrp_basic constructor\n");
}

usrp_basic::~usrp_basic ()
{
        if (d_udh)
                usb_close (d_udh);
}


bool
usrp_basic::start ()
{
        printf("usrp_basic::start - returning true\n");
        return true;          // nop
}

////////////////////////////////////////////////////////////////
//
//                         usrp_basic_rx
```

```
//
/////////////////////////////////////////////////////////////////
usrp_basic_rx::usrp_basic_rx (int which_board)
	: usrp_basic (which_board, open_rx_interface),
	d_devhandle (0), d_ephandle (0),
	d_bytes_seen (0), d_first_read (true),
	d_rx_enable (false)
{
	//kw - diagnostic
	printf("Entering usrp_basic_rx::usrp_basic_rx\n");

	// initialize rx specific registers

//kw - MUST be sorted out - these methods make command calls to USRP (write_cmd in usrp_primc.cc)
/*
  // Reset the rx path and leave it disabled.
  set_rx_enable (false);
  usrp_set_fpga_rx_reset (d_udh, true);
  usrp_set_fpga_rx_reset (d_udh, false);
*/

	// probe_rx_slots (true);
	//kw - diagnostic
	printf("usrp_basic_rx::usrp_basic_rx - calls to fusb\n");

	d_devhandle = fusb_sysconfig::make_devhandle (d_udh);
	d_ephandle = d_devhandle->make_ephandle (USRP_RX_ENDPOINT, true,
					FUSB_BLOCK_SIZE, FUSB_NBLOCKS);
}

usrp_basic_rx::~usrp_basic_rx ()
{
	if (!set_rx_enable (false))
	{
		fprintf (stderr, "usrp_basic_rx: set_fpga_rx_enable failed\n");
		usb_strerror ();
	}
	d_ephandle->stop ();
	delete d_ephandle;
	delete d_devhandle;
}

bool usrp_basic_rx::start ()
{
	printf("usrp_basic_rx::start - calling usrp_basic::start\n");
	if (!usrp_basic::start ())	// invoke parent's method
	{
		printf("usrp_basic_rx::start - usrp_basic::start returned false - exitting
function\n");
		return false;
	}
	else
	{
		printf("usrp_basic_rx::start - usrp_basic::start returned true\n");
	}

	// fire off reads before asserting rx_enable

	printf("usrp_basic_rx::start - attempting to start endpoint streaming\n");

	if (!d_ephandle->start ())
	{
		fprintf (stderr, "usrp_basic_rx: failed to start end point streaming");
		usb_strerror ();
		printf("usrp_basic_rx::start - failed to start end point streaming\n");
		return false;
	}
	else
	{
		printf("usrp_basic_rx::start - no errors encountered during attempt to start
	endpoint streaming\n");
	}

	printf("usrp_basic_rx::start - attempting to enable rx\n" );

	if (!set_rx_enable (true))
	{
		fprintf (stderr, "usrp_basic_rx: set_rx_enable failed\n");
		usb_strerror ();
		printf("usrp_basic_rx::start - call to set_rx_enable returned false\n");
		return false;
	}
	else
	{
		printf("usrp_basic_rx::start - call to set_rx_enable returned true!\n");
	}
	return true;
}

//added by kw
int usrp_basic_rx::reset_sdr_module_fpga(bool on)
{
	return usrp_set_fpga_reset (d_udh, on);
}

usrp_basic_rx *usrp_basic_rx::make (int which_board, bool bCheckForFX2)
{
	//kw - diagnostic
	printf("Entering usrp_basic_rx::make\n");

	usrp_basic_rx *u = 0;
	try
	{
		//kw - diagnostic
		printf("usrp_basic_rx::make - attempting to create usrp_basic_rx object\n");
		u = new usrp_basic_rx (which_board);
		if (!u->initialize ())
		{
			fprintf (stderr, "usrp_basic_rx::make failed to initialize\n");
			throw std::runtime_error ("usrp_basic_rx::make");
		}
		return u;
	}
	catch (...)
	{
		delete u;
		return 0;
	}
	return u;
}

/*
 * \brief read data from the D/A's via the FPGA.
 * \p len must be a multiple of 512 bytes.
 *
 * \returns the number of bytes read, or -1 on error.
 *
 * If overrun is non-NULL it will be set true iff an RX overrun is detected.
 */
int usrp_basic_rx::read (void *buf, int len, bool *overrun)
{
	int	r;

	if (overrun)
		*overrun = false;

	if (len < 0 || (len % 512) != 0)
	{
		fprintf (stderr, "usrp_basic_rx::read: invalid length = %d\n", len);
		return -1;
	}

	r = d_ephandle->read (buf, len);
	if (r > 0)
		d_bytes_seen += r;

	/*
	* In many cases, the FPGA reports an rx overrun right after we
	* enable the Rx path.  If this is our first read, check for the
	* overrun to clear the condition, then ignore the result.
	*/

	if (0 && d_first_read)
	{	// FIXME
```

```
                d_first_read = false;
                bool bogus_overrun;
                usrp_check_rx_overrun (d_udh, &bogus_overrun);
        }

        if (overrun != 0 && d_bytes_seen >= d_bytes_per_poll)
        {
                d_bytes_seen = 0;
                if (!usrp_check_rx_overrun (d_udh, overrun))
                {
                        fprintf (stderr, "usrp_basic_rx: usrp_check_rx_overrun failed\n");
                        usb_strerror ();
                }
        }
        return r;
}

bool usrp_basic_rx::set_rx_enable (bool on)
{
        printf("usrp_basic_rx::set_rx_enable - calling usrp_set_fpga_rx_enable\n");
        d_rx_enable = on;
        return usrp_set_fpga_rx_enable (d_udh, on);
}
```

Test application (Test_usrp_standard_rx.cc)

This application accepts a number of command line parameters, and carries out various tests on a configured FX2-based board. The code was originally written by Larry Doolittle, but has been modified by the author to handle this project's hardware, and to output received data in a comma-separated file format.

```
/* -*- c++ -*- */
/*
 * Copyright 2003 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

#ifdef HAVE_CONFIG_H
       #include "config.h"
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <usb.h>                     /* needed for usb functions */
#include <getopt.h>
#include <assert.h>
#include <math.h>
#include "time_stuff.h"
/* #include "usrp_standard.h" */
#include "usrp_basic.h"
#include "usrp_bytesex.h"

char *prog_name;

static bool test_input  (usrp_basic_rx *urx, int max_bytes, FILE *fp);

static void set_progname (char *path)
{
       char *p = strrchr (path, '/');
       if (p != 0)
              prog_name = p+1;
       else
              prog_name = path;
}

static void usage ()
{
       fprintf (stderr, "usage: %s [-f] [-v] [-l] [-c] [-D <decim>] [-F freq] [-o
       output_file]\n", prog_name);
       fprintf (stderr, "  [-f] loop forever\n");
       fprintf (stderr, "  [-M] how many Megabytes to transfer (default 128)\n");
       fprintf (stderr, "  [-v] verbose\n");
       fprintf (stderr, "  [-l] digital loopback in FPGA\n");
       fprintf (stderr, "  [-c] counting in FPGA\n");
       exit (1);
}

static void die (const char *msg)
{
       fprintf (stderr, "die: %s: %s\n", prog_name, msg);
	exit (1);
}

int main (int argc, char **argv)
{
	bool	verbose_p = false;
	bool	loopback_p = false;
	bool  counting_p = false;
	int   max_bytes = 128 * (1L << 20);
	int	ch;
	char	*output_filename = 0;
	int	which_board = 0;
	int	decim = 8;			// 32 MB/sec
	double center_freq = 0;

	set_progname (argv[0]);

	while ((ch = getopt (argc, argv, "fvlco:D:F:M:")) != EOF)
	{
		switch (ch)
		{
			case 'f':
				max_bytes = 0;
			break;

			case 'v':
				verbose_p = true;
			break;

			case 'l':
				loopback_p = true;
			break;

			case 'c':
				counting_p = true;
			break;

			case 'o':
				output_filename = optarg;
			break;

			case 'D':
				decim = strtol (optarg, 0, 0);
			break;

			case 'F':
				center_freq = strtod (optarg, 0);
			break;

			case 'M':
				max_bytes = strtol (optarg, 0, 0) * (1L << 20);
				if (max_bytes < 0) max_bytes = 0;
			break;

			default:
				usage ();
		}
	}

	FILE *fp = 0;

	if (output_filename)
	{
		fp = fopen (output_filename, "wb");
		if (fp == 0)
			perror (output_filename);
	}

#if 0
	int mode = 0;
	if (loopback_p)
		mode |= usrp_standard_rx::FPGA_MODE_LOOPBACK;
	if (counting_p)
		mode |= usrp_standard_rx::FPGA_MODE_COUNTING;
#endif

//kw
```

```
        printf("Attempting to create basic USRP RX object...\n");

        usrp_basic_rx *urx = usrp_basic_rx::make (which_board);
        if (urx == 0)
                die ("usrp_basic_rx::make");
//kw
        printf("No errors encountered! (yet)\n");

#if 0
        if (!urx->set_rx_freq (0, center_freq))
                die ("urx->set_rx_freq");
#endif

//kw
        printf("Calling urx->start()...");
        urx->start();         // start data xfers
//kw
        printf("urx->start() done!\n");
//kw
        printf("Attempting to reset FPGA before any transfers take place\n");
        urx->reset_sdr_module_fpga (1);
        printf("Attempting to deassert reset FPGA before any transfers take place\n");
        urx->reset_sdr_module_fpga (0);

//kw
        printf("Attempting to call urx->test_input...\n");
        test_input (urx, max_bytes, fp);
//kw
        printf("done urx->test_input!\n");
        if (fp)
                fclose (fp);
        delete urx;
        return 0;
}

static bool test_input  (usrp_basic_rx *urx, int max_bytes, FILE *fp)
{
        int               fd = -1;
        static const int BUFSIZE = urx->block_size();
        static const int N = BUFSIZE/sizeof (short);
        short   buf[N];
        int     nbytes = 0;

        double    start_wall_time = get_elapsed_time ();
        double    start_cpu_time  = get_cpu_usage ();

        if (fp)
                fd = fileno (fp);

        bool overrun;
        int noverruns = 0;

        for (nbytes = 0; max_bytes == 0 || nbytes < max_bytes; nbytes += BUFSIZE)
        {
                unsigned int   ret = urx->read (buf, sizeof (buf), &overrun);
                if (ret != sizeof (buf))
                {
                        fprintf (stderr, "test_input: error, ret = %d\n", ret);
                }
                if (overrun)
                {
                        printf ("rx_overrun\n");
                        noverruns++;
                }
                if (fd != -1)
                {
                        for (unsigned int i = 0; i < sizeof (buf) / sizeof (short); i++)
                        {
                                buf[i] = usrp_to_host_short (buf[i]);
                                fprintf(fp, "Short %d of %d: \t[0x%X]\n", i, (sizeof(buf) /
                        sizeof(short)),(unsigned)(buf[i] & 0xFFFF));
                        }

                        //if (write (fd, buf, sizeof (buf)) == -1)
                        //{
                        //      perror ("write");
                        //      fd = -1;
                //}
            }
            else
            {
                //printf("File error...\n");
            }
        }

        double stop_wall_time = get_elapsed_time ();
        double stop_cpu_time  = get_cpu_usage ();

        double delta_wall = stop_wall_time - start_wall_time;
        double delta_cpu  = stop_cpu_time  - start_cpu_time;

        printf ("xfered %.3g bytes in %.3g seconds.  %.4g bytes/sec.  cpu time = %.4g\n",
        (double) max_bytes, delta_wall, max_bytes / delta_wall, delta_cpu);

        printf ("noverruns = %d\n", noverruns);
        return true;
}
```

Appendix D - Installing GnuRadio

This appendix provides a description of the installation steps required to install *GnuRadio* on an Ubuntu Linux system.

Firstly, use apt-get to install the core *GnuRadio* source (*gnuradio-core*). For some reason the *Ubuntu 6.06* standard repositories do not have all the requisite packages to run as is, and various packages have to be obtained after apt-get has installed the base *GnuRadio* package (at the time of writing).

The full list of *GnuRadio* packages is listed below. Any that are not available through apt must be obtained via CVS from the *GnuRadio* website (see www.gnu.org/software/gnuradio).

Required GnuRadio packages:

• gnuradio-core	- Code base for GnuRadio
• gr-usrp	- USRP modules
• gr-audio-alsa-0.5.orig	- ALSA sound card interface
• gr-wxgui	- Interface with WX GUI
• gr-audio-oss	- Open-Sound System sound card interface
• usrp-0.12	- USRP modules
• gr-howto-write-a-block	- Examples and tutorials

In addition to the above, various compilation and cross-language 'glue' packages are required. These are listed below:

- **SWIG**
  Creates wrappers around C++ functions to make them easily accessible to Python.

- **Python** (including python-dev and libraries, numpy and numarray)
  The top-level applications are written in Python.

- **GCC, G++** (version 3.2, 3.4 or higher. Version 3.3 conflicts with the *GnuRadio* source [5])
  Required to build the source code.

- **Make, Autoconf, Libtools**
  Required to build the source code.

- **FFTW**
  Provides Fast Fourier Transform functionality.

- **SDCC**
  Assembles the FX2 firmware into native 8051 hex format.

- **FXLOAD**
  Not part of the *GnuRadio* package, but used in this project to program the FX2 device directly from the command line.

- **USBVIEW**
  Also not part of the *GnuRadio* package, but this is a useful tool for debugging USB devices.

There are several helpful tutorials on installing *GnuRadio* for various Linux distributions, which can be found through the *GnuRadio* website (www.gnu.org/software/gnuradio).

Bibliography

[1] SDR forum website, http://www.sdrforum.org

[2] J. Mitola, "*Software Radio Architecture Evolution: Foundations, Technology Tradeoffs, and Architecture Implicarions*", IEICE Trans. Commun. Vol. E83-B, June 2000

[3] Gnuradio website, http://www.gnu.org/software/gnuradio

[4] Ettus Research website, www.ettus.com

[5] GnuRadio wiki, http://gnuradio.org/trac/wiki

[6] CY7C68013 EZ-USB FX2 data sheet, Cypress Semiconductor, December 2002

[7] Spartan-3 FPGA data sheet, Xilinx Incorporated, December 2003

[8] AD9238 data sheet, Analog Devices, 2003

[9] Cyclone II FPGA data sheet, Altera Corporation, 2004

[10] Low Voltage Differential Signaling (LVDS) design guide, National Semiconductor, Spring 2000

[11] High Speed USB PCB layout recommendations (application note), Cypress Semiconductor, December 2002

[12] USB 2.0 Specification, USB Implementers Forum, April 2000

[13] PC/104 Specification, Version 2.5, PC/104 Embedded Consortium, November 2005