Design of a High Speed Peripheral Interface

by

Shane Hosking

Student Number: 33706618

Supervisor: Dr Gordon Wyeth

Undergraduate Thesis

Department of Computer Science and Electrical Engineering

Shane Hosking
7 Jenalyn Crescent,
Bundaberg, Qld, 4670
19, October, 1999

Professor Simon Kaplan
Head of School
School of Information Technology and Electrical Engineering,
University of Queensland
St Lucia QLD 4072.

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Electrical Engineering, I present the following thesis entitled " Design of a High-speed peripheral interface". This work was performed under the supervision of Gordon Wyeth.

I declare that the work submitted in this thesis has not been previously submitted for adegree at the university of Queensland or any other institution. To the best of my knowledge and belief, this thesis contains no material previously published or written by any other person, except where reference is made in the text.

Yours sincerely

Shane Hosking

Abstract

This thesis analyses the development of a high-speed peripheral interface between an IPAQ pocket PC and a Super H 4 processor to be used in the vision system of a humanoid. Previously this type of communication has been performed with typically low bandwidth communication techniques such as an RS232 serial link. This communication system utilises a standard PCMCIA interface to transport video, processed and calibration data simultaneously. This thesis will analyse the protocol used the design of the hardware and the software developed for this peripheral interfaces.

This design uses the IPAQ as the master and SH4 as a slave. The hardware design is implemented by using both a PCB and FPGA combination. Software drivers were also written to control the IPAQ and SH4. This thesis covers the design of this the hardware and software in great detail.

Several major achievements were made in the development of this communication system including the successful development of a driver for the SH4 and development of working hardware. Results obtained have been included.

Possible future work can be done on the development of a successful IPAQ driver. This thesis makes several suggestions on how this system could be fully implemented given more time.

Acknowledgements

Many people have contributed to the development of this thesis I would like to thank all of them.

Firstly I'd like to thank my parents who have continually given me the strength to when my motivation was waning and for their proof reading skills in the later parts of my Thesis.

My supervisor Gordon Wyeth who has constantly provided his time and knowledge to help me with my design.

Mark Chang for his tolerance of the constant bombadment of stupid questions that I have sent in his direction.

Wesley Hosking for his help in the development of the IPAQ driver.

Finally all the guys (and girl) in the Robotics Lab for the many distractions that they have provided, which have kept me sane throughout this thesis

List of Figures and Tables

1.0 Introduction

This thesis analyses the design and development of a high-speed peripheral interface designed to act as the optical nerve in the vision system of a humanoid. It will examine in detail how a 32-bit data bus using the PCMCIA protocol will be used to connect the main processor of the humanoid to the vision processor. It will then show how a DMA transfer will be used to transmit the data between the vision processor and the main processor. This thesis will analyse the development of the hardware, software drivers and protocol used to implement this communication system.

1.1 RoboCup

RoboCup is an international competition setup to encourage the development of robots in a competitive atmosphere. This year will see the start of a new experimental competition the humanoid league. This draft regulation for league offers various competitions to test the various challenges in the development of the humanoid. These challenges include balancing on one leg, a penalty shootout and an N on N robot soccer contest. The humanoid league moves the competition closer to its overall goal of developing a robotic soccer player capable of competing with even defeating a human soccer player by the year 2050.

1.1 GUROO

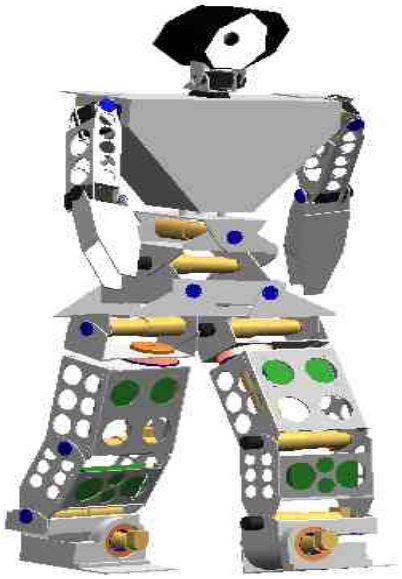

Figure 1.1
The GuROO

The GURoo (Grossly Under funded Roo) as it has become known, is the humanoid being developed to represent the University of Queensland in the humanoid league at RoboCup 2002.

This project combines the talent of twelve undergraduate students in areas such as mechanical design, vision system, control and power and is under the leadership of Dr Gordon Wyeth. The ultimate goal of this project is to design a humanoid that has the ability to find a ball on a given field, walk towards the ball and then kick it. A basic block diagram showing how the different parts of this project relate to each other is shown in figure 1.1. The vision system is an important part of the total system as it allows the humanoid to respond to visible stimulus.

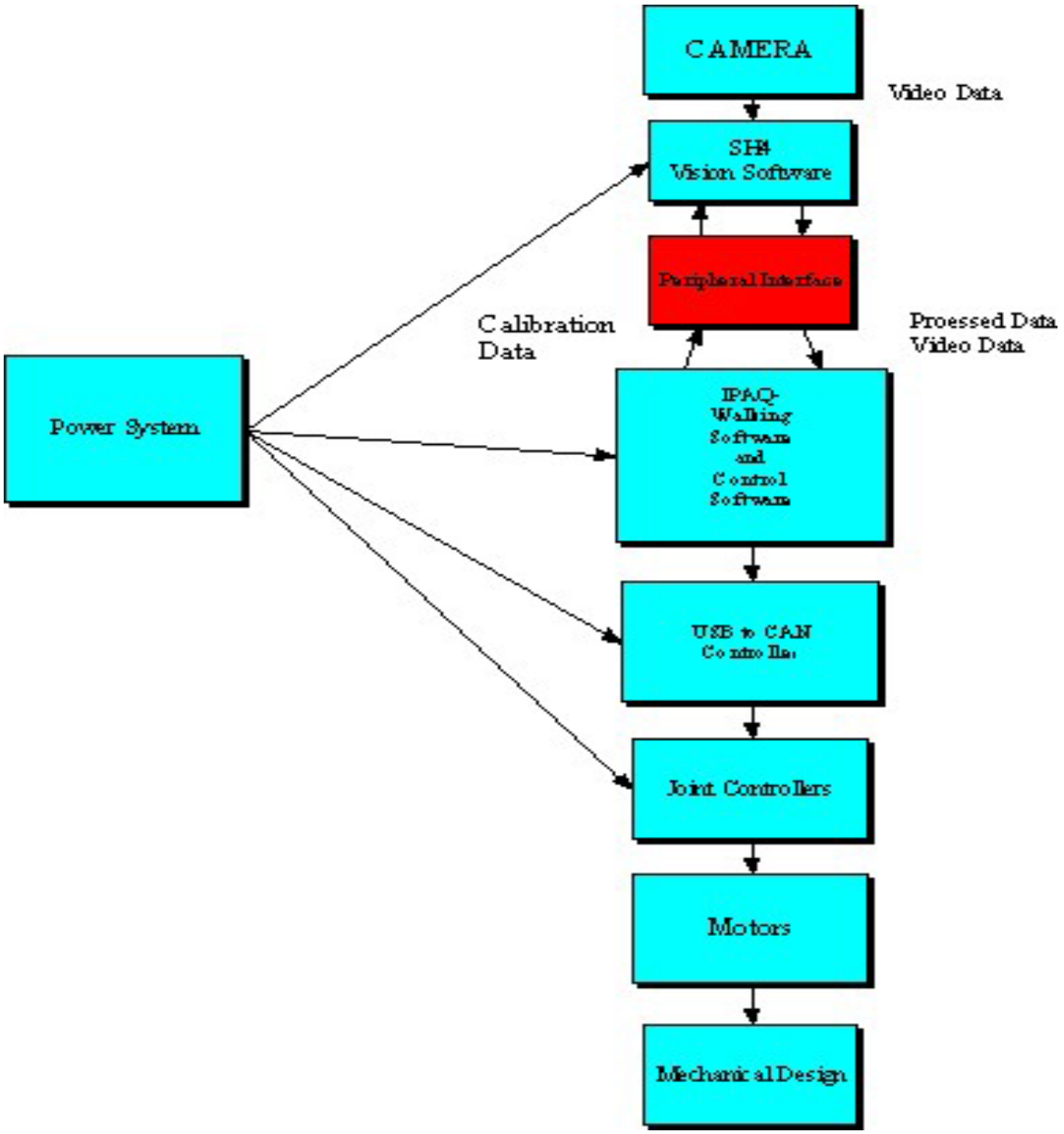

Figure1.2
Project break up of GuROO Project

1.2 The Problem

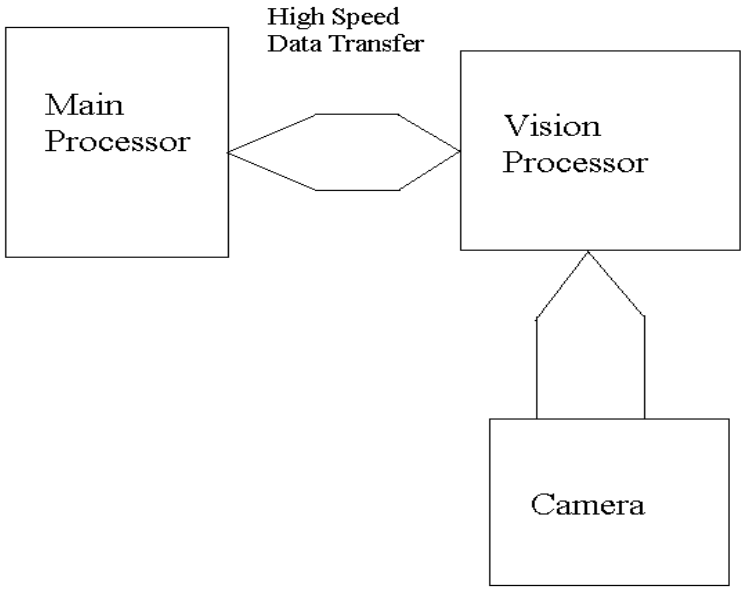

Figure 1.3
The Vision System

The main goal of this thesis is to develop a high-speed communication with low overhead. This communication system is to be used as the optical nerve in the vision system of a humanoid. This communication system is highlighted in red in the block diagram of the humanoid.

The operation of a Humanoid can put a large amount of strain on the main processor without adding the large amount of processing required to operate a high-speed bus. Therefore a communication system with a large bandwidth, low latency but with minimal overhead is needed, as is demonstrated in the block diagram of the system in figure 1.3

The communication channel in this particular humanoid needs to be able to communicate in either direction, with the primary concern being placed on the vision information traveling from the vision processor to the main processor. This information consists of real time video and processed information. The transfer rate in this direction needs to be as fast as possible. Alternatively the data to be transferred from main

processor to the vision processor is part of the vision processor calibration process. This type of transfer will occur at the start of each operating session. It is much less frequent and the information can therefore be passed at a relatively slower rate if necessary.
The processing unit selected, as the main processor is an IPAQ pocket PC. The vision processor selected is a SH4. The two processors are to be connected through a Xilinx Spartan II field programmable gate array to add extra flexibility to the system.

1.3 Justification of Research

This project has been undertaken for two major reasons.

The primary reason is the decision to separate the image processing from the main processor. This system is effective as the workload is spread, but it is important not to set up a bottleneck in the information passing between them. Therefore it is desirable to develop a fast form of communication with a large bandwidth and low over-head to combine their processing power as effectively as possible.

Another reason for completing this research is the fact that the IPAQ is a relatively new device. This means that little work has been done on developing useful expansion packs particularly using the extended 32-bit PCMCIA interface.

1.4 The Achievements

This project has achieved several key results.

- The Hardware has been developed to interface the IPAQ to the SH4. This hardware implements a I/O specific PCMCIA socket
- The FPGA code has been written to interface the PCB to the SH4.
- A software driver for the IPAQ has been developed and is nearly operational.

- A driver for the SH4 has also been written but the operation has not been fully verified.

1.5 Layout

Chapter 2 will examine the current ideas and standards used for high speed peripherals. It will also cover the relevant theory. In particular it will examine the PCMCIA protocol in depth, as it is most relevant to this thesis. The features of the main processor and vision processor will also be discussed.

Chapter 3 will explore the specifications of this communication system in greater depth. It will then illustrate the specifications required by the software and hardware in detail.

Chapter 4 will examine the hardware design implemented in this peripheral interface. In particular it will examine the development of the printed circuit board and FPGA and will illustrate the functionality of this circuit.

Chapter 5 will examine the design of the software drivers used to operate the hardware developed. It will break the software in two sections the IPAQ driver and the SH4 driver. It will then analyse the functions implemented by these drivers in great detail.

Chapter 6 will reveal the overall performance of this system. It examines the results that were achieved in each section and provide reasoning for each result.

Chapter 7 will conclude the findings of this design and will suggest any improvements or work to be done in the future.

2.0 Review of Literature

To understand the ideas developed within this design it is important to have some background knowledge in three key areas. Firstly it is necessary to assess previous ideas in the field of data transfer within robotics. Secondarily it is important to have an understanding of the processors and interfaces used within this transfer. Finally it is necessary to understand the operating systems that will control the processors.

2.1 Previous Ideas

Based upon the research done, the task of transferring large amounts of data within the field of robots can be divided into typically two groups based upon the speed of data transfer necessary this varies in relationship to the robots application.

Typically when a robot needs to have a compact design and minimal hardware data communication is carried out by typically low bandwidth, high latency solutions.

Keane(1999)[12] presents a communication system to communicate between a PC and six mobile robots playing robot soccer. Due to the small size of the robots a compact design is important. In this method a Rs232 serial transmission is also used but data is transferred using a RF transmitter and receiver. This system takes advantage of the greater processing power of the PC computer but maintains the mobility of the robots.

Sung-Hyan (1999) [1] performed a study on Real-Time implementation of a visual feedback for control of a Robotic Manipulator. This system uses a binocular stereovision system to control a four axis Scara Robot (SM5 model). To transport the vision information used to control the main processor a low bandwidth Rs232 connection is used. The final results show that the system was successful but the speed of the transfer had a large effect on the performance of the control system.

Vision systems that require close to real time speed and have to transmit large amounts of data tend to use systems that have higher overhead and more hardware but provide greater transfer speed and bandwidth.

Scassellati(1999)[13] analyses a high speed communication system used to implement a Binocular Activision vision system in the development of the robot COG. This system is required to have high-speed recognition of gestures. The communication system uses a high-speed bi-directional hardware link called comports to communicate between a network of DSP processors. This system transports information at approximately 40Mbits/second. This system then interfaces to a PC through an ISA bus.

[11] illustrates the development of a high performance camera platform for real time active vision. This system transfers video data from a Panasonic GP-KS1000 camera at a frame rate of 30 Hz. To implement this system a high speed Max bus has been used to transfer information from the camera to the main processor.

Our system requires a high bandwidth system as demonstrated in [13] and [11], our system will have to conform to the constraints placed upon it by the chosen processors. Understanding the design principles behind these vision systems can offer insight in to the necessary feature of an embedded transfer system.

2.2 Peripheral Interfaces

There have been many forces involved in shaping the growth of peripheral interfaces. The main source of this growth is the rapid development of mobile computers and their peripherals. This has fueled the development of high-speed peripheral interfaces. Although their uses vary with every specific design, Schmidt (1998) [4] points out that the abstract model of an interface can be made up of four layers. The lowest of the layers is the actual physical interface. This includes the type of material used for construction, the voltages and currents required and the timing of signals. The layer above this is known as the protocol layer. This layer defines the format of bytes and includes error checking if necessary. The next layer describes the behavior of the peripheral device and states how the interface should be operated to account for this behavior. Finally the final layer states any commands that are necessary to control the interface. A good example of this layer is the standardized commands used by printers. The interface presented in this thesis can be broken into all of these layers.

2.3 IPAQ

Figure 2.1
The IPAQ H3630 [20]

The IPAQ H3630 pocket PC is a handheld computer developed by Compaq Presario. It features a thirty-two-bit STRONG-ARM SA1110 RISC processor. The STRONG-ARM SA1110 is a powerful processor designed specifically for low power situations. It is based upon the SA-1 core and features large on-chip caches. There are two major interfaces that support a 32-bit transfer; the SRAM like variable I/O interface and the PCMCIA interface. DMA transfer is only supported by the variable I/O interface, the PCMCIA interface does not support DMA transfer.

The IPAQ has two major ports for expansion, a USB port and a 100-pin connector. These allow the additions of an expansion pack and allow access to the different modules of the chip. The dimensions of the recommended expansion pack are given in [###]. Windows CE, which is produced by Microsoft, was the chosen operating system. This choice of operating system will influence the way that the hardware can be controlled through software.

2.4 PCMCIA

The PCMCIA interface was selected for the IPAQ's interface for reasons that will be made clear in the design chapter. Some background knowledge of PCMCIA is necessary to understand this design.

The development of the mobile computer industry has fueled the need for smaller, more efficient and most importantly portable peripheral devices. This has led to the need to develop standardised peripheral interfaces to cater for the mobile community. As stated in Anderson (2000) [5] the Personal Computer Memory Card International Association was formed in 1989, it's goal was to promote the standardisation and interchangeability of PC Cards. In 1991 the first PCMCIA standard (version 1.0) was released. This first release was designed specifically as an interface for memory cards for portable computers. The main advantage of the PCMCIA interface is the fact that it is a standardized interface. This means that it is highly portable between devices and that a large amount of information on design is available.
The latest version of PCMCIA has expanded the standard to incorporate I/O transfers. The standard defines four major characteristics

- Physical design of PC Card
- Physical design of connector
- Electrical design of connector
- Electrical interface
- Software Architecture
- This chapter will focus in particular on the PCMCIA electrical interface for more information on the full PCMCIA interface consult [5].

2.41 The Electrical Interface

The Regular PCMCIA interface defines a 16-bit data-path with a 26- bit address bus. This interface actually includes three different types of interface; a Common memory interface, an Attribute memory interface and a data I/O interface. These three interfaces are designed for three different types of data transfers. The Common memory interface is used for accesses to external storage memory. The Attributes memory interface is used to access configuration information about the card and also configuration registers. This type of access is designed to determine and control the

properties of the PCMCIA card. Finally the I/O interface is used to transfer standard I/O data from peripherals. Each of these interfaces has 64 Mbytes of address space dedicated to it.

The Electrical signals defined by the PCMCIA can also be broken down into three different areas; the General signals, the Memory signal and the I/O signals. The general control signals defined by the PCMCIA standard are used for initialisation of the PCMCIA card and handshaking during the transfer. The memory interface signals are used for transfers to and from memory. In particular the /PREG signal is used to differentiate between the two memory interfaces. When /PREG is asserted the transfer will access the Attributes memory, while common memory is accessed when /PREG is not asserted. Finally the I/O signals are designed specifically for I/O transfers. A more detailed description of each signal is included in table [2.1].

General PCMCIA Signals

Signal	Description
/CD1, /CD2	Card Detect. These signals are asserted when the PC Card has been installed. They are used to notify the system the PC Card has been installed.
/CE1	When /CE1 is active low it specifies that the data will be transferred using (D7:D0).
/CE2	When /CE2 is active low it specifies that the data will be transferred using (D8:D16).
/WAIT	This signal is asserted to insert wait states into the transfer.

Memory Signals

/OE	Output Enable. This signal is asserted during a read from memory.
/WE	Write Enable. This signal is asserted during a write from memory.
/PREG	This signal is asserted to indicate an access to attribute

	memory.

I/O Signals

/IORD	I/O Read. This signal is asserted during I/O read transfers from PCMCIA.
/IOWR	I/O Write. This signal is asserted during I/O write transfer to the expansion pack.
IOIS16	IO is 16-bits. This signal is asserted during I/O read transfers from PC Cards, if device size is 16 bits.
/IRQ	Interrupt Request. This signal is asserted to indicate the expansion pack has an interrupt request

Table 2.1
PCMCIA Control Signals
And explanations.

A PCMCIA transfer is a synchronous transfer, the timing of each si2.1nal relative to the clock is shown in figure 2.1 obtained from the SA1110 data sheet [4]. To perform correct operation these signals and the data signals need to be correctly buffered. Some glue logic is required to correctly operate the buffers and save power.

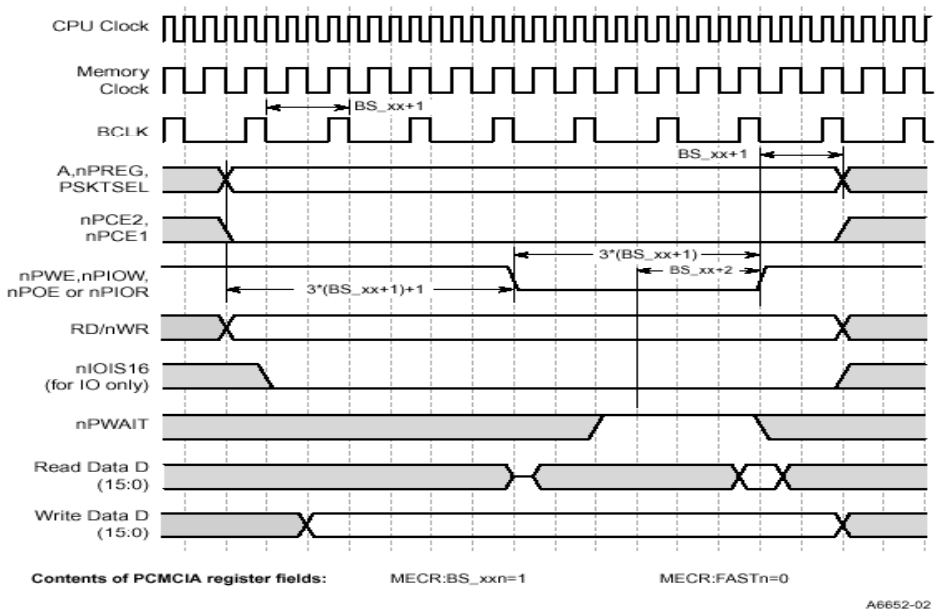

Figure 2.2
PCMCIA Timing diagram [4]

2.42 PCMCIA - IPAQ

The IPAQ provides support for two PCMCIA sockets. It does not however support the PCMCIA DMA transfer. The PCMCIA interface can be divided up into eight partitions based upon the three different types of interface; each of these interfaces is 64 MB in size. A diagrammatic representation of the PCMCIA interface is shown in figure [2.2] from the SA1110 data sheet.

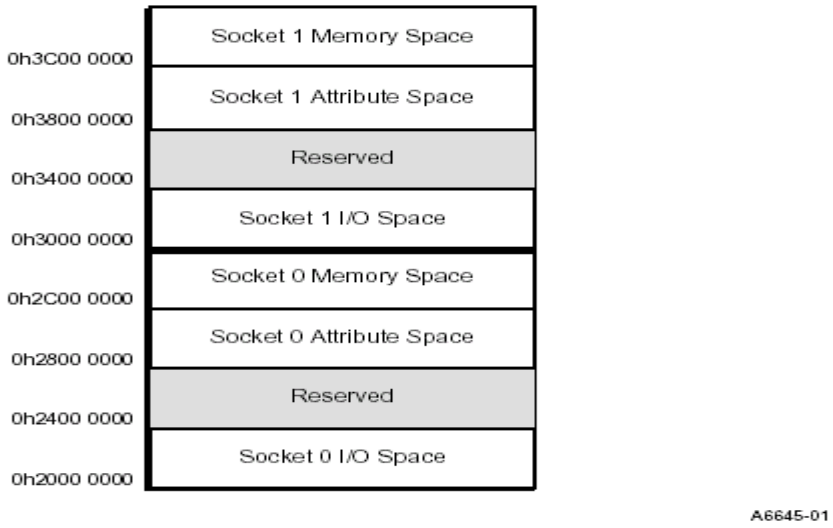

Figure 2.3
PCMCIA Address space [4]

Reading or writing data to this memory space can initiate a PCMCIA transfer. Using the PCMCIA timing register (MECR) can then control the timing of the transfer. A detailed description of this register is contained in figure [2.3], which was also obtained from the SA1110 data sheet [4].

0h A000 0018 MECR Read/Write

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FAST1	BSM1_4	BSM1_3	BSM1_2	BSM1_1	BSM1_0	BSA1_4	BSA1_3	BSA1_2	BSA1_1	BSA1_0	BSIO1_4	BSIO1_3	BSIO1_2	BSIO1_1	BSIO1_0	FAST0	BSM0_4	BSM0_3	BSM0_2	BSM0_1	BSM0_0	BSA0_4	BSA0_3	BSA0_2	BSA0_1	BSA0_0	BSIO0_4	BSIO0_3	BSIO0_2	BSIO0_1	BSIO0_0
et ?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

(Sheet 1 of 2)

Bits	Name	Description
4..0	BSIO0 4..0	Memory clock count for accesses to PCMCIA card slot 0, I/O space.
9..5	BSA0 4..0	Memory clock count for accesses to PCMCIA card slot 0, attribute space.
14..10	BSM0 4..0	Memory clock count for accesses to PCMCIA card slot 0, common memory space.
15	FAST0	Fast mode bit for access to slot 0 I/O, attribute, or memory. If FAST0=1, the set-up time from address generated signals (A, nPREG, PSKTSEL and nPCE) to initial assertion of the read or write strobe (nPWE, nPIOW, nPOE, or nPIOR) is 1*(BS_xx + 1) + 1 instead of the normal 3*(BS_xx + 1) + 1. During I/O accesses, the nPCE set-up time is always reduced from these values by any A-to-nIOIS16 delay. The set-up time from address generated signals to the assertion of the read or write strobe for the second half of a 16-bit access to 8-bit I/O is 2*(BS_xx + 1) instead of 1*(BS_xx + 1). The duration of the read or write strobe remains 3*(BS_xx + 1), regardless of the value of FAST0.

Figure 2.4
PCMCIA Timing register
[4]

Storing a value in this register will control the value of the delay BS_xx in figure[2.1]. The glue logic necessary, to implement a PCMCIA socket on the IPAQ is shown in figure [2.4]. This circuit buffers the signals to remove the effect of stray capacitances and improve the efficiency of the design.

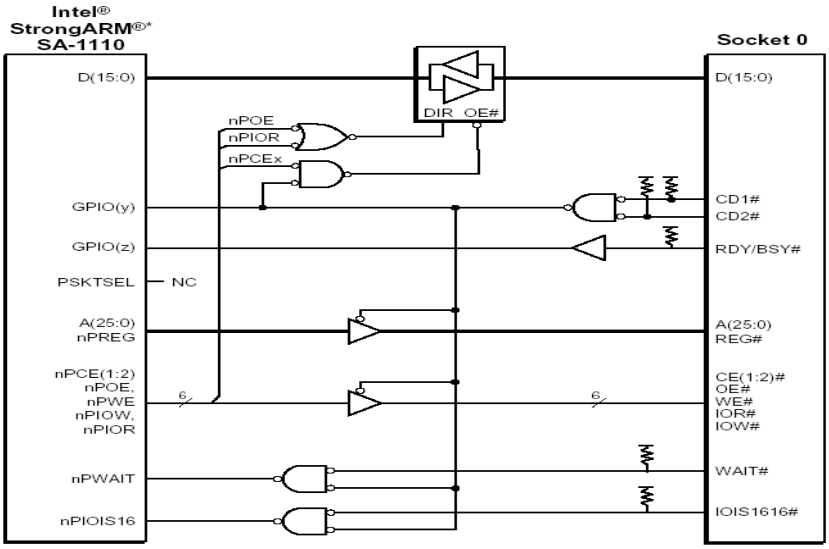

Figure 2.5
PCMCIA socket [4]

The IPAQ allows the extension of the standard 16-bit PCMCIA interface to a 32-bit interface; use of this type of interface will require the development of a driver to control this interface. It must also be noted that using a 32-bit transfer will mean that A0:1 of the address bus must be zero.

2.5 Windows CE

As with most operating systems the hardware in a system using Windows CE is accessed using device drivers. The device drivers in windows CE are implemented by using dynamic link libraries instead of the WDM files used by Windows NT.
When developing a software driver for a windows CE application there are several options available. As is shown in Microsoft's [16] driver development kit there are two basic driver models which are accepted in windows CE 3.0; stream drivers and native drivers. Stream drivers expose a standard interface, this means that the driver must contain a set of standard functions with set input and return parameters. For more information on the stream driver interface consult Platform builders help [10].
Alternatively native device expose a custom made interface to the API level. These Device drivers can be implemented in a single monolithic driver that will communicate directly from the device to the API level or in several layers of device drivers.
Layered Native drivers generally use two layers; a platform dependent layer and a model dependent layer. The platform dependent layer is responsible for managing the hardware while the model dependent layer is responsible for controlling the behavior of the device. A monolithic device uses one driver to control both of these layers. The layered model offers the benefits of being highly modular and therefore easily ported between devices. The monolithic driver offers the advantage of speed, it is also the easiest to implement in a one off situation.

To install a driver into a Windows CE system the compiled DLL file must first be stored in the /Windows directory of the device in question. The device may then be accessed in several ways. If the driver only needs to be accessed by one API the driver can be loaded using the LoadDriver function as detailed in Platform Builder. This

function will load the driver into a memory location and return a handle to the driver instance. The interface of the driver can then be accessed by the API. No other API may access a driver initialized in this way. Alternatively the Register device function or Activate device function can be used to access the driver. These functions will register the driver in the registry, the device may then be accessed by multiple APIs. This type of accessed may not always be possible and securities need to be put in place to ensure that no conflicts occur.

All the information contained in this chapter was attained from Microsoft's Platform Builder [10]. If any further work is to be done on the development of an IPAQ driver using Windows CE it is highly recommended that the Platform builder technical documents be consulted.

2.5 Super H

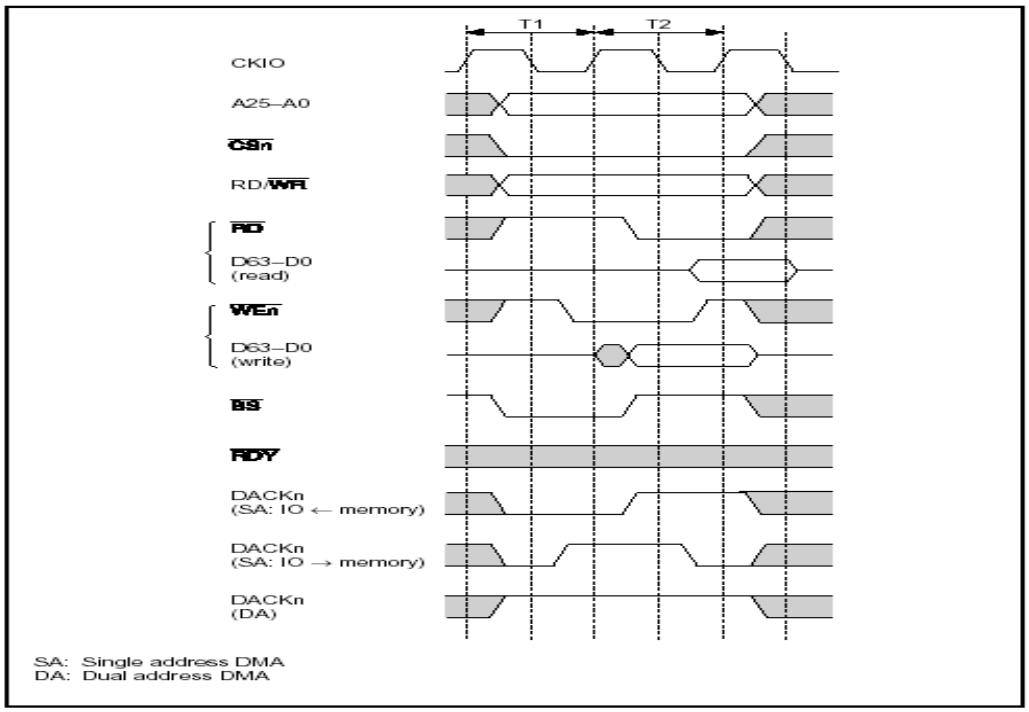

Figure 2.6
Bus Timing Diagram []

Signals	Description
/RD	Read. This signal is asserted when a read operation is initiated.
/WE	Write Enable. This signal is asserted when a write operation is initiated

Table 2.2
Bus Signals and their
definitions

The SH7750, which is the chosen vision processor, is a 32-bit RISC super scalar processor developed by Hitachi Ltd. Based upon the Super H hardware manual [7] it contains twenty-six bit address bus and a possible sixty-four bit data bus. It also supports two major I/O interfaces PCMCIA and SRAM like I/O. The PCMCIA interface is a standard PCMCIA interface and cannot be extended to 32-bits. The SRAM interface is capable of transmission using bus sizes up to 64-bits. The control signals for the SRAM interface and there functions are shown in table [2.2]. Both of these interfaces support direct memory access transfer. The timing of these signals is shown in figure [2.5]
Mark Chang a postgraduate at the University of Queensland has developed the SH4 board. It has been developed as the main processor controlling a group of mobile Robots known as the ViperRoos. He has adapted his design act as the vision processor for the humanoid project. David Prasser an undergraduate student at the University of Queensland has developed the vision software.

2.6 Direct Memory Access

Direct memory access is a function provided by computer bus architectures to allow data to be sent from a peripheral device to memory without involving the main processor. This will remove the latency of a large data transfer from the main processor and will speed up the whole process. The DMA controller controls the operations of the DMA transfers. It will periodically seize the bus from the main processor and transfer a block of data.

The SH4 features four DMA channels available for DMA transfers; only two of these channels can be accessed externally through the DREQ pin. The DMA controller is capable of transfers of 16-bit, 32-bit, 64-bit and 32 bytes.

Two modes of operation are available; single address mode and dual address mode. Single address mode is used when both transfers source and destination are external locations. Dual address mode is used when both source and destination are accessed using an address. This source and destination are then controlled using the DMA source register and DMA destination register.

Two bus modes are also available Cycle steal and Burst mode. The cycle steal mode releases the bus to the main processor at the end of each transfer. The Burst mode will release the bus as soon as the DREQ signal is pulled high regardless of whether the transfer is complete.

3.0 Specifications

This Chapter will outline the specifications of this project in greater detail. It will first give the general specifications for the project it will then analyse the specifications for the hardware and software in more detail.

3.1 General Specifications

As was briefly covered in the first chapter the ultimate goal of this design is the development of a communication system capable of transmitting and receiving large amounts of data between an IPAQ pocket PC and Super H 4 Processor. This system is to be implemented as part of the vision system of a humanoid. Video information will be collected from a OV7620 CMOS Omnivision camera developed by Andrew Blower [15]. This video information will then be passed from the camera to the SH4 vision processing board. Image processing software developed by David Prasser [14] will then segment this image data and gather information on the surrounding environment. The video data and processed data will then be passed to the main processor through the high-speed peripheral interface developed in this thesis.

The data to be transported by this system can then be broken down into three forms; video data, processed data and calibration data. The video data is simply RGB data from the camera. The processed data is data containing vision information obtained from the image processing software. Finally the calibration data is data used to adjust the image processing software to new environments

This communication channel must be able to transfer large amounts of video data as well as a smaller amount of processed information from the SH4 to the IPAQ on a regular basis. It must also be able to transfer a smaller amount of calibration data from the IPAQ to the SH4 on a less frequent basis. This information is to be used in a real time video and control system for the humanoid, it is therefore important that the transfer be as fast and as reliable as possible.

There are three major characteristics that are required from this communications system speed, accuracy, and compatibility. From a design point of view the most important of these characteristics is compatibility. The chosen method of data transmission must be compatible with both the chosen interface for the main processor and the interface selected for the vision processor. The next major criterion, which is as important as the first, is accuracy, this is particularly significant when passing control information. To ensure accuracy an effective form of control should be used to enable handshaking to take place between the two processors.

The transfer system must also be able to differentiate between each form of data. The final characteristic required in this communication system is the speed of the transfer. The maximum size of each image to be transferred is 172.8 kilobytes and the processor can process about thirty frames per second. Therefore the minimum transmission rate for the video data that must be met should be approximately 5.2 Mbytes/second. At the same time it is estimated that about one kilobyte of processed data will have to be transmitted per frame, this will increase the frame rate to approximately 5.3 megabytes/second. Finally approximately 64 Kbytes of data must be transferred for each calibration transfer. It is preferential to get the system much faster then the maximum data transfer rate.

3.2 Hardware Specifications

The hardware design must perform two major roles in the development of a data transfer system. Firstly it must ensure that all data is transferred accurately and efficiently. Secondly it needs to manipulate control signals and merge the interfaces selected for each processor together.

The hardware in this design can be implemented either externally on a PCB or implemented in VHDL on a Xilinx Spartan II FPGA that, is placed on the SH4 vision board. Power for the components on the PCB is available from both the SH4 and IPAQ. Two voltage ranges are available, both the SH4 and IPAQ can supply a 0-3.3V range the SH4 can also supply an additional 0-5V range.

The adopted hardware will also have to conform to the physical dimensions of each interface. In the case of the IPAQ the physical interface is a specific 100-pin connector manufactured by Foxconn Pty Ltd, the dimensions of this connector are included in appendix G. The Sh4's interface on the other hand was a much more standard 100-pin connector that can be purchased from any components store. The hardware implemented including the SH4 will be mounted on the back of the IPAQ as an expansion pack.

3.3 Software Specifications

The Software in this system will be broken down in two sections; the main processor and the slave processor. It will be the job of the main processor to initiate all communication between the two processors while the slave will need to be able to respond correctly to the main processor's commands. In each case some form of handshaking will need to be handled by software. The other major issue with software is the operating systems used by each processor. In the case of the SH4 this does not present a problem as no operating system is used. The IPAQ on the other hand as was previously stated is running Microsoft's Windows CE; this will mean that direct access to hardware

will probably not be possible as there are restrictions on direct access to hardware. It will be necessary to write a driver that can be controlled by the kernel of the operating system.

4.0 Hardware

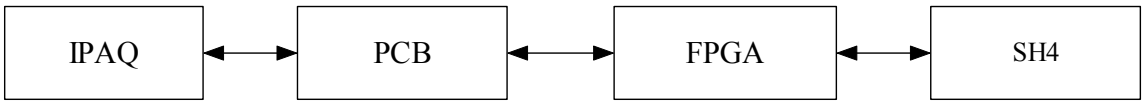

Figure 4.1
Hardware Layout of peripheral Interface

This chapter will analyse the design process that was used in the development of the physical layer of this communication system. There are two principal areas of hardware design the external PCB and FPGA. The actual layout of the design is shown in figure (4.1).

4.1 PCB Design

The first thing to note about the PCB design was that a 0-3.3V voltage range was selected, this meant that the PCB was powered from both the SH4 and the IPAQ. The actual design of the PCB was based upon the selected interface for the IPAQ. The IPAQ has two interfaces that are available for standard 32-bit I/O, SRAM like Variable Latency I/O and PCMCIA. The variable latency SRAM like I/O interface has the advantage of being able to use the DMA features of the SA1110. On the other hand the PCMCIA interface is the standard form of interface used to connect peripheral devices in portable computers. The PCMCIA interface also allows the design to incorporate the hot insertion

and removal features of the IPAQ, unfortunately in this case it does not support direct memory access transfer. Both interfaces were viable alternatives the PCMCIA interface was eventually chosen as it was felt that it had the greatest flexibility, which was important in the earlier stages of design. It also had the benefit of supporting hot insertion and removal, which would be a nice feature if time allowed it to be implemented. This decision was possibly wrong in hindsight as using the DMA capabilities of the SRAM interface would have reduced the strain on the main processor.

The PCB design implemented was a slightly modified version of the standard PCMCIA interface as shown in figure (2.4) obtained from the SA1110 datasheet [4]. The circuit was modified slightly as the standard PCMCIA interface is designed to handle both a memory interface and a standard I/O interface. This design has simply removed the redundant memory logic in favour of a more specific I/O interface. To do this the control signals dedicated to memory access were omitted. Only the /IORD, /IOWR, /IOIS16, /WAIT, Reset and /CD control signals have been included in this design. The PREG signal has also been included but is essentially useless as it is a memory specific signal.

Pull up resistors have been used for all control signals except /IORD and /IOWR. This design was based upon the generic PCMCIA socket shown in figure [2.4]. A pull up resistor is not needed for the IOIS16 signal as this signal is permanently tied low a pull up resistor will simply result in a continuous power consumption and reduce the overall efficiency of the circuit. The pull up resistors used are compliant with the PC Card Standard Volume 2 Electrical Specification as shown in [5].

The standard PCMCIA design has been further modified by extending the data bus from sixteen bits to thirty-two bits, this is beyond the specs of the PCMCIA standard but the Strong Arm processor allows for this extension as is shown in [4]. Restricting the bus to thirty-two bits has altered the way the logic was implemented. Using the thirty-two bit format means that the /IOIS16 signal must always be asserted and A0 and A1 are always equal to zero. The two /CE signals must also be or'ed together to form one chip enable signal. This was done as the chip enable signal is only required to low when both /CE1 and /CE2 go low.

This design only implements an eight-bit address bus. Only eight address bits were needed in this transfer because this form of communication really only uses two different addresses; one address for character interrupt and one address for a DMA request. It must be noted that two of the address lines that I have included are A0 and A1 these lines were essentially useless as they are always zero for a thirty-two bit transfer.

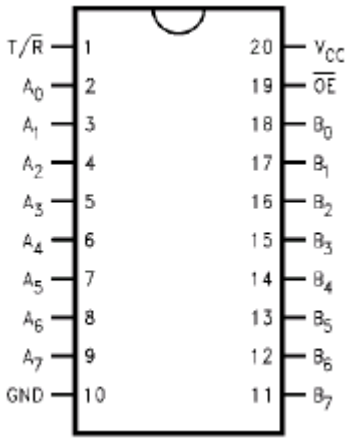

Figure 4.2
74LCX245 Buffer [17]

To reduce the degrading effects of excessive capacitance on the switching speeds of the signals the circuit uses 74LCX245 bi-directional buffers. The pin out for this data buffer is shown in figure 8. The 74LCX245 buffer was chosen for its small propagation delay of 7ns and the low power consumption. The data sheet for 74LCX245 is given in [17].

The output and direction of the 74LCX245 can be controlled by using the /OE pin and T/R pin of the buffer. When the /OE pin is pulled low the buffer will begin to operate, the direction of the buffer is then controlled by the T/R pin. When this pin is pulled low the signals will travel from B to A as shown in figure (4.2). The signals will propagate in the opposite direction when the pin is held high.

Control logic is used to control the output and direction of the buffer this increases the overall efficiency. There are three types of buffers used by this circuit; Address buffers, control signal buffers and data buffers all are controlled by different logic.

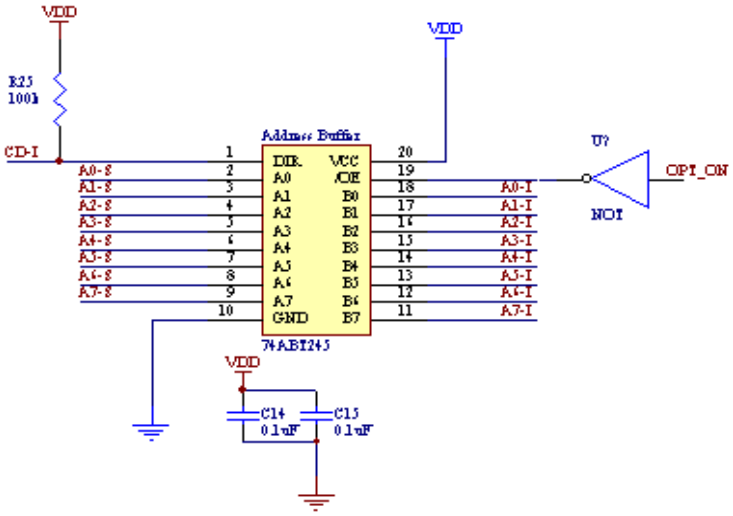

Figure 4.3
Address Buffer

The Address buffer as is shown in figure [4.3] is enabled when the OPT_ON signal is pulled high. When the /CD signal is then pulled low the signal will propagate from the IPAQ to SH4.

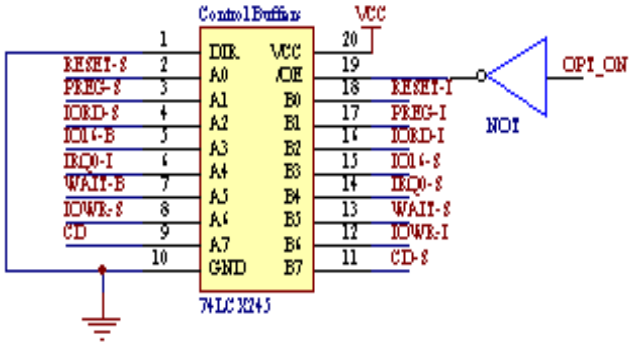

Figure 4.4
Control Buffer

Similarly the control signal buffer shown in Figure[4.4] is also enabled when the OPT_ON signal is asserted. The direction pin is permanently tied low; the control signals will always therefore propagate from B to A.

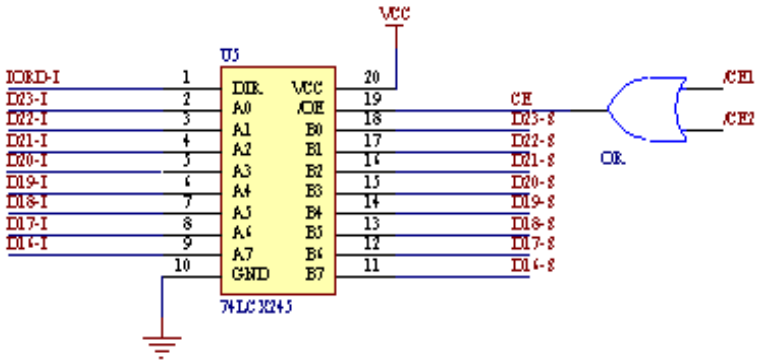

Figure 4.5
Data Buffer

Finally the Address buffer shown in figure [4.5] is enabled when /CE1 and /CE2 pins are pulled low, the direction of the buffer will then be controlled by the /IORD signal. When IORD is low the data signals will propagate from the SH4 to the IPAQ. Alternatively when /IORD is high during a write operation the signals will propagate from the IPAQ to the SH4.

All control logic used in this circuit is based the based around the 3.3 V Low Voltage HCMOS series. This type of logic was selected for its low voltage rating of and low propagation delay of 2.5 nanoseconds.

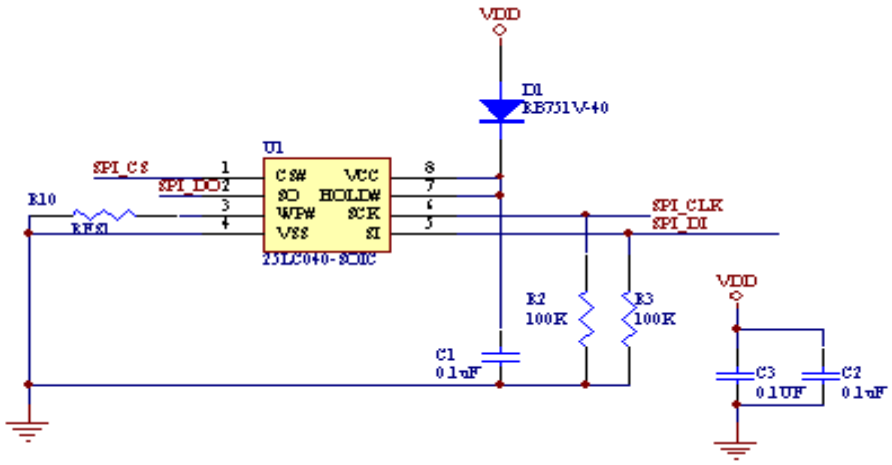

Figure 4.6
Serial EEPROM

As part of the initialisation process a 25LC080 serial EEPROM has been included on the PCB. This serial EEPROM contains the initialisation information required to start the driver and initialize the hardware in software. A detailed description of the data contained within the serial EEPROM is displayed in [18]. The hardware design of the serial EEPROM was based upon the data contained in the 25LC080 and the schematics contained in the IPAQ developer's kit [21]. The full circuit for the PCB is displayed in Appendix A.

4.3 FPGA Design

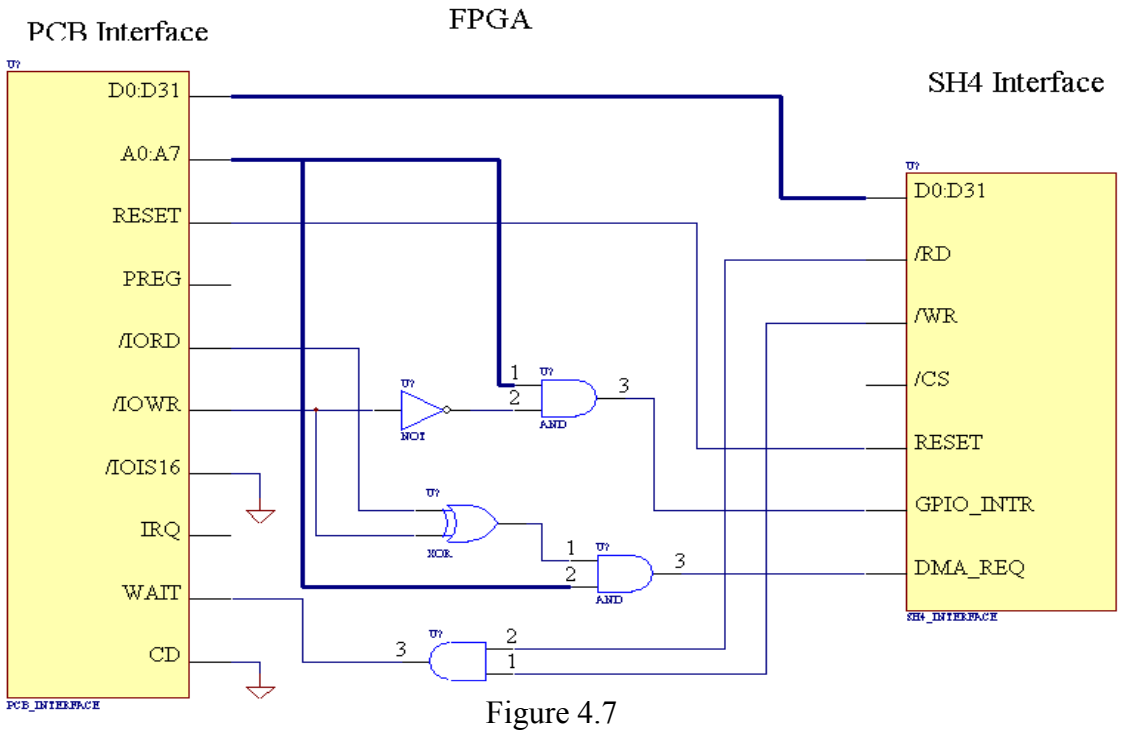

Figure 4.7
FPGA Logic

This design uses a Xilinx Spartan FPGA situated on the SH4 board to interface the PCB to the SH4 processor. FPGA provides three major functions; firstly it initialises the PCB hardware, it then connects the major signals to the SH4 and finally it provides the logic to translate the signals from the PCB into appropriate signals for the interface chosen for the SH4.

The design of the FPGA was based on the chosen interface. A SRAM like interface was chosen for its simplicity and ease of use. This interface provided control signals as listed in table (2.2) these control signals are synchronous, the timing diagram is presented in figure (2.5).

The first function of the FPGA hardware is to initialise the PCB hardware. To do this the FPGA when first connected, will pull the CD signal low, this will intialise the Address bus and the IOIS16 and WAIT control signals. The FPGA will then pull the IOIS16 signal low; this will set the IPAQ data bus to 32-bits. This circuit then connects

the data bus from the PCB to data bus on the SH4 and connects RESET signal to general purpose I/O pin two.

The logic implemented by the FPGA is used to adapt the signals from the IPAQ to the signals used by the SRAM interface. The functionality of the FPGA logic will be examined in the functionality section of this chapter.
This logic was implemented in VHDL code this code is shown in Appendix B. The logic was then compiled and simulated using the Xilinx foundation software.

4.4 Functionality

The functionality of this circuit can be analysed by examining the circuits in figure (4.6) and appendix A. It can be seen that when the expansion pack is initially connected to the IPAQ the pins /ODE1 and /ODE2 are pulled to ground, this will cause an interrupt to occur. Software will then initialize the expansion pack and the OPT_ON signal will then be set high, this causes output enable pin on address buffers and control signal buffers to be pulled low and switches on the buffers, it will also switch on the startup led.

When the SH4 is then connected to the PCB the /CD signal is pulled low, this will remove the buffer from all the control signals going to FPGA. The IOIS16 signal is also tied low when the SH4 is attached to the expansion pack. This was done to set the data bus size to 32-bits permanently. The data bus is then connected to the data bus on the SH4 and reset signal is connected to a general-purpose I/O pin.

After the circuit has been fully initialised the functionality can be fully analysed by examining the timing diagram displayed in figure (2.1). When a read or write occurs the chip enable signal pulls the output enable pin on the data bus buffers low, this triggers these buffers to start operating, the address of the read or write is also placed on the address bus. The IOIS16 signal has been pulled low therefore the PCMCIA transfer is setup for a 32-bit transfer. 3*(BS_xx+1) clock cycles later, the /IORD or /IOWR signal goes low depending on whether the IPAQ is performing a read or write operation. If the IPAQ is performing a write operation to the address 0x2000 0010 in PCMCIA I/O space the IOWR signal will cause a general purpose I/O interrupt. Alternatively a write or read

from the I/O address 0x2000 0008 in PCMCIA I/O space will cause /IORD or /IOWR to cause a DMA request to occur.

To examine the SH4's response it is necessary to refer the timing diagram displayed in figure (2.5). When responding with a read or write operation the /RD or /WR signals will be pulled low, this will assert the wait on the PCB and will cause the IPAQ to hold the data bus until the read or write operation is finished. It must be noted that this is optional as if the timing of each processor is set up properly this will be unnecessary.

5.0 Software Design

To control this hardware with the SH4 and IPAQ required the development of driver software. The driver software design of this embedded system was divided into two sections the IPAQ software and the SH4 software.

5.1 Protocol

Before commencing the development of the software it was important to establish the protocol layer that this software will be based upon. The protocol of this system has several functions that it must perform. Firstly it must decide the responsibilities of each processor. Secondly it provides control for the data transfer and ensures accuracy. Finally it must be able to distinguish between the three types of data transfer.
To develop an effective form of communication it was first necessary to delegate the duties of each processor. It was decided to use the IPAQ as the master and the SH4 as the slave. This method was chosen because in this system the IPAQ is the main processor and therefore controls when information is needed.

As illustrated in the previous chapter there are three forms of data that need to be transferred; video data, processed data and calibration data. To distinguish between each type of data the IPAQ will initially send a character to the SH4 at the start of every new buffer transfer this operation was based on the suggestion of Mark Chang. Based upon this character the SH4 will initialise either a video transfer, processed data transfer or

calibration data transfer. Each transfer will then transmit the entire buffer of data before commencing a new transfer.

During each transfer the receiving processor will need to know the size of the block of data to be transferred. This is not an issue in the transmission of calibration data or video data to the SH4 as these buffers are always the same size. The processed buffer however makes use of run time length encoding therefore the size of data to be transferred is constantly changing. The first read of every transfer from the IPAQ will return an unsigned integer from the SH4 this will be the size of the data buffer to be transferred. The IPAQ will then continue to perform read operations until the entire buffer is transferred. The system will then be reset ready for the next transfer.

5.2 IPAQ Software

The basic function of the IPAQ software driver is relatively simple the task of designing the driver becomes more complex when designing the driver to operate under windows CE. At the present time the design of the IPAQ software is relatively incomplete, this thesis will analyse the present solution.

A monolithic native driver was chosen to implement this design as this allowed the interface of the driver to be custom designed to suit the device, this driver model is also the fastest model available. At the present time the Driver has four major functions that interface to the user level software DLL_main, PCM_Read, PCM_Write and PCM_DeInit. This chapter will discuss how these functions are implemented it will also discuss how the driver is installed and accessed. The actual code is displayed in appendix D.

5.21 DLL_main

The DLL_main function is used to initialise the physical hardware used in this communication process. This function will be automatically called every time the driver is installed into a API.

To initialise the physical hardware it initialises access to the registers, the data bus, the PCB hardware and initialises the timing of the PCMCIA transfer.

Direct access to the physical registers using the physical address of the register in user mode is not possible. This function must first allocate a position in virtual memory and then virtually copy the entire physical address to this virtual memory position. The DLL_main function allocates a position in virtual memory by using the VirtualAlloc function. This function reserves a block of virtual memory of the size of a system page all accesses to this memory are then disabled. The VirtualCopy function then binds the physical memory address of the register to the virtual memory allocated. This block in virtual memory is then enabled for read and write access and the caching to this area is disabled so that the value is written straight to the memory address. A pointer to this memory position is then returned, and this is used to access the physical register or memory position. This was then done for both the PCMCIA registers and data bus. An example of this code is shown in figure (5.1). For more information on these functions refer to Microsoft's platform builder general help [16].

```
#define PHYSADDR  ((PVOID)0x10000000)
#define SIZE  (Size)
LPVOID lpv;
BOOL bRet;
// pointer to virtual address
lpv = VirtualAlloc(0, SIZE, MEM_RESERVE, PAGE_NOACCESS);
bRet = VirtualCopy(lpv, PHYSADDR, SIZE, PAGE_READWRITE |
PAGE_NOCACHE);
```

Figure 5.1

Code for accessing physical Addresses in Windows CE

The expansion pack is initialised using the libraries contained in the IPAQ software development kit [22]. These libraries can be attached to this code by linking the libraries using the compiler linking options. The functions can then be accessed like an ordinary function. At this point in time only the PPC_SET_POWER function is used. This function sets the OPT_ON signal and initialises the buffers on the IPAQ. Future editions of the code could implement this on the expansion pack interrupt and include the other functions such as reading data from the serial EEPROM. This code has yet to be implemented.

Finally the DLL_main function initializes the timing of the PCMCIA transfer. Storing a value in the BSIO 0 0:4 bits in the Expansion Memory Configuration register (MECR) will control the timing of the PCMCIA I/O slot. This value will set the BS_xx parameter in the timing diagram in figure (2.1). The timing is then set to fast mode by setting the Fast0 bit in the MECR this reduces the setup time of the control signals from 3*(BS_xx+1) to 1*(BS_xx+1). The actual values for the timing in the control signals has not yet been determined and will probably have to be determined experimentally using a oscilloscope.

5.22 PCM_Read

At this point in time there is only one read function in this code. There will eventually be two read functions one for the processed data and another for the video data; these two functions will be modeled from the PCM_Read function.

The PCM_Read function has two major tasks; firstly it will initialise the type of transfer that will take place, secondly it will then read in the data and store it in a data buffer. A pointer to the data buffer and the size of the stored data will then be returned.

To initialise a read operation the PCM_Read function first writes a character to the PCMCIA character address this will then cause the SH4 to initialise the transfer. The function will then read in an unsigned integer and store it as the variable buffer_size this is the size of the data buffer on the SH4 to be transferred.

To read in the data the PCM_Read file first performs a read from the Data bus this will return an unsigned long value from the SH4. This value is then stored in the data buffer

and the pointer to the data bus is then incremented to the next position. This process will then continue until the entire data bus is transferred.

5.23 PCM_Write

The PCM_Write Function is used to write the information stored in the calibration buffer to the SH4. It takes a pointer to the Calibration buffer as a parameter and returns void. Like the PCM_Read function it has two major functions, firstly it initialises a calibration transfer and secondly it transfers the Calibration buffer to the SH4.

PCM_Write initialises the transfer in much the same way as the PCM_Read function. The function first writes the character ‘c’ to the PCMCIA character address, this initialises the SH4 to get ready to accept a calibration buffer transfer. The major difference with this transfer is that size of the calibration buffer is known and therefore does not need to be transferred.

To transfer the data to the SH4 the PCM_Write function reads an unsigned long from the Calibration buffer and then writes it to the to the data bus. This is repeated until the entire buffer is transferred.

5.24 PCM_Close

The PCM_Close function is used to unassign the resources used by this driver. It releases the driver’s hold on the virtual memory allocated for the registers and data bus and frees the memory used for the read buffers. The function takes no parameters and returns a Boolean value.

The function first uses the VirtualFree function to free the virtual memory assigned to the registers and data bus. The FREE function is then used to free the memory allocated to the data buffers. The function then returns ‘True’ if the operation was successful. The memory buffers have not yet been implemented in this code.

5.25 Installing Driver

To be effective the driver must be correctly installed into the system, this section will analyse the way this driver is installed on to the IPAQ and accessed by the user level. This information is obtained from [10]

To install the driver on to the IPAQ the compiled DLL file was transferred into the /Windows directory. No registry key was created for this driver as it is only accessed by one application at a time.

When the hardware needs to be accessed by API on the user level the API uses the LoadDriver function. This function maps the code into the address space of the API calling this code. The function takes a pointer to the string "/Windows/Sh4Drv.DLL" which is the driver file that we have implemented and returns a handle to the instances. The API can then access the hardware through the interface that has been detailed in this chapter.

5.3 SH4 Software

The SH4 has been designated the slave in this communication system, for this reason the main function of the SH4's driver is to respond to the commands issued from the IPAQ. Based upon the given specifications the driver was initially divided into three major routines the read video data, read processed data and write calibration data routines. The initial design of the software driver used a separate interrupt to trigger each 32-bit word transferred. This initial design proved to be unacceptably taxing on the processor as to transfer just one frame required 44550 interrupts, well beyond the processors capabilities. It was decided that driver for the SH4 should make use of direct

memory access capabilities to increase the speed of the transfer and remove the latency from the main processor.

The SH4 driver included five major functions Initialise_IPAQ, IPAQ_Read, Calibration, Video_data and Process_data. This paragraph will analyse the initialisation process, the IPAQ_READ function, a Write operation, a Read operation and finally the Interrupt handling process. This code uses lower level code written by Mark Chang to access the hardware.

5.31 Initialisation

This process will initialise the various components of the data transfer including the data buffers, the DMA transfer, the data bus and interrupts. The function that performs the initialization in this code is called Initialise_IPAQ. This function is called from the file 'main.c'.

To initialise the data buffers the code must enable access to the memory where the buffers are stored. To access the data stored within a data buffer this global pointers of type unsigned long are used. This was done so that all functions within the code could have access to the data buffers without having to receive the pointer as a parameter. The actual buffers are initialized by the vision code. The initialisation function of this code sets the pointer for each buffer to point to the address of the first word of data in the buffer.

The next step in the initialisation process is to setup the direct memory access controller. This code was based upon the information contained in the SH4 data sheet [2]. The DMA channel selected for this transfer was channel 0 this was chosen as the channel 1 is being used to transfer video data from the camera. These two channels are the only channels that can be accessed externally by pulsing the DREQ pin.

The DMA transfer needs to be able to control both the source and destination and must be triggered externally. For these reasons dual address externally accessed mode was selected by clearing the RS0:RS3 bits in the DMA operations register (DMAOR). This mode means that both the transfer source and transfer destination are accessed

through the addresses stored in the source address register and destination address register and the transfer is triggered externally by the DREQ pin.

A single transfer will occur when the DMA request coming from the IPAQ goes low, the DREQ pin therefore needs to be set to trigger on a falling edge of the input signal rather then when the signal is low. This is necessary so that only one read or write operation will occur per DMA request. This mode was set by setting the DREQ select (DS) bit in the channel control register to high.

The size of the transfer must then be set. This communication system transfers 32-bits per transfer therefore the transfer size was set to 32-bits. Setting the TS2:TS0 bits in the channel control register to 001 did this.

There are two bus modes available to a DMA transfer; cycle steal mode and burst mode. In cycle steal mode the DMA controller will hold the bus until the DMA transfer has been completed. In burst mode the DMA controller will relinquish control of the bus to the CPU as soon as the DREQ signal goes high. The cycle steal mode was chosen as each transfer, needs to transfer the full 32-bits regardless of whether the DREQ pin goes high. Clearing the Transmit Mode (TM) bit of the channel control register set this mode of operation. Finally channel 0 was made the highest priority DMA channel by clearing the PR1:0 bits in the DMA operations registers. This was done for testing purposes only as in reality the camera would have a higher priority.

Finally the initilisation function is required to setup the general-purpose interrupt used to initialise the type of data transfer to take place. The first general purpose I/O pin was chosen for this interrupt simply because it was first. The initialisation function must initialise this pin to an input and then enable the interrupt. To set the direction of this pin the PB0IO bit within the port control register must be cleared. Clearing the PB0IO pin in the port control register initialised the direction of the pin. Setting the port interrupt enable (PTIREN0) bit in the GPIO interrupt control register to high will then enabled the GPIO interrupt.

5.32 IPAQ_READ function

As previously mentioned this communication system has three different types of data transfer, this function is responsible for determining the type of transfer to initiate. At the beginning of each data transfer the IPAQ will send a character across to indicate the type of transfer that needs to be initialized, this will cause an interrupt on the general-purpose port one. The Interrupt handling routine IPAQ_READ will read in the character from the data-bus and store it as temporary variable ‘mode’. A case statement is then performed on the character. If the character sent is a ‘c’ the routine will jump to the calibration routine, alternatively if a ‘p’ is sent the processed information routine will be called and finally if a ‘v’ is sent the video data routine will be called. The general purpose interrupt will then be cleared and the function will return a Boolean value of true if the routine successfully calls one of the three functions. If a character other then the ones previously mentioned are sent the function will return false.

5.33 Write Operation

When a processed data or video data transfer operation is initiated from the IPAQ_READ function the SH4 will initiate a DMA write operation. This operation will setup the DMA controller to transmit from the given data buffer to the data bus.

To perform the correct transfer it is necessary to know the size of the data stored in the buffer. To find the size the function reads in the unsigned integer stored at the first address in the buffer it then stores this in a temporary size variable.

To initiate the source address of the transfer the address of the corresponding data buffer is stored in the DMA source address register. When a DMA transfer is triggered the DMA controller will then read the data stored at this address. The function then sets the source address to increment after every data transfer by setting the SM0 and SM1 bits in the channel control register to 10. After each transfer the DMA controller will point to the next data address in the data buffer ready for the next transfer.

The destination address register is then set to the first address of area five of the bus state. This area is setup as a SRAM interface. When the DREQ pin triggers a DMA

transfer the DMA controller will read from the Memory buffer and then transmit it to the Data bus, this data will then be sent to the IPAQ. The destination address mode was then set to stationary mode by clearing the DM0 and DM1 bits in the channel control register 0. This was done so that the DMA destination address always points to the data bus address.

The size of the data transfer can be set by storing the size of the data buffer in transfer count register (DMATCR0). The size of this transfer was set to the value stored in the temporary size variable incremented by one to account for the initial unsigned integer. This value will then be decremented after each transfer until the data bus is completely transferred.

After the source and destination addresses have been initialised the IE bit in the Channel control register is set, this enables the DMA transfer interrupt, which will signal the end of the DMA transfer. Finally the DMA transfer is enabled by setting the DE bit in the channel control register to enable the DMA channel and then setting DMA master enable (DME) bit in the DMA operations register enabled the DMA transfer.

Each transfer may then be triggered through pulsing the DREQ pin. Once the entire buffer has been transferred the TE bit will be set to indicate the transfer has ended. This will cause the DMA transfer interrupt to occur and the interrupt routine *dmac_handler* within the file *main.c* to be called. This file will disable the DMA transfer and re-enable the GPIO interrupt. This is done to prevent any illegal DMA transfers and to initialise the GPIO interrupt.
There are two separate functions that handle this for video data and processed data these functions are called Video_Data and Processed_Data.

5.34 Read operation

When the calibration mode is selected the SH4 is required to read in data when the DREQ pin goes low and store it in the calibration buffer. The read operation is similar to the write operation but varies in a few key areas. Firstly the source address is now set to the data bus and the source mode is set to stationary. Secondly the destination address is now set to the calibration buffer and is set to increment mode. Configuring the DMA

transfer like this will mean that when the DREQ pin is pulled low the DMA controller will read from the data bus and store the result into the calibration buffer. The destination will then be incremented to the next position in the calibration buffer ready for the next transfer. The size of the transfer will always be constant therefore the size of the transfer can simply be set as in the read operation. Finally the DMA interrupt will be setup the same as the read operation and will jump to the same interrupt routine.

5.35 Interrupt handling

At the point of writing the interrupt handling routine was based on a look up table system. The interrupt routine is in the assembly file 'entry.s' and has been written by Mark Chang.

When an interrupt occurs the interrupt controller first selects the highest priority interrupt. The interrupt source code is then stored in the interrupt event register (INTEVT), the status registers are then saved and the CPU jumps to the start of the interrupt handler code. This interrupt handler code within '*entry.s.*' then saves all the registers on the stack, it then looks up the interrupt handler vector in a look up table and then jumps to the interrupt routine.

This code uses two major interrupts the general purpose IO interrupt and the direct memory access interrupt. To initialise the interrupts it was necessary to first set the priorities of these interrupts and then write the interrupt handler.

The priority of an interrupt can range from the highest priority of 16, which will always occur to the lowest priority of 0, which will never occur. The priority of the DMA interrupt is set using bits 11-8 in interrupt priority register C while the priority of the GPIO interrupt can be set by using bits 15-12. This was set in '*main.c*' while the interrupt handling routine was in both '*main.c*' and '*entry.s*'. The priority in both cases was set to a 6. When the interrupt occurs the program will jump to the interrupt routine in the entry.s this will then call the individual service routines.

The way that the software initialises an interrupt may change as Mark Chang has now restructured the interrupt handling code so that interrupts can now be dynamically allocated. Work is in progress to adapt the code for this communication system to suit

this new system of allocating interrupts.

6.0 Results

The results of a project often indicate how successful the design is unfortunately due to the natural dependency of each section of this project getting results without all parts working was difficult. This Chapter will outline the results that were achieved in the development of the PCB the FPGA programming, the SH4 Driver and finally the IPAQ Driver.

6.1 PCB

The PCB design as mentioned in chapter three was fully implemented and constructed. To verify the operation of the PCB the power, the control signals and finally the buffering logic was tested.

The first thing that was tested on the PCB was the power and ground planes. The IPAQ was first connected to the PCB so that power was applied to the circuit. Using a multi-meter it was verified that there were no short circuits and power was delivered to each component. The initial states of the control signals were then verified. Some abnormalities were found in the initial states of the control signals. The /IORD and /IOWR signals were grounded when it was expected that they would in fact be at 3.3 volts. The circuit was checked and it was determined that they were not tied to ground. Based upon the IPAQ data sheet no pullup resistor is necessary for these two signals. The actual pins on the IPAQ were then tested and it was found that they were also grounded. These signals are active low and therefore may need to be pulled up to VCC. The remainder of the control signals performed as expected.

The operation of each buffer and the buffer logic was then tested to see that the operation of each buffer was correct. Firstly the OPTON and CD signals were tied low to simulate the initilisation of the PCB. A signal was then applied to the address buffer and control signals as expected the alternate pin followed the source pin. The signals /CE1 and /CE2 were then pulled low and IORD signal was used to control the direction of the data buffers. Once again the buffering and buffering logic performed as expected.

All of the logic met the specified performance but in the interest of speeding up the development most of this logic although operating correctly was circumvented. This reduced the efficiency of the circuit but made the testing of other areas more efficient.

6.2 FPGA

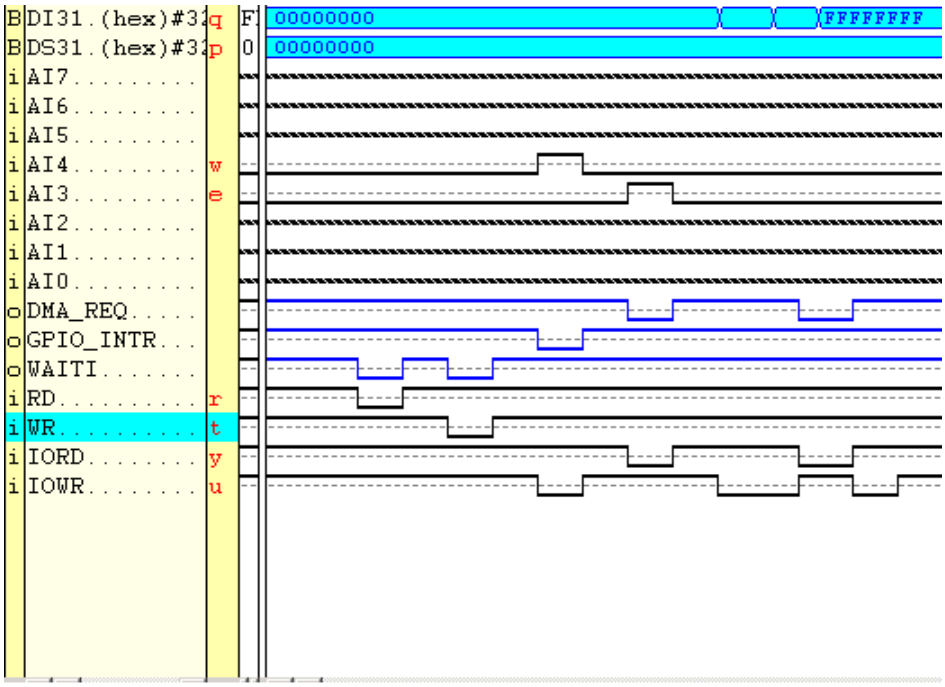

Figure 6.1
FPGA Logic Simulation

At the time of writing the VHDL code has been written and has been simulated. FPGA has been written and is in the process of being simulated. The timing logic for the logic is shown in figure (6.1).

From this simulation diagram it can be seen that all of the logic is functioning properly but problems are being experienced with the I/O bus, for some reason the output will not follow the input when a I/O bus is used. As yet the cause of this problem is yet to be determined.

6.3 SH4 Driver

The SH4 driver operation was very hard to verify. As was previously stated the SH4 driver's main task is to respond to the IPAQ's signals. Therefore without a working IPAQ driver and with the FPGA code as yet not loaded onto the FPGA the full operation of the SH4 driver could not be tested. The DMA operation was however verified to some extent by setting the DMA transfer size to one, the end of transfer interrupt was then set to trigger the on board led three. This operation was successful therefore some form of transfer was taking place. This experiment was then repeated with the GPIO1 interrupt and once again the operation was verified. As to whether the DMA transfer was taking place this could probably have been tested by using an external trigger but performing an internal transfer and then verifying the operation had taken place. This could be tested through the use of the debugger program.

6.4 IPAQ Driver

Most of the time spent on this project was spent on developing a working IPAQ driver. It was initially thought that the easiest way to implement the driver was to implement it within the API itself rather then letting the kernel operate the driver. Although it has previously been stated in the theory that this is not the best way to access the hardware it was thought to be possible based upon the information obtained from [19]. A program was designed to read the information stored in read only memory of the IPAQ to test this tested theory. This was done by copying the physical addresses to a virtual address and then reading the information from the virtual address. This program was successful and so this system was adapted into the IPAQ driver code. When tested this code resulted in read access violation errors. From the results gathered it was concluded that is possible to read or write to a storage memory but not to a registers or I/O ports.

It was then decided to adapt the existing code to the native driver approach as described in chapter three. The driver could then be accessed with the LoadDriver function. This code produced much better results as no read or write accesses violations were experienced. The IORD and IOWR signals were then tested for proof that the system was indeed operating unfortunately this is when it was discovered that there was a slight problem with these two signals and no further tests have been performed.

Although at the point of writing the IPAQ driver is still not operational it was felt that with knowledge obtained through testing an operational IPAQ driver before the demonstration day was not unrealistic.

7.0 Conclusions

The goal of this thesis was to develop a high-speed data transfer system with low overhead and high accuracy. This thesis has developed a model for a communication system to connect the IPAQ to the SH4 processor. It has analysed how a specifically I/O PCMCIA interface can be implemented in hardware on a PCB. It has then illustrated how this PCMCIA socket was connected to a SRAM interface on the SH4 using a FPGA. Driver software to operate this hardware has also been developed for both the IPAQ and SH4.

From this work several key results have been reached. Firstly the logic in this design will function correctly under the stimulus described in the PCMCIA timing diagram figure(2.2). Although the IORD and IOWR did not perform as was specified in the data sheet. Secondly hardware has been developed for the FPGA but as yet has not been properly implemented. Thirdly a large amount of progress has been made on an IPAQ driver. A method for accessing hardware and loading a driver in an API has been detailed. Finally a driver has been developed for the SH4 using a DMA transfer. Continuing work needs to be done on all of these areas of design.

Although this project has failed to develop a working prototype it has developed a model on which a successful data transfer could be based.

7.1 Future Work

Further work needs to be done in several different areas of this project.

Firstly to greatly reduce the overhead of this design a DMA transfer could be implemented by the IPAQ driver as well as the SH4 driver. At this point in time with the SA1110 as the microprocessor this would mean changing the PCMCIA interface to an SRAM like variable I/O interface. PCMCIA was possibly not the best solution in this situation as its only advantage is that it has a standard interface and also standard software architecture that will make designs easy to port between different processors. In a specialized case such as this the SRAM interface may perform better.

This design does not consider possible transfer errors, as this was not a high priority. To better improve this transfer a feedback system for errors needs to be developed to notify the other processor that an error has occurred. A feedback system from the IPAQ to the SH4 would be easy to implement as this could be worked into the character interrupt. Transferring an error from the SH4 to the IPAQ on the other hand could not be done with the existing system. This could be done with the unutilized IRQ0 interrupt. Using this interrupt the SH4 could then initiate a transfer and report any errors to the IPAQ, which could then deal with them.

The actual implementation of this hardware could be improved by implementing more of the hardware in the FPGA. From the information that has been gathered through out this project it was noted that FPGA could be used to implement most of the logic including the buffering and the buffering logic. This would make the system much more flexible and more cost effective as it would reduce the size of the FPGA and reduce necessary components. If the PCMCIA interface is used again I would suggest that a more generic PCMCIA interface is implemented. Designing a specifically I/O interface has removed some of the flexibility of the PCMCIA interface

In regards to the IPAQ driver it was concluded that there were two possible directions for this part of the project to progress. The first option is to continue with the native driver for Windows CE. It was felt that this would give the system the best overall performance.

The second option is to use the existing PCMCIA driver for windows. This driver has all the necessary functions plus extra functions that are probably not required. This

will mean that the transfer will have to be reduced to 16-bits. This is not really a problem as a 32-bit transfer is really overkill at this stage with the limited amount of data that must be passed from the vision processor.

The third option is to move to a more designer friendly operating system such as Linux. The benefit of this is that there is more information available on these drivers and example code. The information is also available free whereas information on Windows CE is expensive. Although there are other considerations before the decision to change the operating system is made as this will quite possibly mean that the other software to go on the IPAQ such as the walking software will also have to be ported.

7.2 Significant Outcomes

- Hardware has been developed to interface the IPAQ to the SH4. This hardware implements a I/O specific PCMCIA socket
- FPGA code has been written to interface the PCB to the SH4.
- A software driver for the IPAQ has been developed and is nearly operational.
- A driver for the SH4 has also been written but the operation has not been fully verified.

References

[1] Sung-Hyan Han, W.H. Seo, S.Y Lee, S.H. Lee, H. W Lee Higuchi Toshiro, Kyungnam, *A Study on Real-Time Implementation of Visual Feedback Control of Robot Manipulator*, 1999 IEEE International conference on systems and Cybernetics, Vol 2.

[2] Hitachi, SH7750 series Hardware Manual, rev 5 2001

[3] Compaq, IPAQ H3000 series expansion pack development guide, Literature No 213235-002.

[4] Intel, StrongARM SA-1110 Microprocessors Development Manual, Literature No 278240-003

[5] Anderson D, *PCMCIA System Architecture – 16-Bit PC Cards*, Addison-Wesley, Harlow, England, 1995.

[6] Schmidt F, The SCSI Bus and IDE Interface- Protocols, applications and programming, Addison-Wesley, Harlow, England 1998

[7] Hague F, *Inside PC Card – CardBus and PCMCIA Design*, Butterworth-Heinemann, Newton, MA, 1996.

[8] Microsoft Press, *Microsoft Windows CE Programmer's Guide*, Microsoft Press, Redmond, WA, 1998,

[9] Brown, Vranessic, Fundamentals of Digital Logic with VHDL Design, McGraw-Hill, Singapore, 2000.

[10] Microsoft, Microsoft Windows CE Platform builder 3.0

[11] Brooks, Dickins, Zelinsky, Kieffer, Abdallah, *A High-Performance Camera Platform for Real-Time Activision*, The Australian national University Canberra, FSR97-Active, 1997.

[12] Keane D, High Speed Communication for Robots- A RoboRoos Project, University of Queensland, 1999.

[13] Scasellati B, A Binocular, Foveated Active Vision System, MIT, scaz-3heads, 1999

[14] Prasser D, Vision Software for Humanoid soccer, University of Queensland, 2001

[15] Blower A, Development of Vision System for Humanoid Robot, University of Queensland, 2001

[16] Microsoft, Microsoft Driver Development Kit, 2000

[17] Fairchild Semiconductor, 74LCX245 Datasheet, www.fairchildsemi.com

[18] Microchip, 25LC080 serial EEPROM data sheet, www.microchip.com

[19] Microsoft, CEGadgets, www.CEGadgets.com, 18/10/01

[20] Compaq, Compaq Solutions Alliance, http://csa.compaq.com, 18/10/01

[21] Compaq, Ipaq developers Manual,

[22]Compaq, IPAQ Software Development Kit, http://csa.compaq.com 18/10/01

Appendix A – PCMCIA Socket

Appendix B – VHDL Code

```
library IEEE;
use IEEE.std_logic_1164.all;

entity IPAQ_LOG is
   port (
      DI: inout STD_LOGIC_VECTOR (31 downto 0);
      AI: in STD_LOGIC_VECTOR (7 downto 0);
      DS: inout STD_LOGIC_VECTOR (31 downto 0);
      RESETI: in STD_LOGIC;
      RESETS: out STD_LOGIC;
      WAITI: out STD_LOGIC;
      WR: in STD_LOGIC;
      RD: in STD_LOGIC;
      GPIO_INTR: out STD_LOGIC;
      DMA_REQ: out STD_LOGIC;
      IOIS16: out STD_LOGIC;
      CDI: out STD_LOGIC;
      IORD: in STD_LOGIC;
      IOWR: in STD_LOGIC
   );
end IPAQ_LOG;

architecture IPAQ_LOG_arch of IPAQ_LOG is
begin
  -- Control Signals
  RESETS <= RESETI;
  WAITI <= RD and WR;
  -- GPIO interrupt
  GPIO_INTR <= not(AI(4) and (not IOWR));
  -- DMA request
  DMA_REQ <= not ((AI(3) and (not IOWR)) or (not(IORD)));
  IOIS16 <= '0';
  CDI <= '0';
    -- I/O Bus
PROCESS (IORD, DS, DI, IOWR)
  BEGIN
      IF ((IORD = '0') and (IOWR = '1')) THEN
            DI <= DS;
      ELSIF ((IOWR = '0') and(IORD = '1')) THEN
            DS <= DI;
      END IF;
  END PROCESS ;
end IPAQ_LOG_arch;
```

Appendix C – SH4 Driver

C.1 – Sh4Drv.h

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

                                    sh4drv.h

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

/* data bus registers*/
#define BCR1 0xFF800000
#define BCR2 0xFF800004
#define WSCR1 0xFF800008
#define WSCR2 0xFF80000C
#define WSCR3 0xFF800010
#define MCR 0xFF800014
#define PCR 0xFF800018
#define RTCSR 0xFF80001C
#define RTCNT 0xFF800020
#define RTCOR 0xFF800024
#define RFCR 0xFF800028

/* data bus */
#define DATA_BUS 0x14000000

/* data Buffers */
#define CAL_BUFFER 0x15000000
#define CAL_BUFFER_SIZE 0x16000000
#define VIDEO_BUFFER 0x17000000
#define PROCESS_DATA 0x18000000


/* define interupt register */


// Functions
void Initialise_Ipaq (void);
BOOL IPAQ_Read (void);
BOOL Video_data (void);
BOOL Process_data (void);
BOOL Calibration (void);
BOOL Init_Interrup t(void);
BOOL Disable_Interrupt (void);
```

C.2 – Sh4Drv.c

```
include <common/stdtypes.h>
#include <sh/sh4drv.h>
#include <sh/dmac.h>
#include <sh/GPIO.h>


// Global Variables
unsigned char *PVideoBuffer;
unsigned char *Pdata;
unsigned char *PCalBuffer;
unsigned char *P_Proc;


/* This function intialises the system */
void Initialise_Ipaq(void){
        Pdata = (unsigned char*) DATA_BUS;         // set pointer to data buss

        PCalBuffer = (unsigned char *) CAL_BUFFER; // set to calabration buffer

        PVideoBuffer = (unsigned char *) VIDEO_BUFFER; //set to video data

        P_Proc = (unsigned char *)PROCESS_DATA; //set to processed data

        dmac_set_resource(DMAC_CHAN_0, 0x00);//set to dual address external

        dmac_set_src_mode(DMAC_CHAN_0, DMAC_CHCR_SM0); //intiailise source mode

        dmac_set_transfer_size(DMAC_CHAN_0, DMAC_SIZE_LONG);//set transfer size

        dmac_txmode_cyclesteal(DMAC_CHAN_0);  // set to cycle steal mode

        dmac_set_priority(DMAC_PRIORITY_0123);//set priority.

        Init_Interrupt();
}

BOOL IPAQ_Read (void){
        char mode = *Pdata;
        BOOL result;
        switch (mode){
        case 'c':
                Calibration();                  // perform calibration
                result = TRUE;
                break;
        case 'v':
                Video_data();                   // perform video data
                result = TRUE;
                break;
        case 'p':
                Process_data();                 //          perform process data
                result = TRUE;
```

```
                break;
        default:
                result = FALSE;
        }
        return result;
}

BOOL Calibration(void){
        dmac_set_src_mode(DMAC_CHAN_0, 0x00);// set source mode to stationary

        dmac_set_dst_mode(DMAC_CHAN_0, DMAC_CHCR_DM0);// set destination mode to
increment

        dmac_set_src_address(DMAC_CHAN_0, Pdata);      // set source address to bus

        dmac_set_dst_address(DMAC_CHAN_0, PCalBuffer); // set destination

        dmac_set_trn_counter(DMAC_CHAN_0, CAL_BUFFER_SIZE); // set size of data transfer

        dmac_interrput_enable(DMAC_CHAN_0);

        dmac_enable(DMAC_CHAN_0);   // enable dma transfer

        return TRUE;
}

BOOL Video_data(void){
        unsigned int video_size = *PVideoBuffer;

        dmac_set_src_mode(DMAC_CHAN_0, DMAC_CHCR_SM0); // set source mode to increment

        dmac_set_dst_mode(DMAC_CHAN_0, 0x00); // set destination mode to stationary

        dmac_set_src_address(DMAC_CHAN_0, PVideoBuffer); // set source address to bus

        dmac_set_dst_address(DMAC_CHAN_0, Pdata);// set destination to data buffer

        dmac_set_trn_counter(DMAC_CHAN_0, video_size+1);  // set size of data transfer

        dmac_interrput_enable(DMAC_CHAN_0);

        dmac_enable(DMAC_CHAN_0);  // enable dma transfer

        return TRUE;
}

BOOL Process_data(void){
        unsigned int proces_size = *P_Proc;

        dmac_set_src_mode(DMAC_CHAN_0, DMAC_CHCR_SM0); // set source mode to increment

        dmac_set_dst_mode(DMAC_CHAN_0, 0x00); // set destination mode to stationary

        dmac_set_src_address(DMAC_CHAN_0, P_Proc);  // set source address to bus

        dmac_set_dst_address(DMAC_CHAN_0, Pdata); // set destination to data buffer
```

```
        dmac_set_trn_counter(DMAC_CHAN_0, proces_size+1);// set size of data transfer

        dmac_interrput_enable(DMAC_CHAN_0);

        dmac_enable(DMAC_CHAN_0);   // enable dma transfer

        return TRUE;
}


BOOL Init_Interrupt(void){
  gpio_set_portA_dir(GPIO_DIR);
  gpio_intr_enable(GPIO_INTR);

        return TRUE;
}

BOOL Disable_Interrupt(void){
  gpio_intr_disable(GPIO_INTR);

        return TRUE;
}
```

C.3 – GPIO.h

```
#define GPIO_PULUP1             0x00
#define GPIO_DIR                0x00
#define GPIO_INTR               0x01


BOOL gpio_set_portA_dir(unsigned long direction);
BOOL gpio_set_portA_pul(unsigned long pullup);
BOOL gpio_intr_enable(unsigned short intr);
BOOL gpio_intr_disable(unsigned short intr);
```

C.4 – GPIO.c

```
#include <common/stdtypes.h>
#include <sh/GPIO.h>

typedef struct
{
        volatile unsigned long      PCTRA;
        volatile unsigned short     PDTRA;
        volatile unsigned long      PCTRB;
        volatile unsigned short     PDTRB;
        volatile unsigned short  GPIOIC;
```

```
} GPIO_REGS;

#define GPIO      (*(volatile GPIO_REGS *) 0x1F80002C)

BOOL gpio_set_portA_dir(unsigned long direction)
{
        GPIO.PCTRA |= direction;                    // set direction of GPIO
        return true;
}

BOOL gpio_set_portA_pul(unsigned long pullup)
{
        GPIO.PCTRA |= pullup;                       // Set pull up
        return true;
}

BOOL gpio_intr_enable(unsigned short intr)
{
        GPIO.GPIOIC |= intr;                                // enable GPIO interrupt
        return true;
}

BOOL gpio_intr_disable(unsigned short intr)
{
        GPIO.GPIOIC |= intr;                        // Disable Interrupt
        return true;
}
```

C.5 – main.c

```
/*** setup IPAQ ***/
// Set up IPAQ interface routines
        SetIntrPriority(INTR_GPIO_GPIOI, 6);
        SetIntrPriority(INTR_DMAC_DMTE0, 6);
        Initialise_Ipaq();

void gpio_handler(void)
{
        led3_on();
        IPAQ_Read();
        gpio_intr_disable(GPIO_INTR)
}

void dmac_handler(void)
{
        dmac_disable(DMAC_CHAN_0);
        gpio_
        led3_off();

}
```

Appendix D – Ipaq Driver

D.1 IPAQDRV.c

```
// IPAQDRV8.cpp : Defines the entry point for the DLL application.
#include "stdafx.h"

#include "IpaqDrv.h"
#include <windows.h>
#include <types.h>
#include "p2.h"
#include "pkfuncs.h"
#include <Afx.h>

void PCM_Init(int count, LPVOID Timing);
BOOL pcm_set_count(ulong mode, LPVOID Timing);
BOOL pcm_set_fast(LPVOID Timing);
BOOL write_word(ulong data, LPVOID Data_bus);
int PCM_Read(LPVOID Data_bus);
int PCM_Write(LPVOID Data_bus);
ulong read_word(LPVOID Data_bus);
BOOL pcm_clear_fast(LPVOID Timing);

#define MECRADDR  ((PVOID)0xA0000018)
#define DATAADDR  ((PVOID)0x20000000)
#define DATASIZE          4
#define MECRSIZE          4
#define PCM_MECR_MASK 0x1F
#define PCM_FAST_MASK 0x80000000


LPVOID Timing;
LPVOID Data_bus;
BOOL bRet;

BOOL APIENTRY DLLMAIN( HANDLE hModule,
                DWORD  ul_reason_for_call,
                LPVOID lpReserved
                                        )
{

        Timing = VirtualAlloc(0, MECRSIZE, MEM_RESERVE, PAGE_NOACCESS);

        bRet = VirtualCopy(Timing, MECRADDR, MECRSIZE, PAGE_READWRITE |
PAGE_NOCACHE);

        Data_bus = VirtualAlloc(0, DATASIZE, MEM_RESERVE, PAGE_NOACCESS);

        bRet = VirtualCopy(Data_bus, DATAADDR, DATASIZE, PAGE_READWRITE |
PAGE_NOCACHE);

        PCM_Init(0x0F, Timing);

        return TRUE;
}
```

```
void PCM_Init(int count, LPVOID Timing){
        pcm_set_count(count, Timing);           // Set timing PCMCIA timing Register
        pcm_set_fast(Timing);                   //set to fast transfer
}

BOOL PCM_Read(LPVOID Data_bus){
        uchar video = 'v';
        int buffer_size;
        write_word((ulong)video, Data_bus);     // transmit ‘v’ character
        buffer_size = (int)read_word(Data_bus); // read in buffer size
        return  true;
}
BOOL PCM_Write(LPVOID Data_bus){
        uchar calibration = 'c';
         write_word((ulong)calibration, Data_bus); // transmit ‘c’
         write_word(0xFFFF,Data_bus);           // Write test word FFFFF
         return true;
}

BOOL PCM_Close (void){
        VirtualFree(Timing, MECRSIZE, MEM_RELEASE);     //release timing rgister
        VirtualFree(Data_bus, DATASIZE, MEM_RELEASE);   //release data register
}

ulong read_word(LPVOID Data_bus){
        ulong data = *((ulong*)Data_bus);       // read in unsigned long from data bus
        return data;
}

BOOL write_word(ulong data, LPVOID Data_bus){
        *((ulong*)Data_bus) = data;             // write word to data bus
        return true;
}

BOOL pcm_set_count(ulong mode, LPVOID Timing){
        if (mode <= PCM_MECR_MASK) {
                *((ulong*)Timing) &= ~PCM_MECR_MASK;    //clear timing register
                *((ulong*)Timing) |= mode;              //set timing
                return true;
        } else {
                return false;
        }
}
BOOL pcm_set_fast(LPVOID Timing){
        *((ulong*)Timing) |= PCM_FAST_MASK;     //set fast bit
        return true;
}
BOOL pcm_clear_fast(LPVOID Timing){
        *((ulong*)Timing) &= ~PCM_FAST_MASK;    //clear fast bit
        return true;
   }
```

Appendix E - Photographs

E.1 Vision System

Figure E1.1
Disassembled vision
system

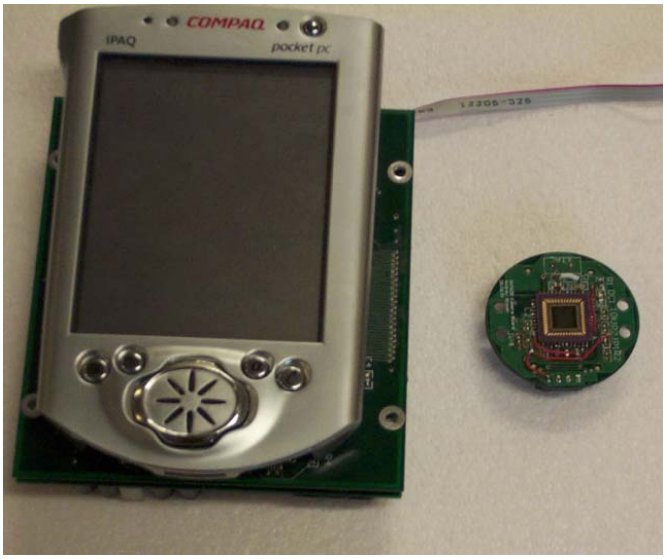

Figure E1.2
Assembled vision
system

E.2 Communication Board

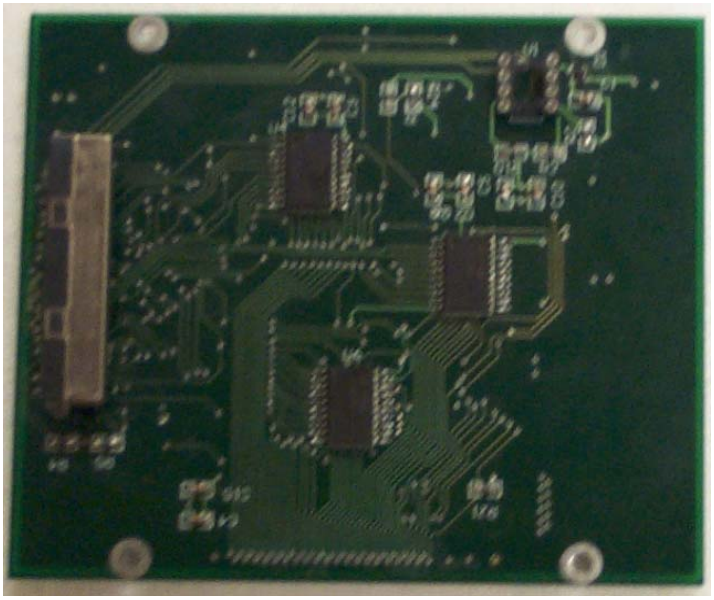

Figure E2.1
PCB bottom view

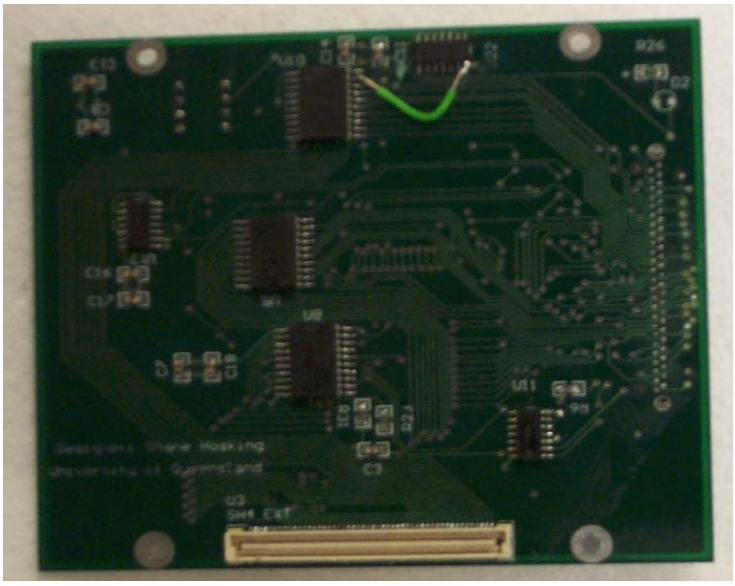

Figure E2.2
PCB Top View

Appendix F - PCB

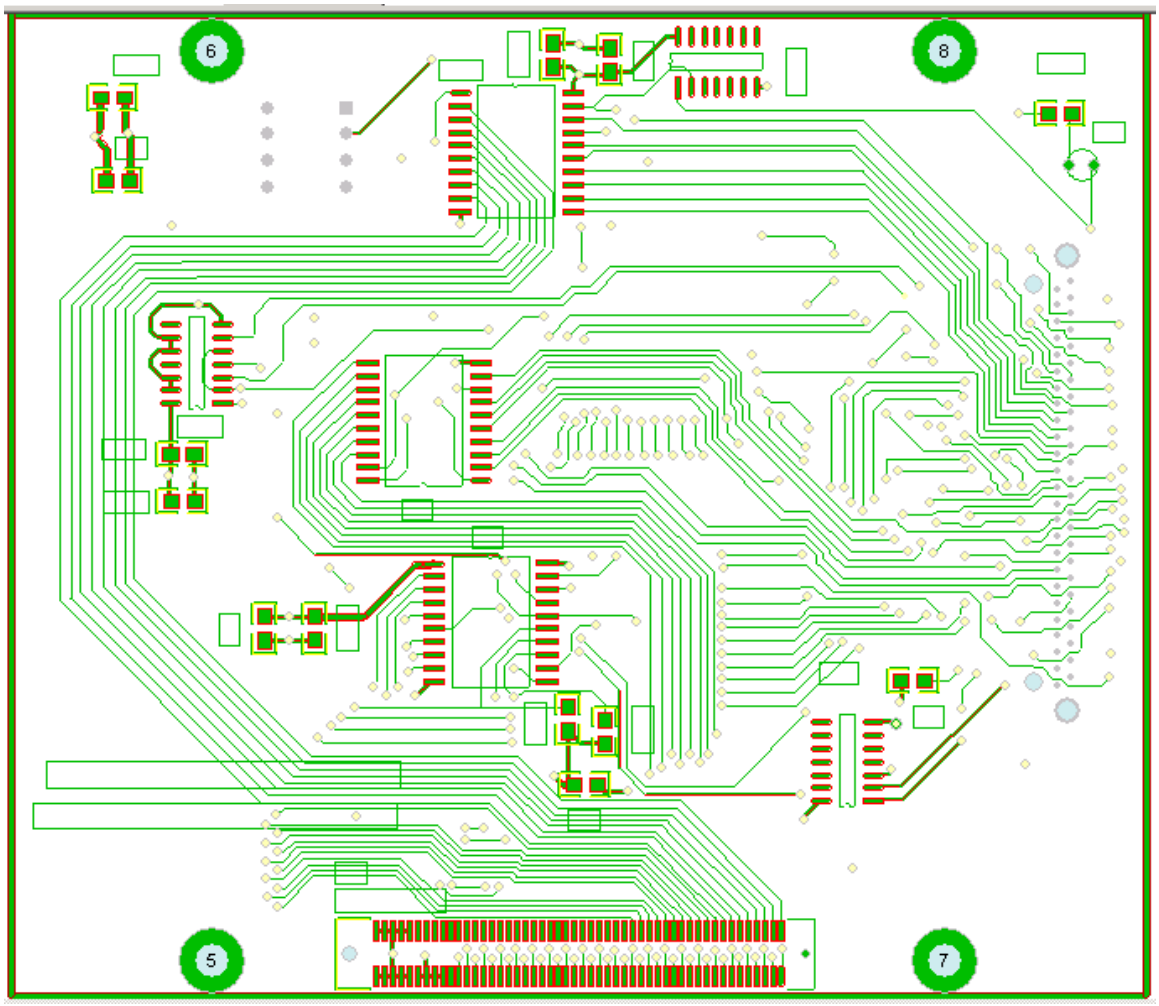

Figure F1.1
Top layer of PCB

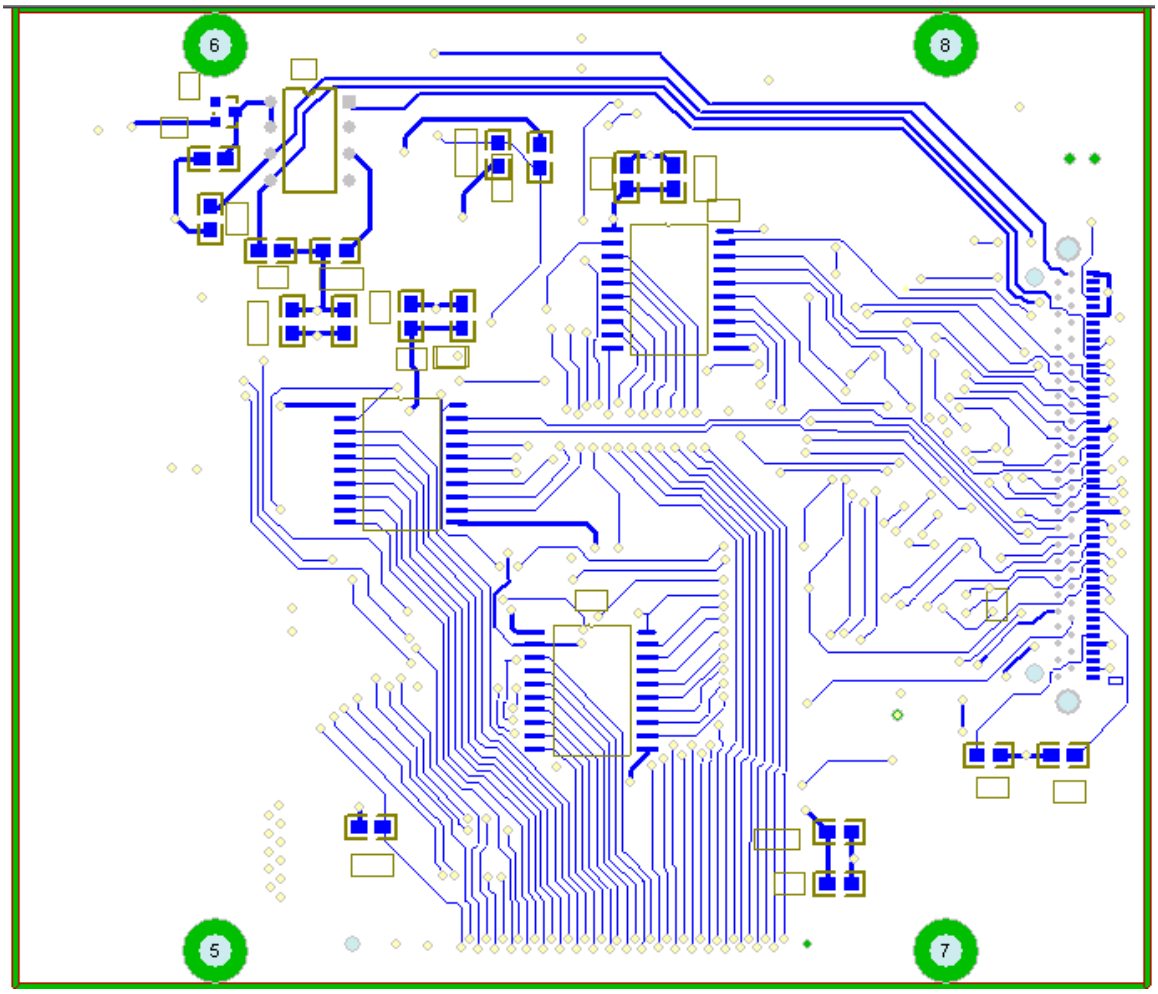

Figure F1.2
Bottom Layer of PCB

Appendix G – IPAQ 100 pin connector

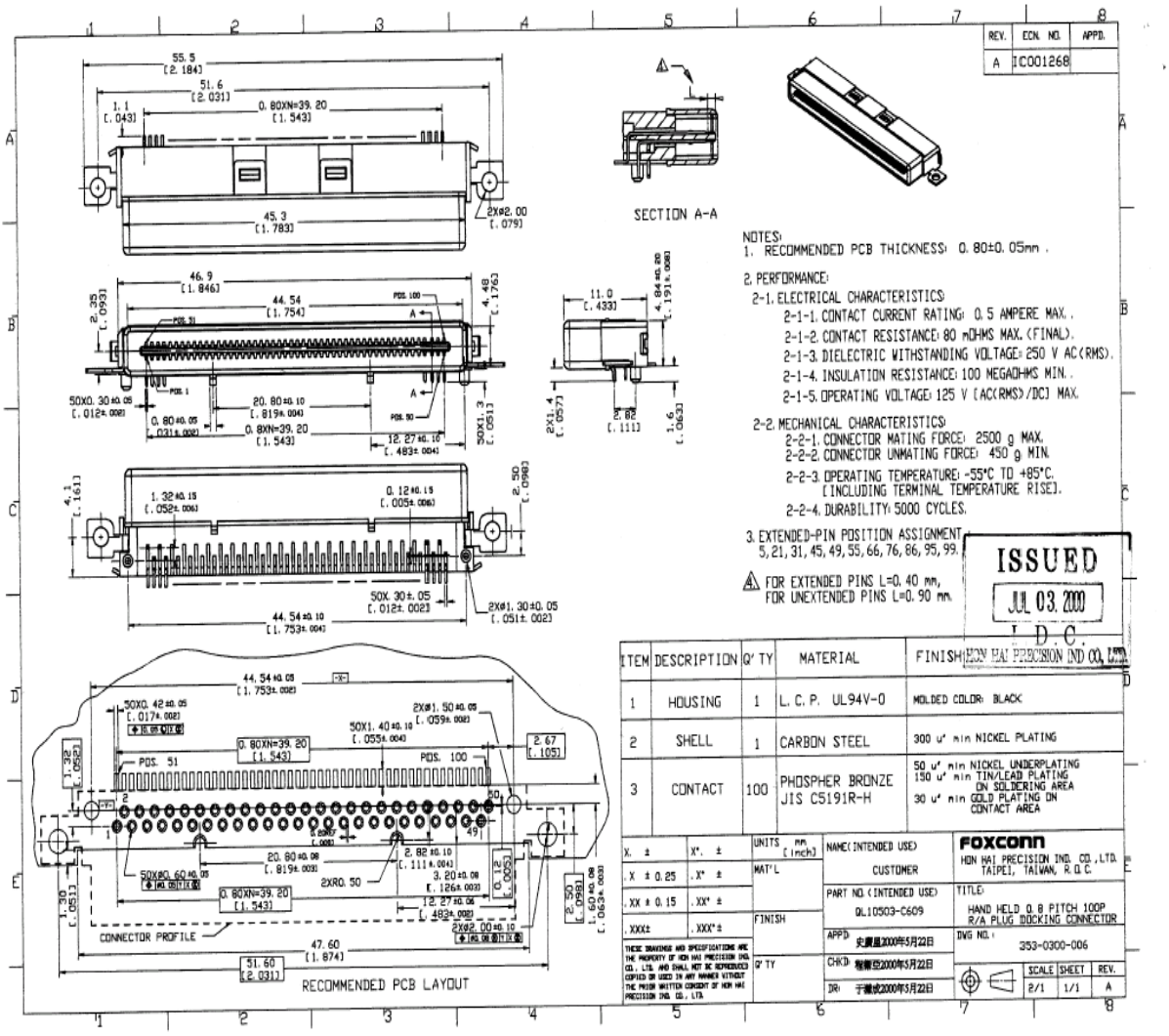

Figure G1.1
IPAQ 100pin connector