A TOASTER WITH A WORLD WIDE WEB INTERFACE

By

NICHOLAS RYAN

Department of Electrical and Computer Engineering,
University of Queensland.

Submitted for the degree of
Bachelor of Engineering (Honours)
in the division of Computer Systems Engineering

October 1997.

[redacted]

[redacted]

20 October, 1997

The Dean
School of Engineering
University of Queensl
St. Lucia, Q 4072

Dear Professor Simmons,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Computer Systems, I present the following thesis entitled "A Toaster with a World Wide Web Interface". This work was performed under the supervision of Mr. Gordon Wyeth.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours faithfully,

NICHOLAS RYAN

ACKNOWLEDGMENTS

Thanks to my supervisor Mr. Gordon Wyeth for excellent supervision and guidance. Thanks also to Elizabeth and my family for their support and encouragement over the last year.

Much of the code in this thesis is based on the XINU operating system developed by Douglas Comer.

ABSTRACT

The use and popularity of the World Wide Web and the new generation of powerful embedded processors together have created the rapidly developing technology of hand-held Web browsers. This thesis uses much of this existing technology in producing a device based on a cutting edge concept – an embedded Web server. Such a device is implemented using an embedded processor, memory, an Ethernet controller and software consisting of an Operating System and a Network Protocol suite. An appliance incorporating this system is able to be connected to, controlled and monitored from anywhere on the globally connected World Wide Web. For demonstration purposes, the developed system is interfaced (rather than embedded) to a toaster. By employing new technology and popular protocols and architectures, the Web server developed in this thesis guarantees ease of use, flexibility and widespread compatibility.

This document outlines in Chapter 1 the motivation for and specific goals of the thesis. An investigation of the background concepts and technologies associated with the project appears in Chapter 2. The development and implementation of the necessary hardware and software is examined in Chapter 3 and Chapter 4 respectively. The progress achieved, future work required and system operation are outlined in Chapter 5. Chapter 6 contains the conclusions drawn from the progress achieved.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CODE LISTINGS

1.0 OVERVIEW

The project that this thesis concerns aims to demonstrate the convergence of two technologies - the World Wide Web and embedded microprocessors. The goal is to develop a simple, cost effective, embedded controller with a Web server interface. This will allow control and communication with the appliance that the controller is embedded into through a standard Web browser. To allow for such communication the controller will be connected to a Local Area Network (LAN). For demonstration purposes the controller will be interfaced to a household toaster. The controller will produce dynamically created Web pages when connected to from a Web browser. These pages will provide the user with information such as whether the toast is up or down as well as allowing control of the heating element.

1.1 Introduction

Use of the Internet has expanded phenomenally in the last few years since the development of the World Wide Web (WWW or Web) and the introduction and subsequent success of graphical Web browsers. This success is based on the simple, intuitive interface and user-level platform independence provided by such browsers. The popularity of the Web has created a new hardware and software market with great commercial potential. Software developers are releasing new 'Web enabled' application software while hardware and telecommunications manufacturers are announcing plans for hand-held Web browsers and pay-terminals. These commercial developments, coupled with the proliferation of PC use around the world and the forecast convergence of television and computer technology, guarantees that the popularity and penetration into society of Web technology will undoubtedly continue and expand.

Embedded microprocessors are another technology that have seen rapidly increasing use recently. This is due to their continuing reduction in cost and size together with increased integration and processing power. Embedded microprocessors allow electronic devices to provide greater functionality and control with an associated cost overhead that is minimal or more commonly, cheaper than pre-microprocessor implementations. The extent to which embedded microprocessor power has increased becomes apparent with the advent of devices that incorporate networking technology

allowing connection to the Internet and World Wide Web. Such devices include hand-held Web browsers, hand-held computers and networked equipment such as printers, routers and hubs with Web front-end control [20]. Such devices represent a convergence of both the Internet and embedded microprocessor technologies.

1.2 Motivation

Besides demonstrating the convergence in technologies mentioned above, there are a number of other motivations for this project. There will be a number of significant differences from related previous and current commercial devices. The aim is to create a relatively *simple* and *cost effective* embedded Web server. This will be achieved by aiming toward a low component count and by providing only simple appliance interfacing. Such an appliance interface would consist of a number of parallel I/O pins. This differs from current embedded Web server technology (discussed in the following chapter) that aims to provide complex interfacing, at an associated cost, to interface to peripherals such as touch sensitive LCD screens, disk drives and wireless communication modules. With a simple and cheap implementation, it is hoped to show the possibility of incorporating such technology at a low add-on cost into a large array of electronic items.

Having a WWW interface allows global connectivity. This may not be important for a toaster, but there are many applications in which this could be a great benefit. Allowing control and monitoring of a device from anywhere in the world presents a new, previously untapped use for the Internet and World Wide Web. The simple, intuitive graphical interface that is used to access a globally connected network of information, may now also be used to control and monitor an appliance located anywhere on the same globally connected network.

1.3 Approach

A simplified system block diagram in Figure 1 forms the basis for the design and planning of the embedded Web server. An embedded microprocessor will interface through a bus to an integrated Network Interface Controller (NIC) as well as some system RAM and an EPROM containing the system software. The microprocessor will interface to an electronic appliance through some parallel I/O pins and to the development computer through a serial port.

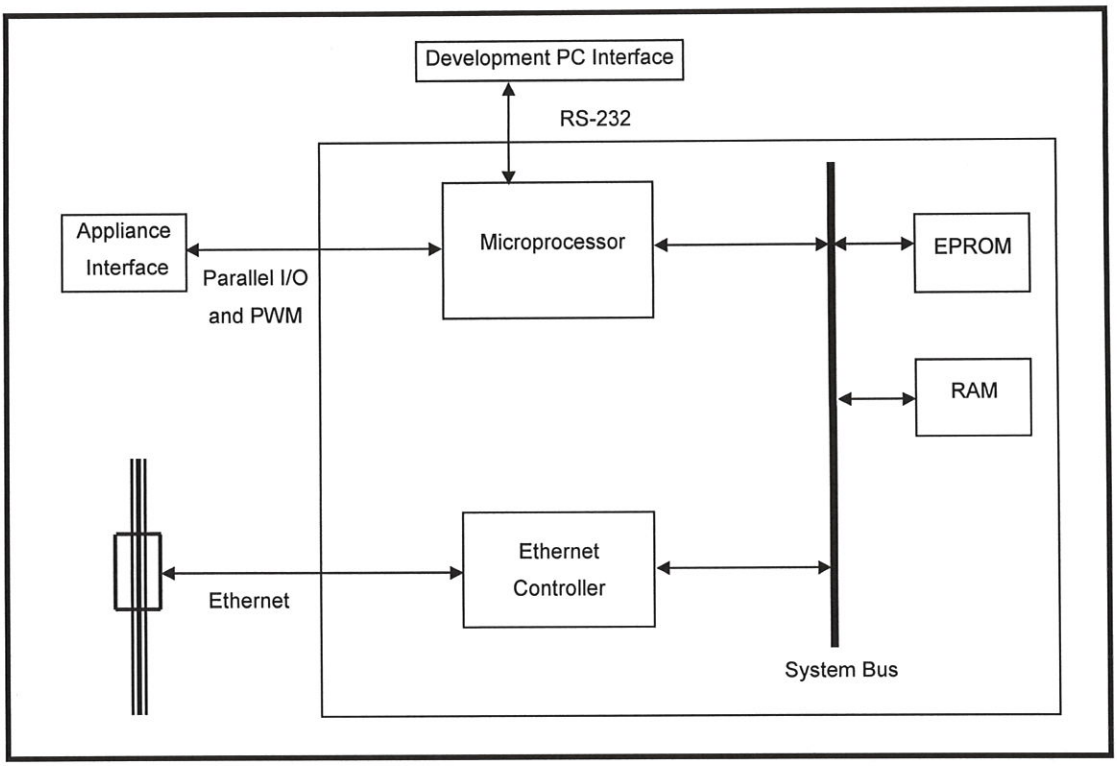


FIGURE 1 : Simplified System Block Diagram.

While the software is being developed, bootstrap code will be required for the microprocessor that runs when the system is reset. This will provide the capability to download code under development from the development computer via the serial interface. This bootstrap code must exist in the non-volatile EPROM seen in Figure 1 so that it is available when the system is reset. Once the software has been developed it may be transferred to an EPROM thus replacing the bootstrap code.

An Operating System (OS) kernel will be written for the processor that will provide simple multi-tasking, interrupt processing, semaphores and process communication capabilities. Due to the specific nature of the project, the OS code will only need to provide enough features to support the networking software and a device driver that will be written to allow control and configuration of the NIC. The device driver will provide a simple interface to higher level software. The code for interfacing to the electrical appliance is to be written into the higher level software as its complexity will be minimal.

The networking software will be based on the Transmission Control Protocol / Internet Protocol (TCP/IP) software that is used on the Internet. (TCP/IP will be

discussed in the next chapter.) An Application Program Interface (API) will need to be written. The API provides an interface between the networking software and the high level application software [25, p.258]. The HyperText Transfer Protocol (HTTP) is an application level protocol that will be used by an HTTP server application [7]. This is the protocol used by the WWW and will allow the system to act as a Web server. As this system has no disk and no virtual file system is to be implemented, the Hypertext Markup Language (HTML) pages use by the WWW will be created and served dynamically.

1.4 Specific Goals

The specific goal of this thesis project is to develop a simple, cost effective, embedded controller with a Web server interface. The basic componenets of the system will be an embedded processor, memory, an Ethernet controller and software consisting of an Operating System and a Network Protocol suite. For demonstration purposes, the developed system will be interfaced to a toaster. Such a system will then enable a device to be controlled by a simple graphical interface viewed on any Web browser running on the same network.

The use of the WWW and the underlying Internet has largely been dominated by information transfer and communication. Few examples exist of the WWW being used as an interface to the real world to control, change or manipulate physical objects. Those that do exist are largely based around a computer connected to a network and a separate real world interface. The creation of an embedded Web Server to interface to electronic devices will show that the possible uses of the Internet encompass far more than just information exchange.

2.0 REVIEW OF RELATED TECHNOLOGY

The concepts related to this thesis and the technologies it uses were thoroughly researched before design of the system began. This allowed sensible design decisions and a clarification of the system requirements. The research results are outlined below.

2.1 Concepts

The concept of the embedded processor lies at the heart of this project. Initially, embedded processors were used as stand-alone devices. However, as the use and complexity of electronic systems increased, often a number of processors were used to control separate sub-systems. In order to reduce replication, wiring overhead and complexity and increase sub-system coordination, networked systems were developed. An example of such an evolution can be seen in the automobile industry where the Controller Area Network [22] protocol was introduced by the Robert Bosch company due to the increasing number of microprocessor controlled sub-systems in automobiles. This standard provides a serial network that allows communication between compatible components such as microprocessor based devices, actuators and sensors, at data rates of up to 1Mbits per second. In 1995 there were 3 million CAN bus systems in vehicles and twice as many in non-automotive industrial applications. Furthermore, the high volume domestic appliance market is expected to increase these figures.

The global communication possibilities provided by this thesis project rely heavily on the pre-existing global inter-network of computers called the 'Internet'. The roots of the Internet lie in the development of an experimental network system in the 1960's and 1970's by the Advanced Research Projects Agency (ARPA) [25, p.198]. The result was the ARPANET which linked military, university and research sites to aid in computer science and military research. During the 1980's the ARPANET was divided so that the ARPANET was separated from the MILNET which was for military use only. It was during this time that the TCP/IP protocol suite (described below) was introduced as a standard. The ARPANET together with the NSFNET, which was created in 1987 to link a number of supercomputer centres, were the basis for what is now commonly known as the Internet. The research results from the

ARPANET project allowed many networks to be incorporated into a single large interconnected network.

Using this technology, the World Wide Web was developed as an application layer protocol that runs on top of the TCP/IP protocol. Its development began in 1989 when its project proposal was written by researchers at the European Laboratory for Particle Physics (CERN) [28]. The first World Wide Web prototype and associated text mode browser were then developed in 1990. This was followed by a public release of the text mode browser in 1992. The popularity of the World Wide Web increased dramatically after September 1993 when the National Centre for Supercomputing Applications (NCSA) released it's Mosaic graphical Web browser for all major computer platforms. Following this, Web browser technology grew quickly as major software companies competed for the potential gains of this technology.

The formal release of SUN's JAVA programming language and compilers in May 1995 signalled a major technological advance as it allowed programs that were compiled into machine independent bytecode to be downloaded and run through a user's Web browser [18]. Today, the latest advance is that of 'push' technology in which information is pushed to a user's Web browser automatically, rather than being retrieved by the user. With such advances in a very small number of years it is certain that the World Wide Web and its associated technologies will continue to become both more complex and more powerful in the future.

Unrelated, until recently, the development of embedded microprocessing technology has been advancing at the same rate as that of the Internet. For several years embedded microprocessors have been used in non-industrial products such as electronic diaries and electronic organisers. The capabilities and uses of such devices were traditionally limited due to the lack of underlying computing power, limited communications support and small storage space. Communication, if any, usually involved storing and retrieving data onto a desktop computer to save space on the organiser. With the popularity and capabilities of the WWW described earlier, the benefits of implementing TCP/IP networking support into such devices became obvious. Unfortunately, the powerful and flexible TCP/IP protocol is complex and requires relatively large amounts of computing resources. It is only recently that

embedded microprocessor power has been able to support the implementation of the TCP/IP protocol.

The latest generation of electronic organisers, now termed Personal Data Assistants (PDAs) provide TCP/IP networking together with features such as LCD touch screens, wireless communications and large memory capacity (in both RAM and ROM). An example of such a device is the well known Apple Newton [1]. The latest model, the MessagePad 2000 may be seen in Figure 2. It provides handwriting recognition, modem, fax, e-mail, limited WWW browsing and communication with a computer through direct or wireless modem, serial or infra-red connections. It also allows connection to an external keyboard and a variety of printers.

FIGURE 2: Apple Newton MessagePad 2000

Another device based on the technologies mentioned above, but more closely related to this thesis project was demonstrated by the German based company 3Soft at a conference in early 1997. A coffee machine had been fitted with 3Soft's embedded Web server hardware and software [17]. This allowed Java programs to be downloaded from the coffee machine and run from a Web browser, enabling control and monitoring from anywhere on the network. This cutting edge demonstration represents a shift of focus on the uses of the WWW which this thesis also aims to demonstrate. Instead of using the WWW to provide a powerful, user-friendly interface to a globally connected network of information, the aim is to use the same powerful, user-friendly interface to control and monitor a device located anywhere on the same globally connected network. This thesis project is based on much of the technology

used in the PDA style devices described above but it is designed for control through an indirect networked front-end. As there are no display or large storage space requirements, it will be possible to create a cheap and simple device that could be embedded in many and varied electronic products at a minimal additional cost. One way to minimise cost is to use highly integrated hardware components. The next section examines the hardware technology with this in mind.

2.2 Hardware

Embedded microprocessors are not only becoming more powerful but are becoming increasingly integrated. Such integration leads to minimal external components requirements. This is ideally suited to emerging embedded applications that are usually characterised by the requirements of low cost, compactness and low power consumption. Until recently most embedded processors had been based around relatively simple, low power processor cores such as the IBM compatible XT and AT processors. However, with more sophisticated manufacturing technology and greater demand the latest generation embedded processors have been based around more powerful processor cores. The Hitachi SH7032 embedded processor chosen for this thesis is an example of such a current generation processor [8, p.2]. It is a high performance RISC processor that is available in two clock speeds : either a 20 Mhz, 5 volt or 12.5 Mhz, 3.3 volt package. The 20Mhz version has a performance rated at 16 Mips. The SH7032 has a five stage pipeline, 32 bit internal architecture and a 16 bit external data path. Integrated into this package are 8Kbyte of internal RAM, a bus state controller that supports direct interfacing to static RAM (SRAM), dynamic RAM (DRAM) and address / data multiplexed buses, a 4 channel Direct Memory Access (DMA) controller and an interrupt controller that controls 9 external interrupts, 31 internal interrupts and provides 16 programmable priority levels. Also provided are a two channel Serial Communications Interface (SCI), a 5 channel 16 bit programmable timer, an 8 channel A/D converter and 40 parallel I/O pins. As can be seen from Figure 3, Hitachi is advertising this chip as today's processor for tomorrow's embedded applications.

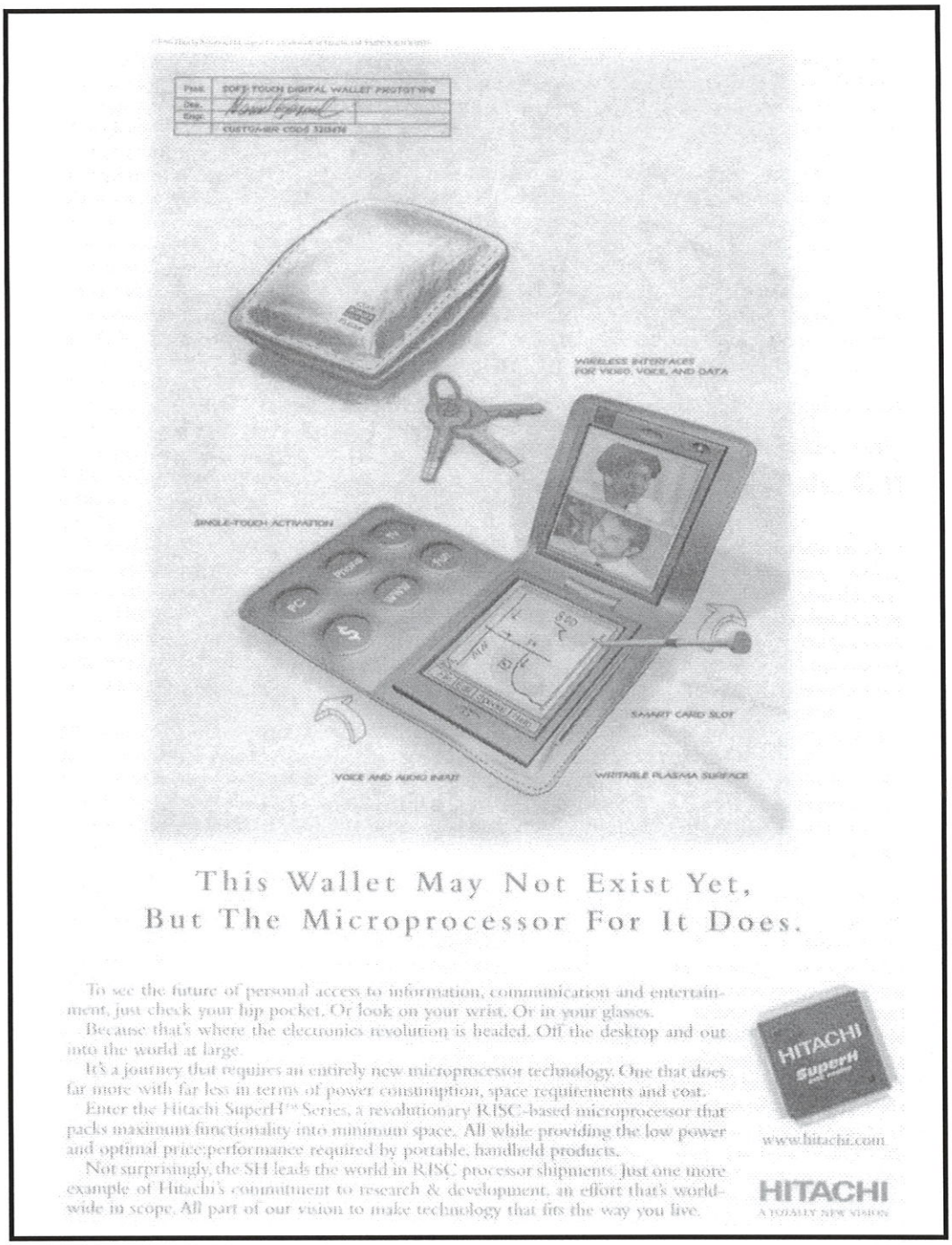


FIGURE 3: Advertisement for Hitachi SuperH processors. (WIRED, March 1997. Page 115)

Memory of some form was required for the system both to store the developed code and to provide variable space and stack space. DRAM Single Inline Memory Modules (SIMMs) are widely used in computers today due to their relatively low cost and large memory sizes. However since the DRAM stores data as capacitive charges, it requires continual refresh cycles [11, p.812]. This is traditionally the disadvantage of using such memory as a separate refresh controller must be implemented. Access time to the DRAM is slower than SRAM as the address input lines are multiplexed [11, p.812].

As mentioned above, the SH7032 supports direct interfacing to DRAM by providing multiplexed row and column memory addresses, as well as providing automatic control and refresh strobes. For this reason a 1Mbyte SIMM is to be used for the main system memory. The standard 30 pin module SIMM that has now been superseded will be used due to their relatively low cost. The pinout for a 30 pin SIMM may be seen in Figure 4. As well as power, data and address pins it is seen that /WE (Write Enable – active low) is used to specify a read or write cycle. /RAS and /CAS (Row Address Strobe and Column Address Strobe – active low) are used to both latch the address lines and to perform refresh of the memory [14]. QP, DP and /CASP are used in the 9 bit (8 data bits + 1 parity bit) parity SIMMS if parity is supported by the microprocessor [26].

30 pin SIMM configurations:

256kbyte x 8
256kbyte x 9 (parity)
1Mbyte x 8
1Mbyte x 9 (parity)
4Mbyte x 8
4Mbyte x 9 (parity)

Pin	Function	Pin	Function
1	Vcc	16	DQ4
2	/CAS	17	A8
3	DQ0	18	A9
4	A0	19	A10
5	A1	20	DQ5
6	DQ1	21	/WE
7	A2	22	GND
8	A3	23	DQ6
9	GND	24	N/C
10	DQ2	25	DQ7
11	A4	26	QP
12	A5	27	/RAS
13	DQ3	28	/CASP
14	A6	29	DP
15	A7	30	Vcc

Note:

QP, /CASP and DP are not connected on all 8 bit modules
A9 is not connected on 256Kbyte modules
A10 is not connected on 256Kbyte and 1Mbyte modules

FIGURE 4: Pinout for 30 pin DRAM SIMM [26].

SRAM will be used to store the code downloaded from the development computer. This is necessary as the DRAM cannot be used until the processor has been initialised by the software. As mentioned earlier once the code has been developed it may be transferred to an EPROM and the SRAM will no longer be required.

A device that is to operate on the Internet must have some form of direct connection to a Local Area Network (LAN). Ethernet (IEEE 802.3 standard) is one of a number of protocols that may be used for the physical connection of nodes on a Local Area Network. Due to its wide use, both throughout the world and within the University of

Queensland engineering department, this protocol will be implemented to allow connectivity with a large amount of pre-existing networks. Ethernet uses Carrier Sense and Multiple Access with Collision Detection (CSMA/CD) which means that all Ethernet transceivers communicate on a single medium, that only one may transmit at a time, and that all may receive simultaneously [21]. If two transceivers attempt to transmit at the same time, a transmit collision is detected, and both devices wait a random period before attempting retransmission. The standard physical connections used with Ethernet today are either 10Base-2 (Thin coaxial cable or 'Cheapernet') or 10Base-T (Universal Twisted Pair). 10Base-2 uses a coaxial cable with BNC connectors. The nodes on a 10Base-T network are daisy-chained together using 'T' junctions and the endpoints of the cable are terminated with an impedance matched load to stop signal reflections. 10-Base-T uses a twisted pair cable to minimize cross-talk with modular RJ45 connectors [21]. Nodes on a 10Base-T network are connected in a star topology with all connections coming from a central hub. An alternative connection is through an Auxiliary Unit Interface (AUI) which is 15 pin 'D' type plug that has signal as well as power connections. This interface may be connected to either 10Base-2 or 10Base-T through an appropriate adapter.

The data transmitted over the Ethernet network is Manchester encoded. Manchester encoding combines a clock signal and Non-Return to Zero (NRZ) serial data into a signal that may be transmitted on a single pair of wires [23, p.3-16]. This means that a separate clock signal is not required to be transmitted. Encoding is done by exclusively-ORing the clock and data. As may be seen in Figure 5 this results in a zero transition in the middle of every bit signal. Upon reception, the clock and data may be accurately decoded using a Phase Locked Loop (PLL).

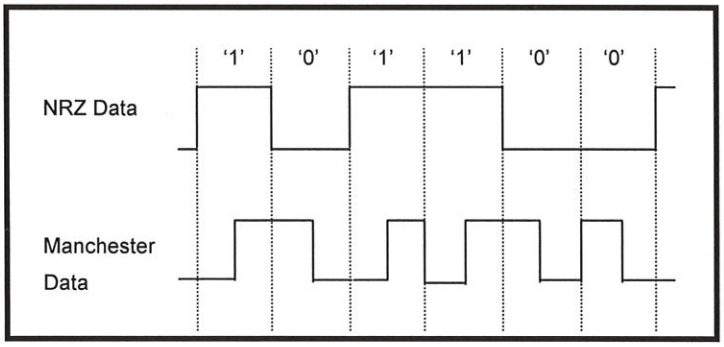


FIGURE 5: Manchester encoding [23, p.3-16].

To allow for this a Manchester encoded preamble field consisting of 62 alternating 1 and 0 bits is prepended to the actual data packet that is to be transferred. Some of these are lost during transmission over the network but the remainder allow the receiving device to achieve bit synchronization.

A standard IEEE 802.3 Ethernet packet may be seen in Figure 6. It consists of the following fields: preamble, Start of Frame Delimiter (SFD), destination address, source address, length, data, and Frame Check Sequence (FCS) [5, p.20]. All fields are of fixed length except for the data field. The 62-bit preamble was described above. The SFD allows the receiving device to determine byte alignment of the incoming data. The Preamble, SFD and FCS fields are generated and appended during transmission. The Preamble and SFD fields are stripped during reception [16, p.12].

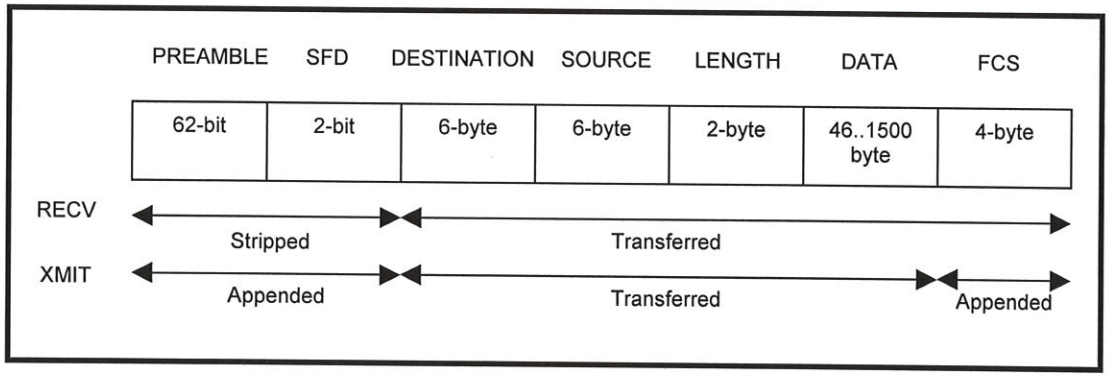


FIGURE 6: Standard IEEE 802.3 Ethernet Packet [5, p.20]

The FCS is passed through to allow for error checking. Every Ethernet controller has its own unique address which consists of 6 bytes. Each Ethernet controller listens for incoming packets with a destination address that matches its own physical address. As well, all controllers listen for the standard broadcast address of "FF-FF-FF-FF-FF-FF". Multicast addressing is similar to broadcast addressing but allows the definition of a subset of all the Ethernet controllers on the network. Thus only Ethernet controllers programmed to listen to a multicast address will do so.

High component integration has also influenced Ethernet controller hardware recently. It has allowed major manufacturers to produce chips that require little external hardware to interface between a processor bus and an external Ethernet connection. Most Ethernet controller chips are designed to interface to standard PC buses. The Ethernet controller chosen for this thesis is the National Semiconductor

DP83902. It is available in an 84 pin PLCC package and provides a complete Ethernet Network Interface Controller (NIC) solution for the popular 10Base-T and 10Base-2 networks [16]. It consists of three main sections which are a Media Access Controller (MAC), a Manchester Encoder / Decoder (ENDEC) Module and an Auxiliary Unit Interface (AUI) module. The MAC provides dual 16-bit DMA channels which are used to transfer packets to and from system and local buffer memory. A 16-byte internal FIFO is use to compensate for timing differences between the network and system data rates. These are both controlled by an integrated, intelligent buffer management system to provide simple and efficient packet transmission and reception. Automatic filtering and recognition of physical, multicast and broadcast addresses are provided as well as three network error counters. The ENDEC performs Manchester encoding of data to be transmitted and Manchester decoding of data received. If the twisted pair transceiver is used, network collisions are detected and reported to other modules in the chip. The AUI and 10Base-2 interfaces both rely on the externally connected transceiver to detect and report collisions. Internal and external loopback modes are also available to test a number of the functions that the chip performs. The AUI differentially drives a transmit wire pair and has differential inputs for receive and collision pairs. This interface is used rather than the direct 10Base-2 or 10Base-T connections provided as it allows connection to either of these networks through an appropriate adaptor.

2.3 Software

An investigation of software to run on control the hardware chosen above was performed. As is to be expected, the power of embedded processors will always lag behind that of desktop computers. As well, data and code storage space is minimal as most embedded processor devices lack disk drives. Thus code developed for such embedded processor based devices must be fast, highly efficient and compact. As mentioned previously, the TCP/IP networking protocol is fairly large and complex and requires a sophisticated operating system for its operation. The OS must provide multi-tasking as each layer of the TCP/IP suite runs as a separate process [6, p.7]. Due to this, process control such as creating, suspending and killing processes must also be implemented. Inter-process communication is used to allow network packets to be transferred between layers of the TCP/IP suite while event timers are relied heavily upon by the TCP layer. Sophisticated memory management is also required to ensure

that one process does not use all the available memory and thus create a deadlock [4, p.231].

A number of moves towards creating operating systems and networking packages that will run on limited resources have been made in the last couple of years. A number of established Operating Systems developed specifically for embedded applications provide compact and efficient code. However, probably the most publicised example of an embedded OS would be Windows CE released by Microsoft in November 1997 [3]. This is a scaled down version of the Windows 95 OS and is designed for use in handheld computing applications. It comes with built in cutdown versions of Microsoft Word and Excel for Windows. Windows CE is used in the Casiopeia [3], a handheld computer produced by Casio which is actually based on a more powerful version of the Hitachi processor used in this thesis.

A better example of compact and efficient software can be seen on a demonstration disk released early in 1997 by QNX Software Systems. This is a standard 3½ inch floppy disk that requires only a 386 compatible processor, 6Mbyte of RAM and *no* hard drive. Once booted from, the disk provides an operating system with a Windows 95 'like' Graphical User Interface (GUI), a WWW browser, a WWW server, a text editor and a virtual file system. Such a demonstration is encouraging as the resource requirements will be significantly less for this thesis as no graphical display, file system or WWW browser are required.

These examples are both commercially developed products. The OS for this thesis is based on a non-commercial product. XINU is an operating system developed by Douglas Comer for an operating system design course in the U.S.A. [4]. Because of this it is readily available, has excellent documentation and a relatively simple source. It is a simple operating system based on the design of Unix, providing all the features mentioned above and has full TCP/IP networking support [6]. Significant amounts of the machine dependent task swapping, interrupt control and memory management will need to be rewritten for the Hitachi processor.

The networking software required for communicating over the Internet runs on top of the OS as higher level processes. The diagram in Figure 7 illustrates the 7-layer Open

Systems Interconnection (OSI) model for networking software that the TCP/IP protocol suite is based upon [25, p.174].

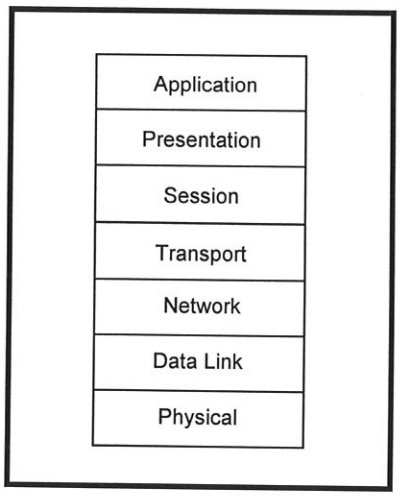


FIGURE 7: Open Systems Interconnection (OSI) model [25, p.174].

Figure 8 shows the simplified layering model used in the TCP/IP protocol suite and its relationship to the standard OSI model [25, p.199].

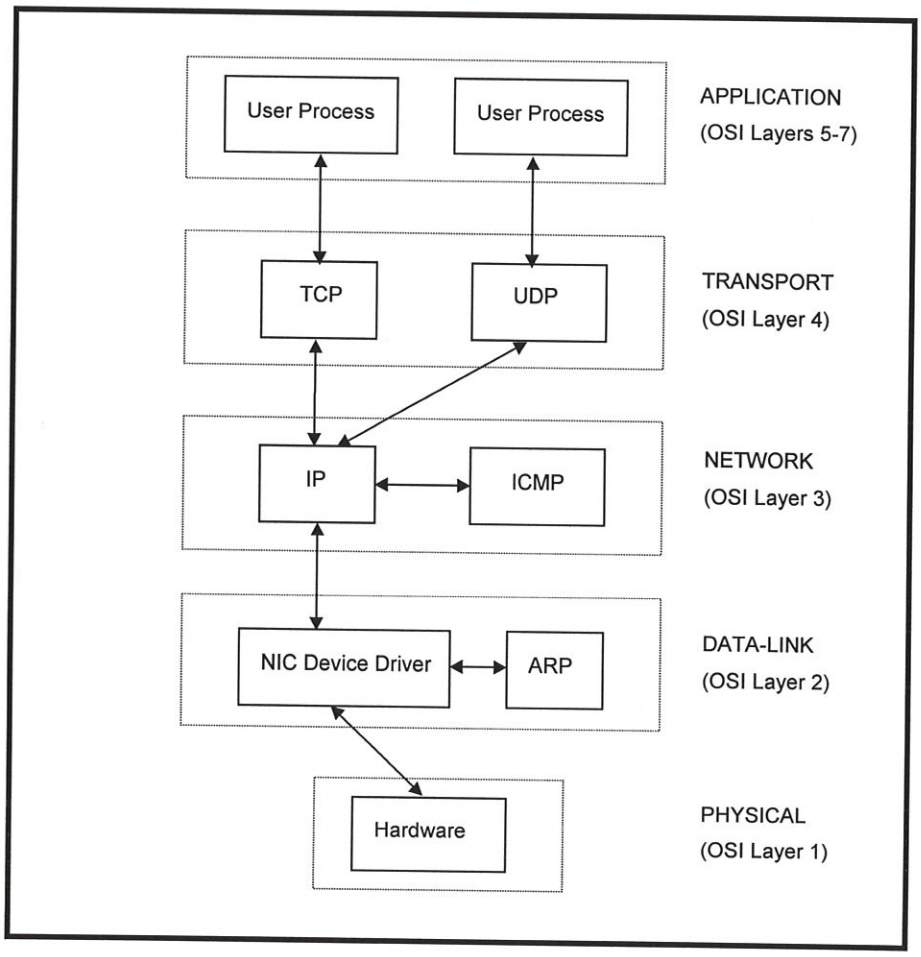


FIGURE 8: Layering used in the TCP/IP protocol suite [25, p.174].

The physical layer consists of the Ethernet controller and connections discussed earlier. The data-link layer consists of an Ethernet device driver which will be specifically written for the DP83902 NIC. A typical NIC device driver consists of initialisation routine, a send packet routine to either transmit a packet on the Ethernet or queue it if one is already being sent and an interrupt routine. The interrupt routine receives packets and transmits any queued packets. It also notifies high level software of both successful and erroneous transmission and reception of packets [15, p.1].

Apart from the hardware specific Ethernet device driver, the Data-Link layer also requires implementation of the Address Resolution Protocol (ARP). ARP provides a mechanism to translate a 4-Byte logical IP addresses into 6-Byte physical Ethernet addresses so that a packet may be sent to the correct network node [24]. This is necessary because IP addresses and Ethernet addresses are selected independently and thus algorithmic translation is impossible. The translation is done only for outgoing IP packets as this is when an IP header and an Ethernet header are prefixed to the packet being sent. ARP keeps a mapping table and controls table look-up and maintenance [6, p.39]. When the ARP table cannot be used to translate an IP address, a request packet with a broadcast Ethernet address is transmitted on the network. Each ARP module running on the network receives the message. If the target IP address of the message matches its own IP address, it sends a response directly to the source Ethernet address with its own Ethernet address. The ARP protocol also specifies that the responding module must store the source addresses of the request packet in its own table [6, p.53]. Packets that are queued while waiting for an address translation are automatically sent by ARP once the translation response has been received.

The Internet Protocol (IP) is used to implement the Network layer. IP provides an unreliable delivery of IP packets – it does not guarantee that packets are delivered or delivered correctly [24]. Reliability must therefore be provided by higher level protocols. IP is also connectionless meaning that IP datagrams are transmitted independently and must therefore contain all the information required for delivery including destination and source IP addresses. An IP message may be forwarded from one physical network to another by a gateway that it is connected to two physical networks which has a NIC for each network. This allows the IP layer and its associated IP addresses to build a single logical network from multiple physical

networks - IP hides the underlying network structure from higher level applications. This interconnection between physical networks is the source of the term 'internet' [24]. IP maintains a routing table which is used to determine how to route packets around an internet [6, Ch.6]. It provides information such as the shortest route to a given address and which gateway is connected to which network. IP also provides for fragmentation and re-assembly of long higher level datagrams into smaller IP packets [6, Ch.7]. The Internet Control Message Protocol (ICMP) is also implemented in the Network Layer and allows error messages and routing changes for the IP process to be transmitted over the network [6, Ch.8].

For this project, as the device will not be acting as a gateway, the IP layer can be greatly simplified [5, p.96]. A routing table is not required as the destination address will either be to a node on the local network or a gateway on the local network. As well, this allows the ICMP process to be simplified so that the only task it performs is to provide replies when requested [5, p.103]. At this stage of development, a Packet InterNet Groper (PING) client can be used to send an ICMP echo message request to a host and wait for a reply [25, Ch.11]. Thus the reachability of the node on the network and the functioning of the IP and ICMP processes may be tested.

Either the User Datagram Protocol (UDP) or the Transmission Control Protocol (TCP) may be used to implement the Transport layer [25, p.199]. UDP is simpler to implement but is not used by the WWW. The more complex TCP provides a connection-oriented, reliable, full-duplex byte-stream service [13]. Connection-oriented means that when two applications are communicating using TCP, a logical connection between them exists. Thus TCP modules hold state information that is used to define a full-duplex virtual circuit. TCP uses a sliding window system to guarantee reliability. This means that outgoing data must be acknowledged by the receiving end within a window size which is specified using the number of transmitted bytes. As TCP is a byte-stream service: bytes rather than assembled packets of fixed length are transferred. Each end of the TCP full-duplex, virtual circuit may exert flow-control to prevent buffer overruns. TCP also implements port numbers that are specified when establishing a connection. This allows the TCP process to be multiplexed between a number of client processes on the one system.

An Application Program Interface (API) to the TCP/IP communication protocol provides an interface to applications for network communication [25, p.258]. The interface consists of functions providing connection establishment, data transfer and disconnection by simply specifying connection type, IP addresses and TCP ports. The TCP/IP protocol suite and the API are to be based on the implementation provided with the XINU operating system [6, Ch.17].

The HyperText Transfer Protocol (HTTP) is used in the application layer and communicates with the TCP/IP software through the API described above. This is the protocol used by the WWW since 1990 [7]. The HTTP protocol is based on a request / response paradigm in which a client has available a set of methods with which to indicate the purpose of a server request. It uses Uniform Resource Identifiers (URI) to identify resources with which the request method is concerned. A client request consists of a request method, URI, and protocol version, followed by a message containing request modifiers, client information, and possible body content. A server response consists of a status line and a message containing server information and possible body content. HTTP requires a reliable transport mechanism such that is provided by TCP/IP. The HTTP software needs only to consist of a simple text parser to handle incoming requests and a text generator to dynamically produce the server response which will contain an HTML page as the body content.

With research and investigation of all areas of the project completed and the main system components chosen it was now possible to start developing the system hardware. The next chapter examines the hardware implementation used for this system.

3.0 HARDWARE IMPLEMENTATION

This chapter examines the design of the system hardware. Figure 9 shows a block diagram of the developed system.

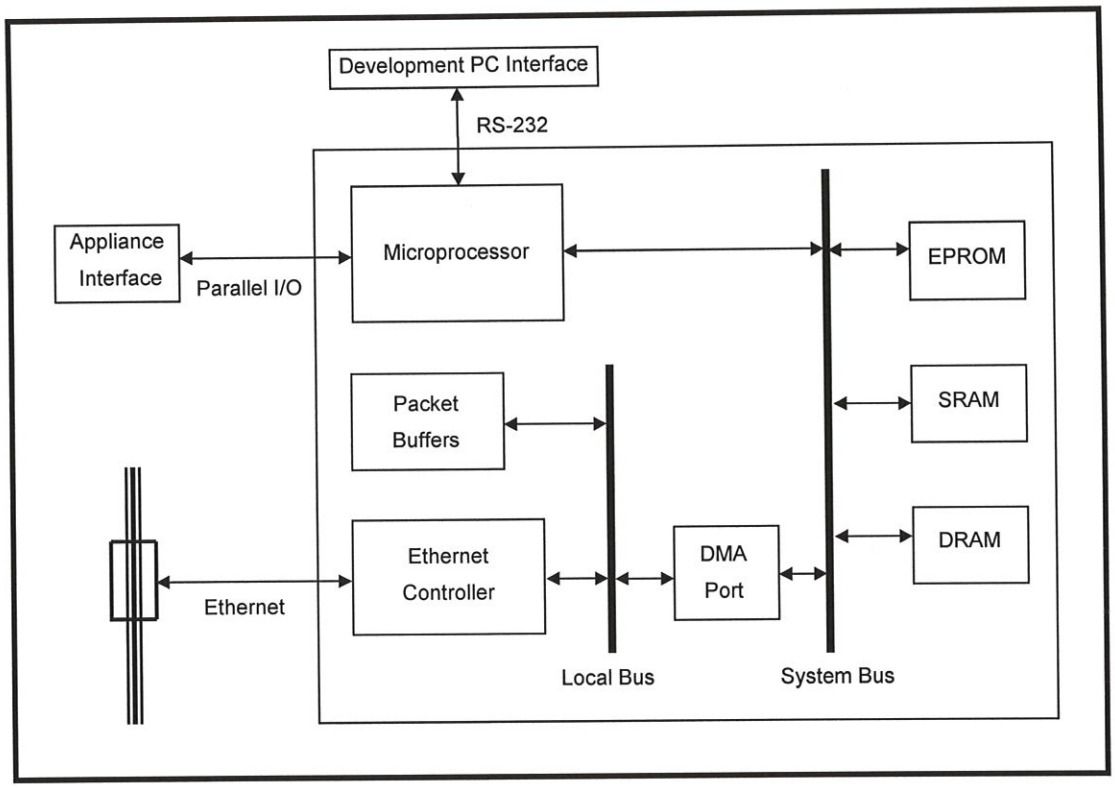


FIGURE 9: Complete system block diagram.

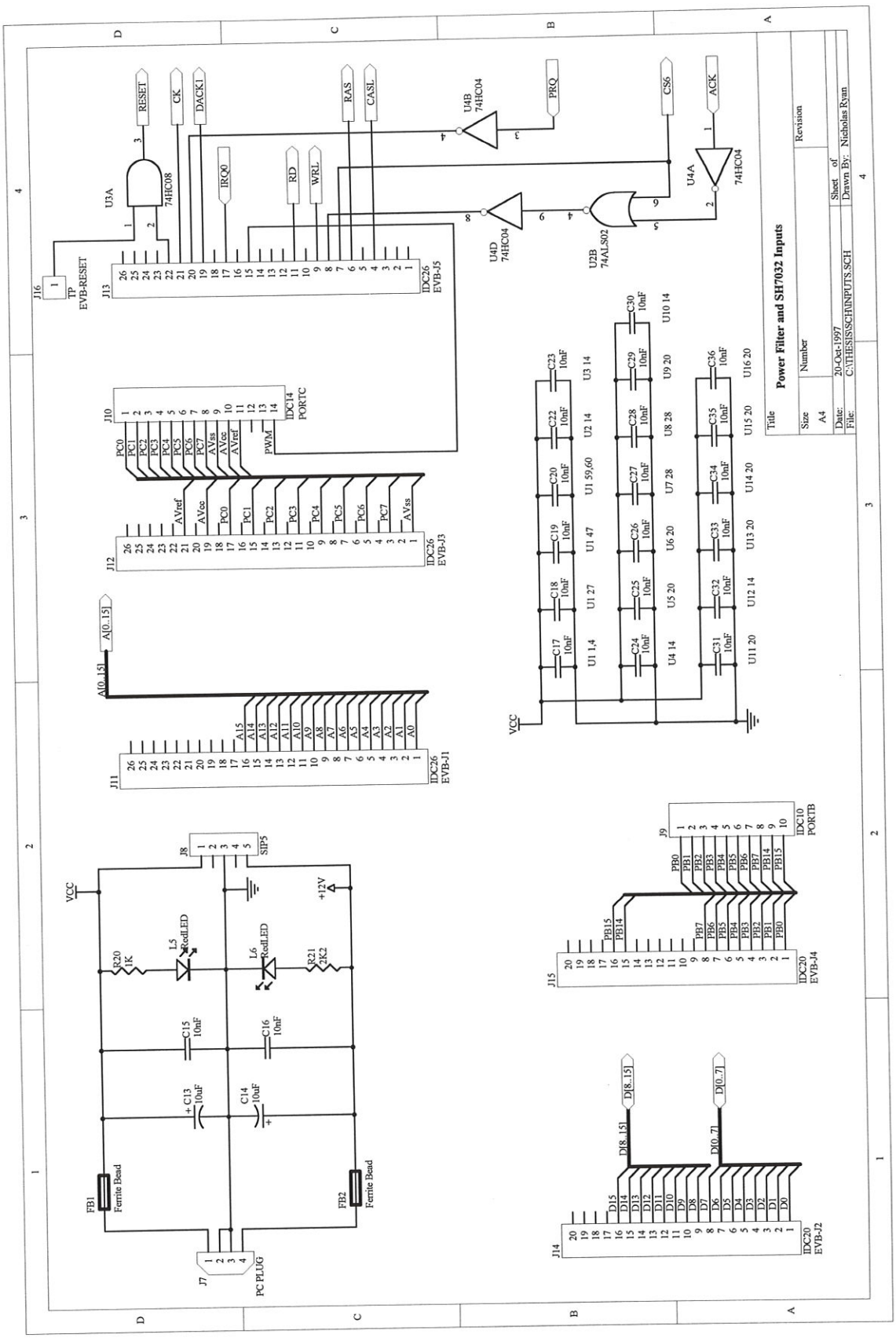
3.1 Microprocessor

An evaluation board for the SH7032 microprocessor was purchased from Hitachi. Based on the SH7032, this board consists of an EPROM, RAM, two serial ports, a reset and a Non-Maskable Interrupt (NMI) pushbutton, a simple power supply and access to all of the processor pins. The EPROM contains startup code and a serial communications and debugging package. The RAM consists of two 32Kbyte x 8 SRAMs that can be accessed simultaneously to allow for 16 bit data transfers. The serial ports consist of two 9 pin male 'D' type connectors. One port is used for communication with the development computer, while the other is available to the user. The ports are interfaced using a MAX-232 serial transceiver that translates RS-232 signals to logic levels and vice-versa. The reset pushbutton uses a Dallas Semiconductor DS1233A reset generator for debouncing and to provide a 350mS reset signal to the processor. The NMI pushbutton is debounced using two cross-coupled AND gates and an SR flip-flop.

A regulated 5 volt supply is required for the evaluation board. The power smoothing components on the evaluation board are only sufficient for driving that board and thus a separate power filter was required for the thesis project board that was to be constructed. The thesis project board also required a regulated 12 volt supply for output on the AUI. It was decided that the thesis board would be built to the same dimensions as the evaluation board (5inch x 5½ inch) so that they could be piggybacked together. The reset signal and all processor signals were connected to the thesis board through a number of segments of flat ribbon cable and IDC connectors. A number of parallel I/O pins and a Pulse Width Modulation (PWM) output were made externally available for interfacing to the toaster (or any appliance) at a later date. These connections may be seen in the *Power Filter and SH7032 Inputs* schematic overpage.

3.2 Memory

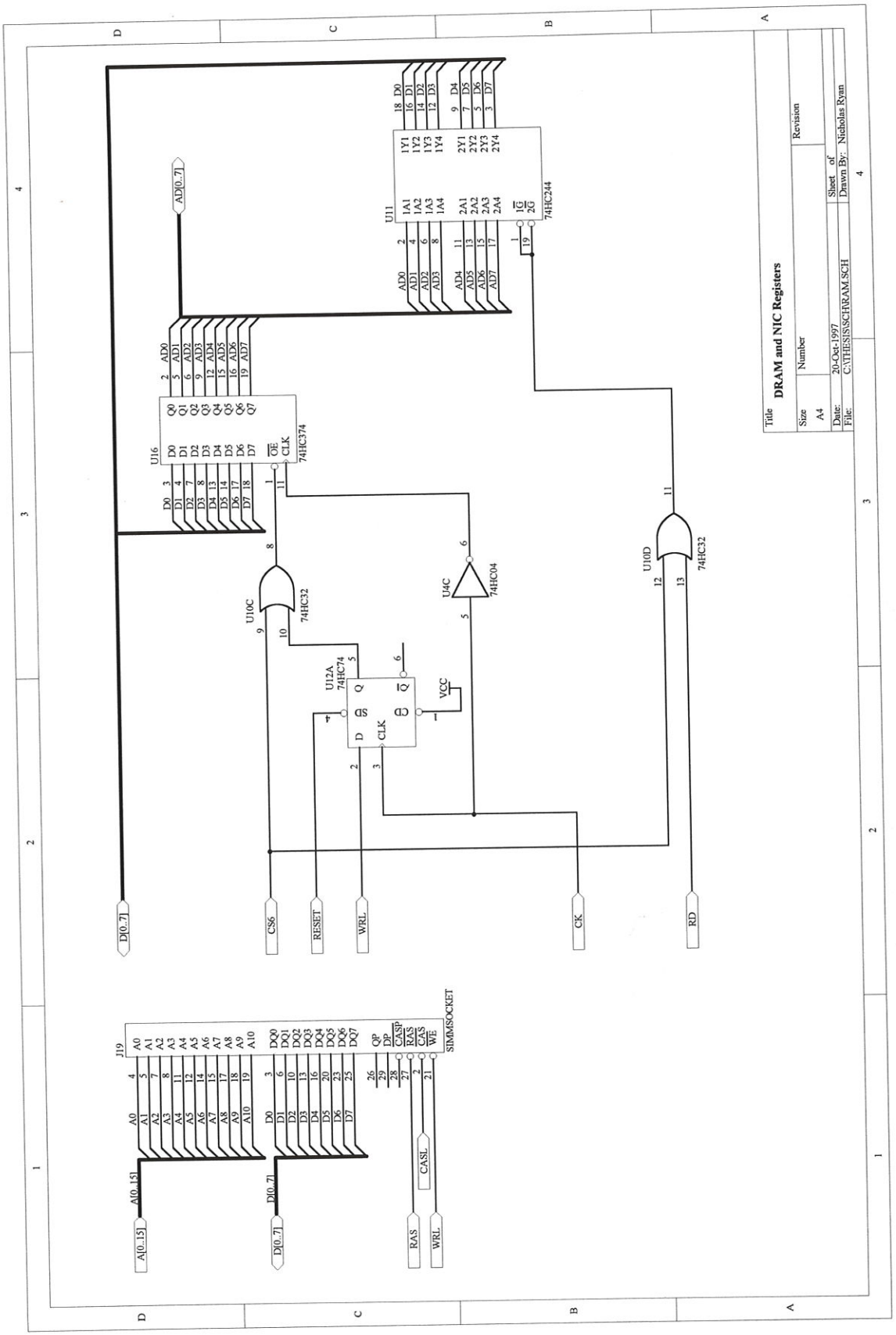
The DRAM implementation was the simplest part of the hardware design due to the SH7032's direct DRAM support. Address and data lines were connected directly as well as the refresh control lines and write enable line. Whilst it is possible to use parity checked DRAM with the SH7032, it is not necessary, and thus the three parity checking lines QP, /CASP and DP (See Figure 4) were not connected. This decision allows either a parity or non-parity DRAM SIMM to be used. The addressing of a DRAM is performed by strobing in the multiplexed row and column addresses by using the /RAS and /CAS strobe signals. A read or write cycle is performed by strobing the /RAS signal first with the row address as input, followed by strobing the /CAS signal with the column address as input. The read or write cycle is determined by the state of the /W signal. As mentioned earlier, the DRAM requires refreshing. A row in the RAM is refreshed whenever accessed, so to perform refresh each row in the RAM must be accessed within the minimum specified time. A 70nS 1Mbyte DRAM SIMM was used for this thesis. Bits in this DRAM required refresh every 8mS, thus a row refresh was required every 15.6uS (512 rows) [14, p.7-36].


Power Filter and SH7032 Inputs
Title
Size
A4
Number
Revision
Date: 20-Oct-1997
File: C:\THESIS\SCH\INPUTS.SCH
Sheet of
Drawn By: Nicholas Ryan
RESET
CK
DACK1
IRQ0
RD
WRL
RAS
CASL
PRQ
CS6
ACK
U3A
74HC08
U4B
74HC04
U4D
74HC04
U2B
74ALS02
U4A
74HC04
J16
TP
EVB-RESET
J13
IDC26
EVB-J5
J10
IDC14
PORTC
PWM
J12
IDC26
EVB-J3
A[0..15]
A[0..15]
J11
IDC26
EVB-J1
VCC
C17 10nF
C18 10nF
C19 10nF
C20 10nF
C22 10nF
C23 10nF
C24 10nF
C25 10nF
C26 10nF
C27 10nF
C28 10nF
C29 10nF
C30 10nF
C31 10nF
C32 10nF
C33 10nF
C34 10nF
C35 10nF
C36 10nF
J9
IDC10
PORTB
J15
IDC20
EVB-J4
D[8..15]
D[0..7]
J14
IDC20
EVB-J2
J8
SIP5
R20 1K
L5 RedLED
L6 RedLED
R21 2K2
+12V
C15 10nF
C16 10nF
C13 10uF
C14 10uF
FB1
Ferrite Bead
FB2
Ferrite Bead
J7
PC PLUG

This project uses /CAS-before-/RAS refresh in which /CAS is strobed before /RAS. This increments an internal counter on the DRAM which generates the row address to be refreshed [14, p.7-50]. The connections to the DRAM may be seen in the *DRAM and NIC registers* schematic overpage. The timing diagrams for a DRAM read cycle, write cycle and refresh cycle may be seen in Appendix B. The programming of the DRAM control on the SH7032 will be discussed in the following section.

The two 32Kbyte SRAM devices on the evaluation board were replaced by two 128Kbyte SRAM devices halfway through the project. This was done in anticipation of a large increase in the prototype code size as the TCP/IP protocols were implemented [6, p.549]. As discussed earlier however, once the software has been written and completely tested it may be transferred to EPROM and thus only EPROM and DRAM would be required.

The SH7032 has an on-board Bus State Controller (BSC) that divides the 32 bit address space into eight areas (0 – 7) [8, p.93]. This allows the data, address and control lines for each area to be configured independently allowing a different type of memory access for each region. Area 7 is used to access the internal RAM whilst areas 0 – 6 are external and have the associated /CS0 - /CS6 enable lines available for use. Only area 1 may be used to interface to DRAM and only area 6 is able to perform address / data multiplexing. All addresses are 32 bits, although bits 28 - 31 are actually ignored and not output externally. Bit 27 of an address is used to determine whether the data should be accessed with an 8 (Bit 27 = 0) or 16 (Bit 27 = 1) bit datapath. Address bits 24 - 26 are used to select one of the eight memory areas and are used as output to the /CS0 - /CS6 lines. Address bits A0 – A21 are available as output from the chip. This allows 4Mbyte of physically addressable space in each area. Bits 22 and 23 are used as physical address lines only when the address is multiplexed for access in the DRAM area. This allows up to 16Mbyte of DRAM to be used. In the other areas these bits provide shadow addresses for the physical 4Mbyte available. These shadow areas provide a logical 16Mbyte address space. On the evaluation board, the EPROM is set up to use memory area 0 whilst the SRAM uses memory area 2. On the thesis project board, the DRAM uses memory area 1 whilst the control registers of the NIC (discussed later) use memory area 6. Figure 10 shows the system memory map for this thesis project.


Title
DRAM and NIC Registers
Size
A4
Number
Revision
Date: 20-Oct-1997
Sheet of
File: C:\THESIS\SCHRAM.SCH
Drawn By: Nicholas Ryan
U16
74HC374
U11
74HC244
U10C
74HC32
U10D
74HC32
U4C
74HC04
U12A
74HC74
J19
SIMMSOCKET
AD[0..7]
D[0..7]
A[0..15]
CS6
RESET
WRL
CK
RD
RAS
CASL
VCC

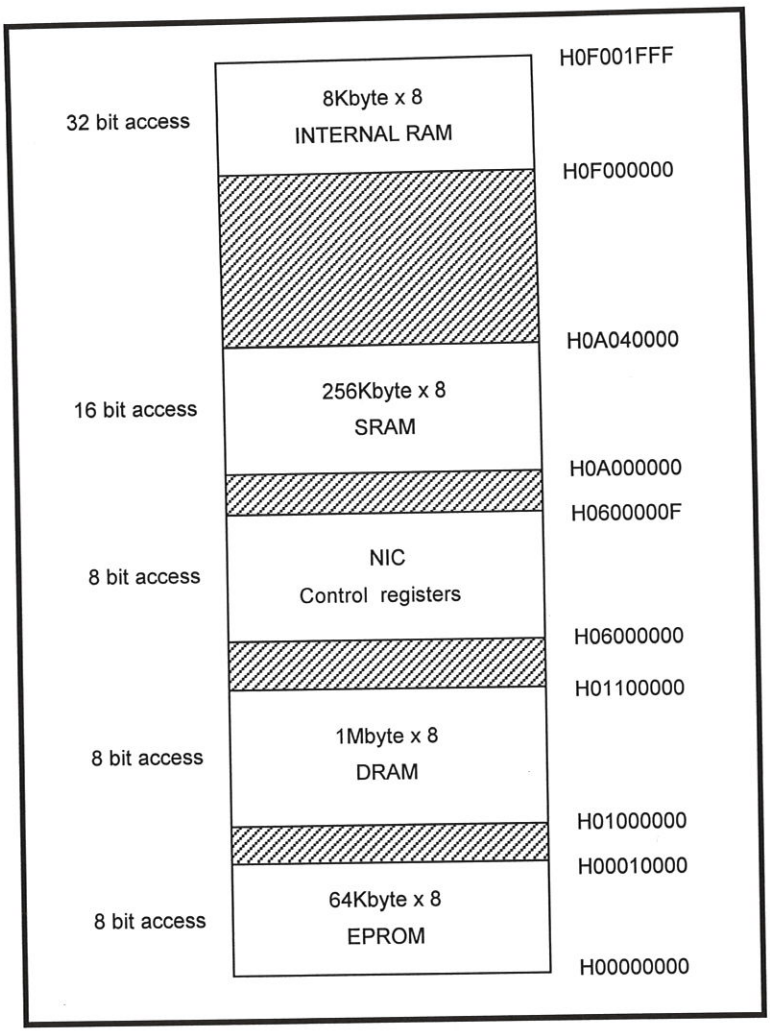
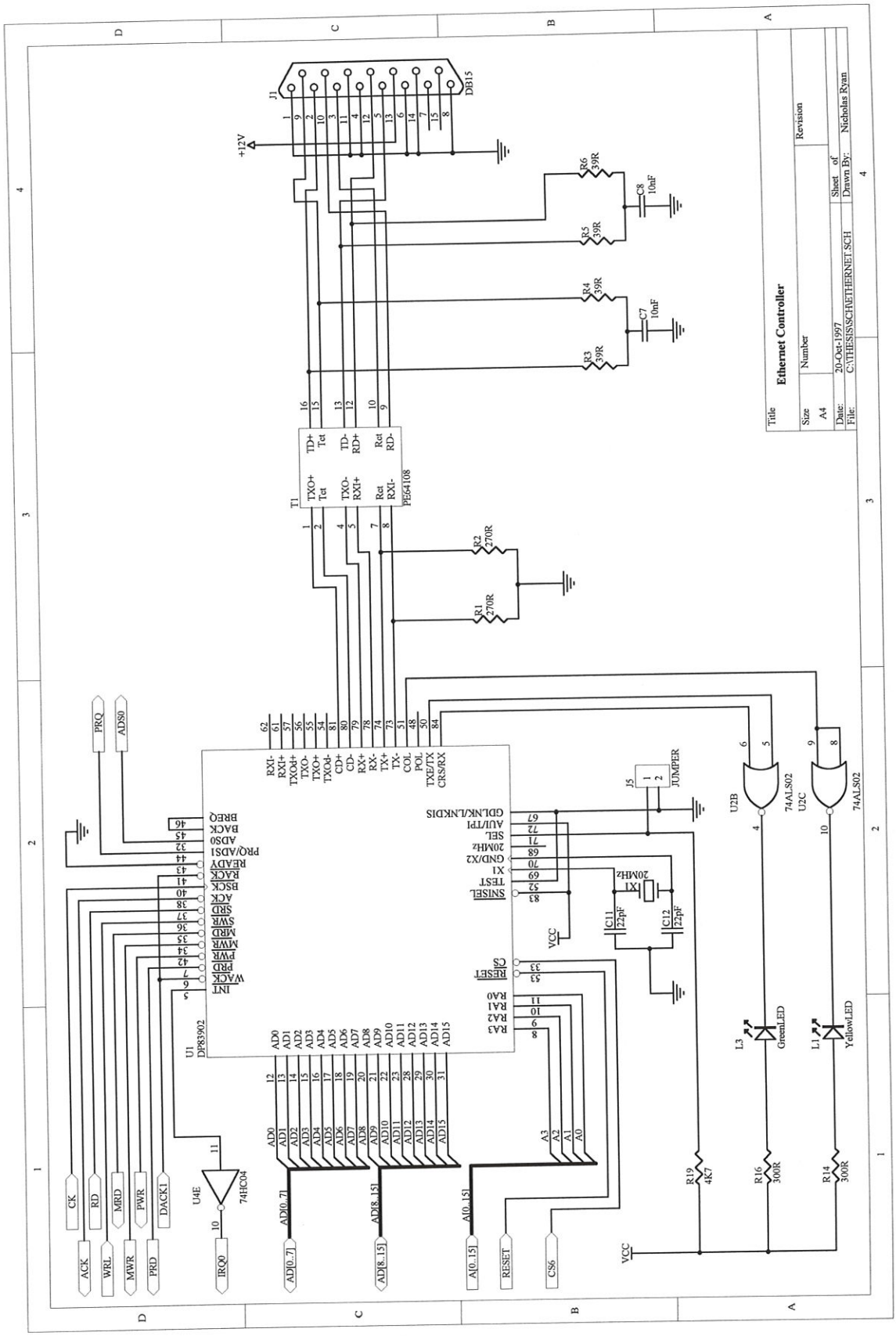


FIGURE 10: System Memory Map.

3.3 Ethernet Controller

Choosing the AUI network interface of the DP83902 Network Interface Controller meant that only a small number of components were required for the network side of the project board. When using the AUI, the external 10Base-2 or 10Base-T adaptor provides collision detection, thus three signal pairs connect the DP83902 to the AUI plug. These are: TX±, RX± and CD±. A pulse transformer is used to provide isolation on these three pairs between the controller and the network connection. The DP83902A connections may be seen in the *Ethernet Controller* schematic overpage.

The DP83902 provides three outputs to signal network collisions, data transmission and data reception. These provide a 50mS active high TTL signal [16, p.6]. These were connected so that a green LED signalled data activity (reception or transmission) and a yellow LED signalled a collision.


Title
Ethernet Controller
Size
A4
Number
Revision
Date:
20-Oct-1997
Sheet of
File:
C:\THESIS\SCH\ETHERNET.SCH
Drawn By:
Nicholas Ryan
+12V
DB15
R6 39R
R5 39R
R4 39R
R3 39R
C8 10nF
C7 10nF
R2 270R
R1 270R
T1
PE64108
U1
DP83902
U4E
74HC04
U2B
U2C
74ALS02
J5
JUMPER
20MHz
C11 22pF
C12 22pF
L3
GreenLED
L1
YellowLED
R19 4K7
R16 300R
R14 300R
VCC
CK
RD
MRD
PWR
DACK1
ACK
WRL
MWR
PRD
IRQ0
AD[0..7]
AD[8..15]
A[0..15]
RESET
CS6
PRQ
ADS0

The DP83902A uses an external 20.00000MHz oscillator connected between X1 and X2 [16, p.6]. This is used for external network timing. The processor system clock signal, CK is connected to the DP83902 BSCK input. This is used for timing when performing data transfers with the system bus [16, p.5]. As mentioned earlier, the reset signal from the evaluation board was used on the project board. A wire from the reset output pin (Pin 2) of the DS1233A reset generator was connected to the /RESET input of the DP83902A.

The DP83902A has three pages of registers that are all 16 bytes in size. Only pages 0 and 1 are used in operation as page 2 is used for factory testing [16, p.25]. The pages are selected by bits in the command register which is at relative address 0 on each page. The registers in pages 0 and 1 may be seen in Table 1.

TABLE 1: Page 0 and 1 DP83902A Control Registers [16, pp.24..25]

PAGE 0		
RA0 – RA3	**RD**	**WR**
H00	Command	Command
H01	Current Local DMA Address 0	Page Start
H02	Current Local DMA Address 1	Page Stop
H03	Boundary Pointer	Boundary Pointer
H04	Transmit Status	Transmit Page Start Address
H05	Number of Collisions	Transmit Byte Count 0
H06	FIFO	Transmit Byte Count 1
H07	Interrupt Status	Interrupt Status
H08	Current Remote DMA Address 0	Remote Start Address 0
H09	Current Remote DMA Address 1	Emote Start Address 1
H0A	Reserved	Remote Byte Count 0
H0B	Reserved	Remote Byte Count 1
H0C	Receive Status	Receive Configuration
H0D	Tally Counter 0	Transmit Configuration
H0E	Tally Counter 1	Data Configuration
H0F	Tally Counter 2	Interrupt Mask

PAGE 1		
RA0 - RA3	**RD**	**WR**
H00	Command	Command
H01	Physical Address 0	Physical Address 0
H02	Physical Address 1	Physical Address 1
H03	Physical Address 2	Physical Address 2
H04	Physical Address 3	Physical Address 3
H05	Physical Address 4	Physical Address 4
H06	Physical Address 5	Physical Address 5
H07	Current Page	Current Page
H08	Multicast Address 0	Multicast Address 0
H09	Multicast Address 1	Multicast Address 1
H0A	Multicast Address 2	Multicast Address 2
H0B	Multicast Address 3	Multicast Address 3
H0C	Multicast Address 4	Multicast Address 4
H0D	Multicast Address 5	Multicast Address 5
H0E	Multicast Address 6	Multicast Address 6
H0F	Multicast Address 7	Multicast Address 7

It was decided to use area 6 of the SH7032 memory space to interface to the DP83902A control registers. To achieve this, the processor /CS6 signal was connected to the DP83902 /CS input and the lower four address bus lines were connected to RA0..RA3. When the /CS line goes low, the DP83902A enters slave mode and its control registers may be accessed [16, p.5]. The /RD and /WRL processor signals are connected to the slave read, /SRD and slave write, /SWR inputs respectively. As only the lower four address lines are used and memory area 6 is accessed with an 8 bit datapath, the addresses of the control registers are H06000000 to H0600000F.

Unfortunately, when writing to the DP83902A control registers, the data hold time of the SH7032 is not as long as the DP83902A requires. (The SH7032 provides a minimum 0nS data hold time.) This means that the data bus could not be connected directly to the DP83902A address / data lines. To solve this problem, the circuit seen in the *DRAM and NIC registers* schematic (shown previously) was used. Here, the bus clock signal is inverted and used to latch the write data into the 74HC374 octal D-type flip-flop. The /WRL signal is used to control the output of the octal flip-flop so that data is output on the address / data bus only when required. However, the /WRL signal is first latched through a D-type flip-flop before connecting to the 74HC374 /OE input. This causes the output of the data byte to be delayed slightly and thus meet the hold time requirements of the DP83902A. To allow the control registers to be read using this configuration, a 74HC244 octal buffer with tri-state output was used as seen in the *DRAM and NIC registers* schematic (shown previously). The /RD signal is used to enable the output from the DP83902A onto the system data bus. Note that both the /WRL and /RD signals in this circuit have been OR'ed with /CS6 so that the 74HC374 and 74HC244 are only enabled onto the local and system data buses respectively when access to the DP83902A registers is taking place.

The /ACK output from the DP83902A is used to signal the processor that it may continue with a control register read or write operation. This signal needed to be inverted before connecting to the SH7032 /WAIT pin. It was also OR'ed with the /CS6 signal to ensure it would only be used when access to the DP83902A registers was taking place. The implementation for this, seen in the *Power Filter and SH7032 Inputs* schematic (shown previously), was more complex than it could have been, as it

was implemented after the thesis board was built and existing spare logic gates were used.

For each memory area, monitoring of the /WAIT signal can be enabled or disabled [8, p.93]. For area 6, used by the DP83902A, the SH7032 was configured to monitor the /WAIT signal during a read or write cycle, allowing the insertion of wait states until the DP83902A was ready.

The active high INT signal from the DP83902A is asserted whenever the reception or transmission of an Ethernet packet is completed [16, p.4]. This signal was inverted and connected to the /IRQ0 SH7032 input. By doing this, an Interrupt Service Routine (ISR) for Interrupt Request 0 (IRQ0) could be used to handle the software requirements of reception and transmission of packets. This ISR written to do this will be discussed later.

The Ethernet controller's address and data buses may be interfaced to the system buses in a number of ways. The ideal method for high performance is a Bus Master Architecture seen in Figure 11 [27, p.4].

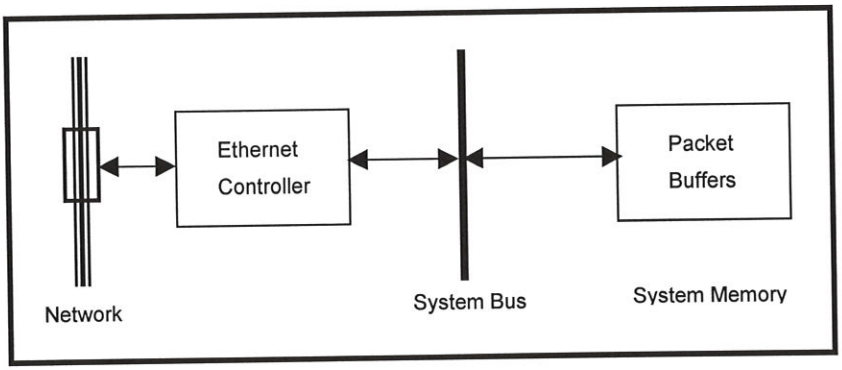


FIGURE 11: Bus Master Architecture [27, p.4].

In such a configuration, the Ethernet controller is connected directly to the system bus. Received packets and packets to be transmitted are DMA'ed directly to and from system memory. This arrangement requires that the processor must pass control of the system bus to the Ethernet controller quickly so that data coming from the network is not lost. The DP83902 and SH7032 are both able to operate in this arrangement. However, the DP83902A requires that it becomes and remains bus master without interruption while it is performing a DMA transfer. Unfortunately for this project, the

SH7032 automatically revokes bus ownership every time automatic DRAM refresh is required [8, p.149]. The DP83902A drops any data that is transferring if interrupted in this way [16, p.43].

Hence, an alternative I/O Mapped Slave Architecture, as seen in Figure 12, was required [27, p.3]. In this arrangement, the Ethernet controller has its own local data and address bus connected to some additional packet buffer memory. This allows the DP83902A controller to DMA packet data to and from the local packet buffer without interruption. The data port connection between the local and system bus allows the two to be connected on request to transfer data between local and system memory. The processor is notified whenever a packet is received or sent and the DP83902A is programmed to use a separate remote DMA controller to transfer data between the system memory and the packet buffer memory through the data port.

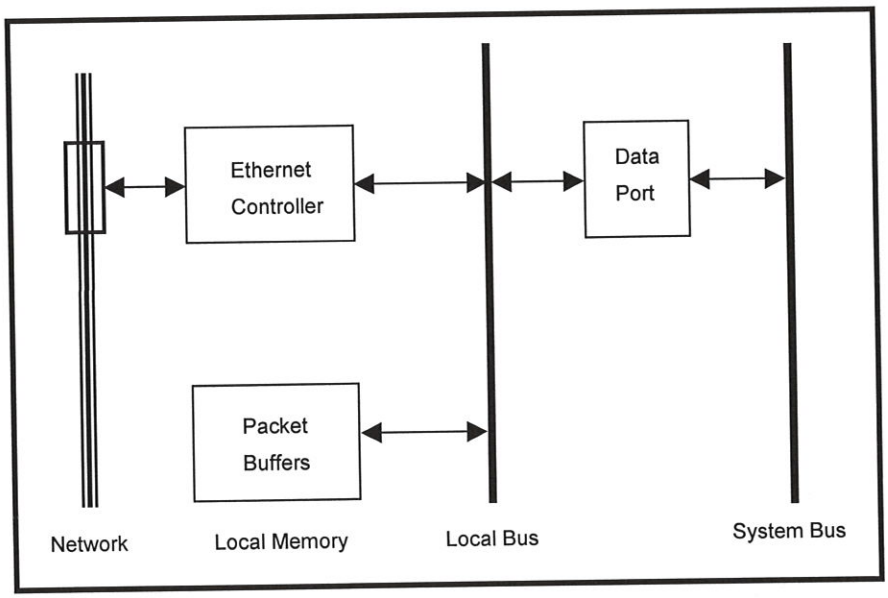
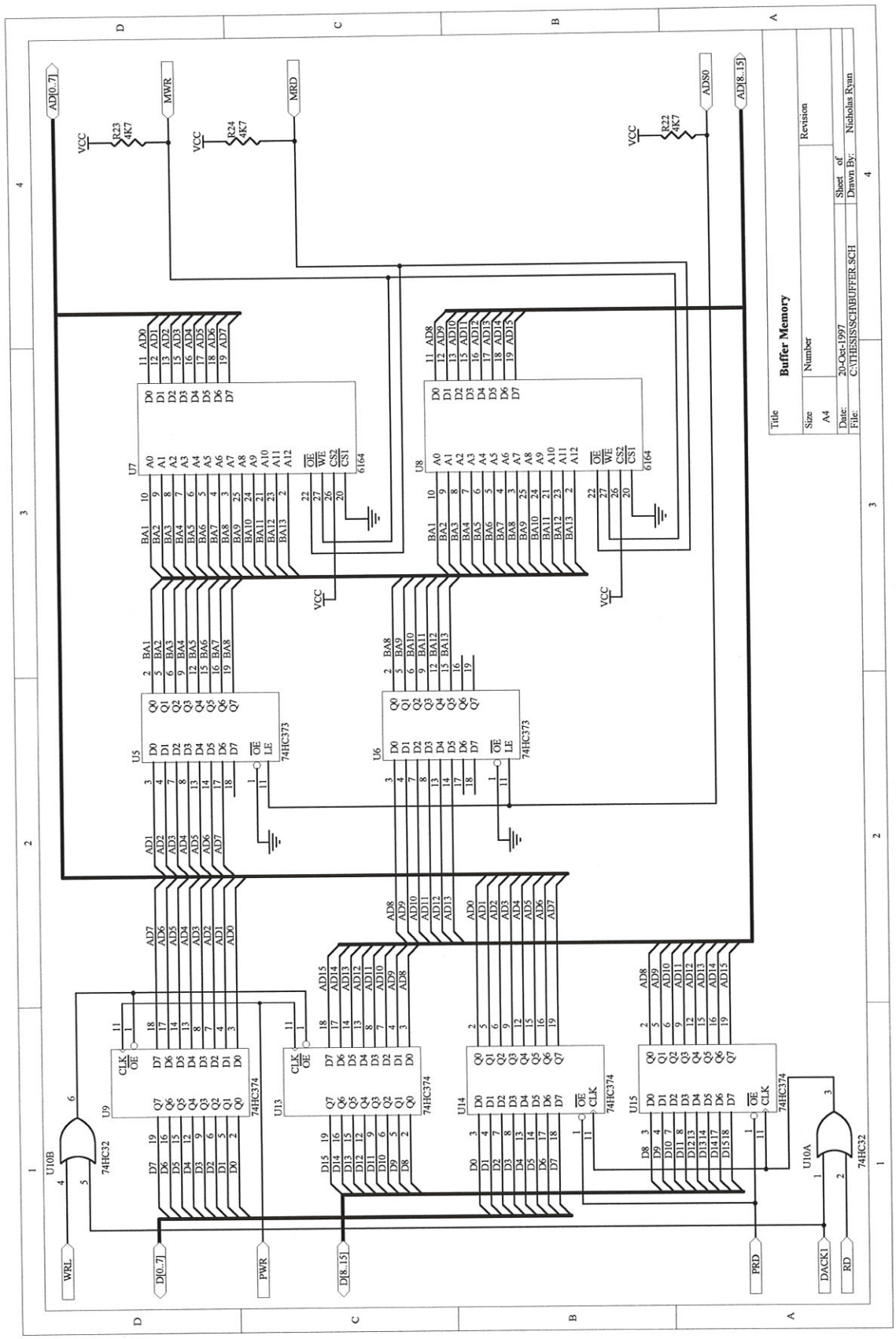


FIGURE 12: I/O Mapped Slave Architecture [27, p.3].

The circuit seen in the *Buffer Memory* schematic overpage implements this architecture. Two 6164 8Kbyte SRAMs are used to provide 16Kbyte of buffer memory accessed using a 16-Bit data path. The DP83902A multiplexes address and data onto a single 16-bit bus. Because of this two 74HC373 Octal latches were used to latch the addresses when accessing the SRAM. Latching is performed by using the DP83902A signal ADS0 which is provided for this purpose. The Master Read Strobe, /MRD and Master Write Strobe, MWR signals from the DP83902A are used to select whether to read from or write to the SRAMs.


Title
Buffer Memory
Size
A4
Number
Revision
Date:
20-Oct-1997
Sheet of
File:
C:\THESIS\SCH\BUFFER.SCH
Drawn By:
Nicholas Ryan
U5
U6
U7
U8
U9
U10A
U10B
U13
U14
U15
74HC373
74HC374
74HC32
6164
R22
R23
R24
4K7
VCC
AD0..7
AD8..15
AD80
MWR
MRD
WRL
RD
PWR
PRD
DACK1
D0..7
D8..15

Four 74HC374 Octal D-Type Flip-Flops were used as the interface between the local bus and the system bus. These provide a 16-Bit wide bi-directional data path. Data written to the buffer memory is latched onto the local bus using the SH7032 signals read, /RD and DMA Acknowledge 1, /DACK1 which are OR'ed together. These signals are output in response from a DMA request on Channel 1 from the DP83902A. The /DACK1 acknowledges the request and the /RD signal is present as the data has been *read* from system memory. Similarly the write, /WR and DMA Acknowledge 1, /DACK1 signals are used to transfer data from the local bus to the system bus. In this case, the data is being *written* into system memory.

The schematics for the thesis board are compiled in Appendix A, while associated timing diagrams may be seen in Appendix B. The PCB layout for the developed thesis may be seen in Appendix C. A photograph of the completed board piggybacked onto the evaluation board may be seen in Figure 13.

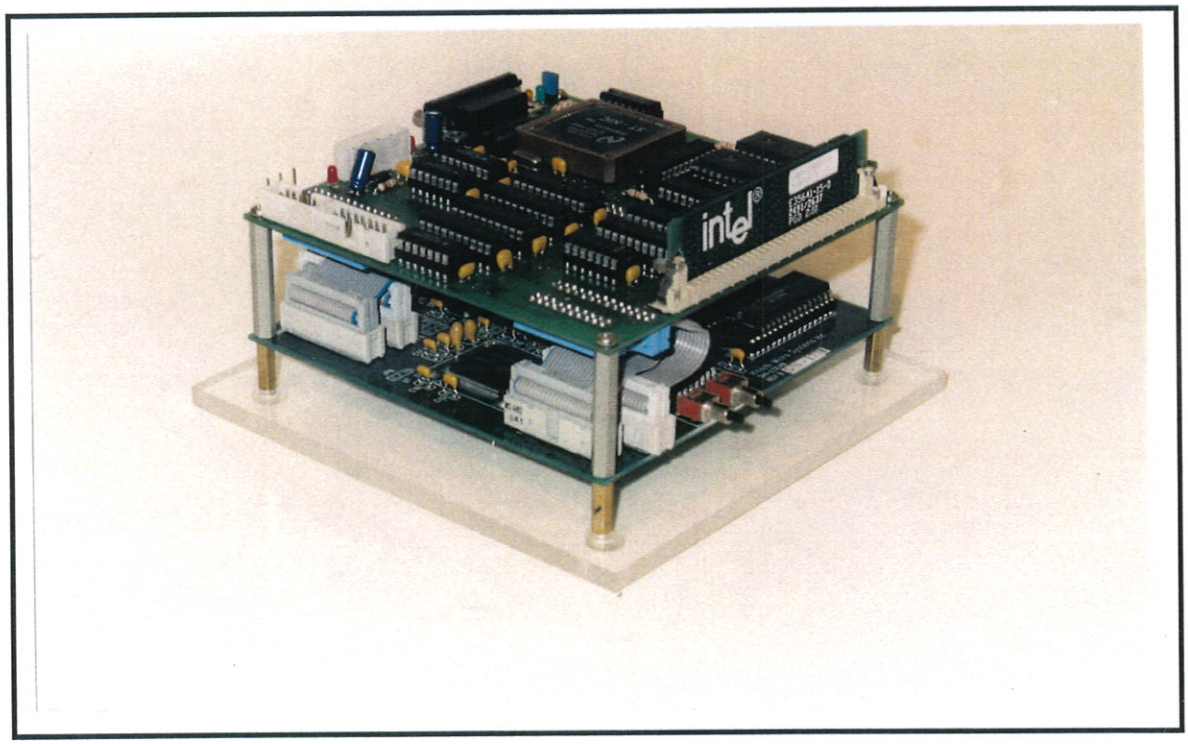


FIGURE 13: Photograph of the completed thesis board.

3.4 Appliance Interface

For demonstration purposes, a Sunbeam toaster was procured for use. The toaster was an old design in which a mechanical switch was triggered by inserting a slice of bread. This caused the toaster to lower the bread and turn on the heating element if the power was on. For this project the mechanical toast sensing switch was altered so that it closed two contacts connected between 5 volts and a parallel input on the project board.

The toaster lowered the bread by using a piece of metal that expanded as the heating elements heated it. Thus when the elements turned off, the toast would rise as this strip cooled down. A sliding darkness control on the front moved a metallic strip closer to a mechanical power switch inside that turned off when the strip expanded. Thus by adjusting this strip's position the heating time could be altered. This darkness control was removed and the internal mechanical power switch was made permanently on. Power to the toaster was instead controlled by a Solid State Relay (SSR) that could switch 240volt AC at 30A. Control of this SSR was by a 5 volt signal and required approximately 0.6mA. A parallel output from the thesis board was used to drive this. Note that only the active line of the mains supply is switched. Ideally for safety, in a commercial product, both the active and neutral lines would be switched. A diagram of this configuration may be seen in Figure 14.

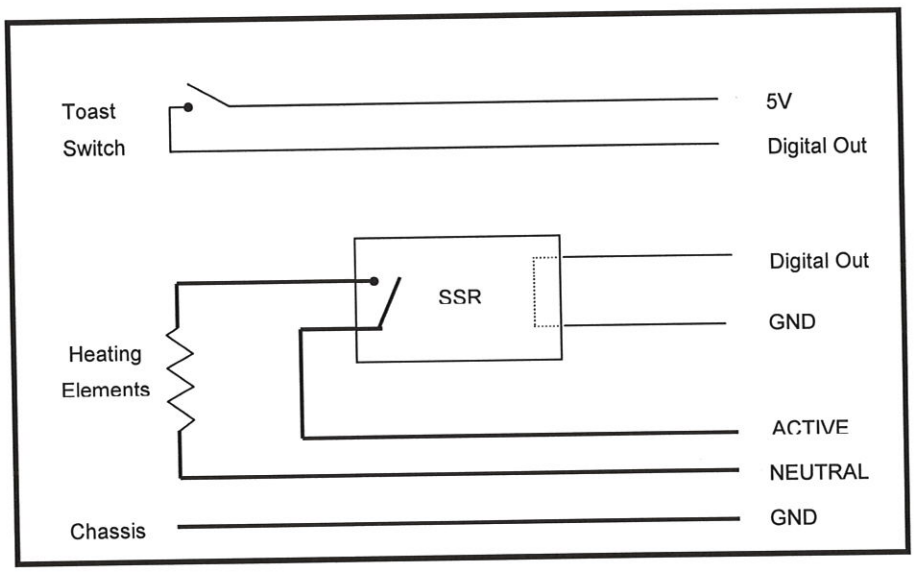


FIGURE 14: Interfacing to the toaster.

The configuration outlined above allowed the processor to determine whether toast had been put into the toaster as well as controlling the amount of time the heating elements were switched on. Remember also that the toast was lowered or raised automatically by the piece of metal that expanded and contracted as a result of the heat generated by the heating elements. A photograph of the complete system ready for operation may be seen in Figure 15.

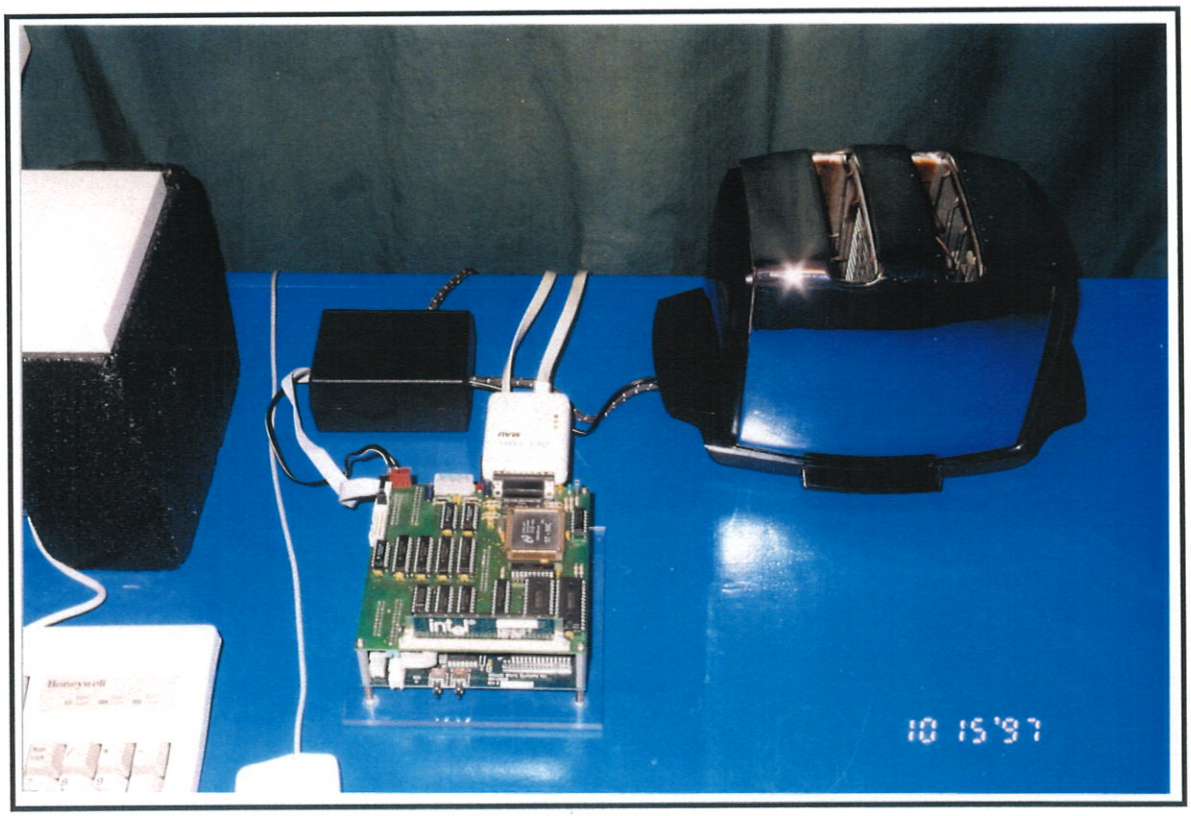


FIGURE 15: Photograph of the complete thesis project.

4.0 SOFTWARE IMPLEMENTATION

Development of the system software began after the hardware had been designed and manufacture of the PCB was awaited. The software required includes a simple, low-level system initialisation file, a multi-tasking Operating System, an Ethernet device driver, a TCP/IP protocol implementation, an API to this software and a simple, application layer HTTP server. The implemented source code may be found on the accompanying 3 ½ inch floppy disk included at the back of this thesis. The software file structure follows the hierarchy shown in Appendix E.

4.1 System Initialisation

The assembler file *start.s* configures a number of the internal SH7032 peripherals. The peripherals have a large number of memory-mapped control registers that are located in the address range H5FFFEC0 -- HFFFFF7. Many of the SH7032 pins have a number of functions. *Start.s* firstly configures the SH7032 Pin Function Controller (PFC) which selects which function each of the SH7032 pins will perform [8, Ch.15]. Table 2 summarises the options selected.

TABLE 2: Pin Function Controller Configuration

REGISTER	ADDRESS	VALUE	CONFIGURATION
Port A Control 1	H5FFFFC8	HF122	Enable DREQ1, DACK1, IRQ0 and TIOCA1 (used for PWM)
Port A Control 2	H5FFFFCA	HBF98	Enable /RD, /WRH, /WRL, /WAIT, /CS6 and /RAS.
Port A I/O	H5FFFFC4	H0400	Enable TIOCA1 as output.
Port B Control 1	H5FFFFCC	H00AA	Enable PB15, PB14, PB13, PB12, (output to toaster) RxD0, TxD0, RxD1 and TxD1 (serial ports).
Column Address Strobe Pin Control	H5FFFFEE	H6FFF	Enable /CS1 and /CAS.

The second peripheral to be configured is the Bus State Controller (BSC) [8, Ch.8]. This allows the configuration of the memory areas. In particular the sampling of the /WAIT pin, the number of wait states to automatically insert and the refreshing and access of DRAM may be selected. Table 3 shows the configuration used for this project.

TABLE 3: Bus State Controller Configuration

REGISTER	ADDRESS	VALUE	CONFIGURATION
Bus Control	H5FFFFA0	H8800	Area 1 DRAM enabled. Area 6 data / address multiplexing disabled.
Wait State Control 1	H5FFFFA2	H42FF	Area 6 /WAIT pin sampling enabled. DRAM wait state insertion enabled.
Wait State Control 2	H5FFFFA4	H0000	DMA transfer wait state insertion disabled.
Wait State Control 3	H5FFFFA6	H7800	External control of /WAIT pin enabled. 4 wait states inserted for areas 0 and 2. 3 wait states inserted for area 6.
DRAM Control	H5FFFFA8	H2600	2 state /RAS pre-charge Multiplex address 10 x 10
Refresh Control	H5FFFFAC	H5A80	/CAS-before-/RAS refresh enabled
Refresh Time Constant	H5FFFFB2	H9696	Refresh timer constant of 150 pulses
Refresh Timer Control/Status	H5FFFFAE	HA508	Timer based on system clock / 2

The SH7032 Interrupt Controller (IC) allows the assignment of priority levels for each of the external and internal interrupt sources on the chip [8, Ch.5]. The priority levels range from 0 (lowest level) to 15 (highest level). It also allows the setting of an interrupt mask value that masks off interrupts that have a priority level lower than the mask value. The mask value may be between 0 (all interrupts enabled) and 15 (all interrupts disabled). The Non-Maskable Interrupt has a priority level of 16 and thus may not be masked off. At system startup, the mask is set to 15 so that all interrupts are disabled. Table 4 summarises the interrupt settings for this project.

TABLE 4: Interrupt Controller Configuration

REGISTER	ADDRESS	VALUE	CONFIGURATION
Interrupt Priority A	H5FFFF84	H7000	Assign priority level 7 to IRQ0 (used by DP83902A).
Interrupt Priority C	H5FFFF88	H00A0	Assign priority level 10 to Timer 0 (used by OS as a Real Time Clock).

The Direct Memory Access Controller (DMAC) provides for four control registers for each of the four DMA channels supplied by the SH7032 with an additional master control register [8, Ch.9]. The DMA Channel Control Register 1 is used to configure DMA channel 1 which is used by this project. The only setting made at this stage is to make the DACK1 output an active low signal.

The Integrated Timer Pulse Unit (ITU) of the SH7032 provides five 16-bit timers [8, Ch.10]. Timer 1 is setup to be used for Pulse Width Modulation and is used in conjunction with the TIOCA1 pin. Timer 0 is configured to provide an interrupt every

26.2mS which is used as a Real Time Clock (RTC) for the operating system. This is done by setting its clocking frequency to the system clock divided by 8 (20Mhz / 8) and allowing it to trigger an interrupt and reset its count whenever it matches the default value of HFFFF in the Timer 0 General Register A. The settings for this timer are summarised in Table 5.

TABLE 5: Integrated Timer Pulse Unit Configuration

REGISTER	ADDRESS	VALUE	CONFIGURATION
Timer Mode	H5FFFF02	H02	Timer 0 normal, Timer 1 in PWM mode
Timer 0 Control	H5FFFF04	H23	Reset Timer 0 count on a compare match. Timer 0 frequency is system clock / 8.
Timer 0 Interrupt Enable	H5FFFF06	HF9	Enable Timer 0 compare match interrupt.
Timer Start	H5FFFF00	HE1	Start Timer 0.

After setting up the peripherals, *start.s* copies a number of interrupt vectors that are used by the MONITOR debugging package provided on the evaluation board EPROM into SRAM. This is to allow interrupt vectors to be defined by the user. The Vector Base Register (VBR) is used to point to the base of the vector table so that no matter where the vector table is placed in memory, the relative interrupt vector offsets are still valid. The VBR is changed to point to the beginning of the vector table in the SRAM. *Start.s* then zeroes out the memory space used by uninitialized global variables. The addresses for this area are provided by the project linker file which will be discussed later. The code then jumps to the address of the procedure *main* which is written in C and is the start of the operating system code.

4.2 Operating System

The structure of the OS will be discussed first before describing the initialisation procedures contained in the procedure *main*. As mentioned before, the OS maintains a real time clock that is triggered by the Timer 0 interrupt. The ISR for this interrupt may be seen in the file *time.c*. This procedure decrements two counters: one that counts from 40 and one that counts from 4. These are used to count in units of approximately 1S and 100mS respectively. (Recall that the Timer interrupts every 26.2mS) The 1S counter is used to increment the number of seconds for which the system has been operating. The 100mS counter is used to both wakeup sleeping process and to trigger rescheduling of processes.

The operating system provides for the creation, suspension, resumption and killing of processes. The system calls that perform these functions may be found in *sys.c*. Each process has its own associated data structure which is defined in *process.h*. These process structures are stored in process table and indexed by the process ID which is assigned to each process as it is created. The data stored in the process structure include the value of registers that need to be saved and restored during context switching, the process priority, the current state of the process, the process' stack address and length and the address of the code for the process. When a new process is created with the *create* system call, the address of the process code, the stack size and process priority are passed as arguments. *Create* allocates space for the process stack, generates a new process ID and uses this to index into the process table and define a new process data structure. The system call *kill* simply frees the stack space used by the process to be killed and sets its associated entry in the process table as free. If the killed process was currently running, process rescheduling is performed.

The OS maintains a queue of processes that are ready to run. Every time a context switch is required (either due to rescheduling pre-emption or a currently running process becoming no longer eligible to run), the procedure *resched* in *proc.c* is called. This procedure inserts the currently running process into the ready queue and removes the highest priority process from the ready queue as the next process to run. It then calls the assembler routine found in *switcher.s* to save the pre-empted process' registers in its associated data structure and restores the new current process registers. When this routine returns, the process removed from the ready queue is now running. A process may be suspended by the system call *suspend*. This removes the specified process from the ready queue so that it is no longer available for rescheduling. If the process was currently running, *resched* is called to select a new current process. The system call *resume* makes a suspended process eligible for scheduling again by calling the procedure *ready* in *proc.c* which simply adds the specified process to the ready queue.

A process may put itself to sleep by calling *sleep* which puts the process into a sleep state for a period specified in seconds. To do this the process state is marked as SLEEPING and the process is put into a sleeping queue. Every time the 100mS timer is incremented in the clock interrupt, the sleeping queue is checked to see if any

processes should be woken up. To wake up a process, it is removed from the sleeping queue and marked as READY. The diagram in Figure 16 shows process states and transitions in this OS. Note that the states WAITING and RECEIVING will be discussed below.

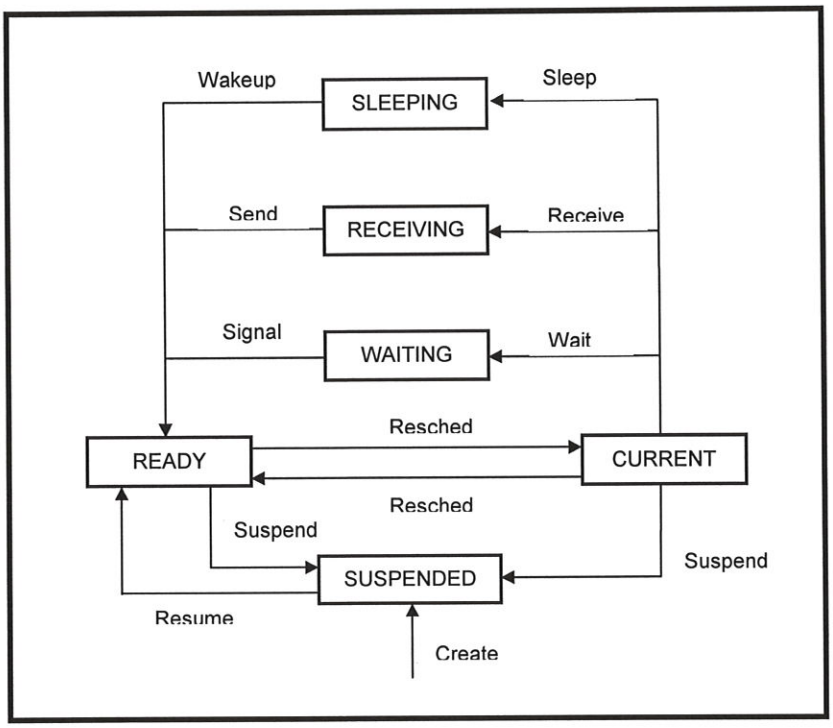
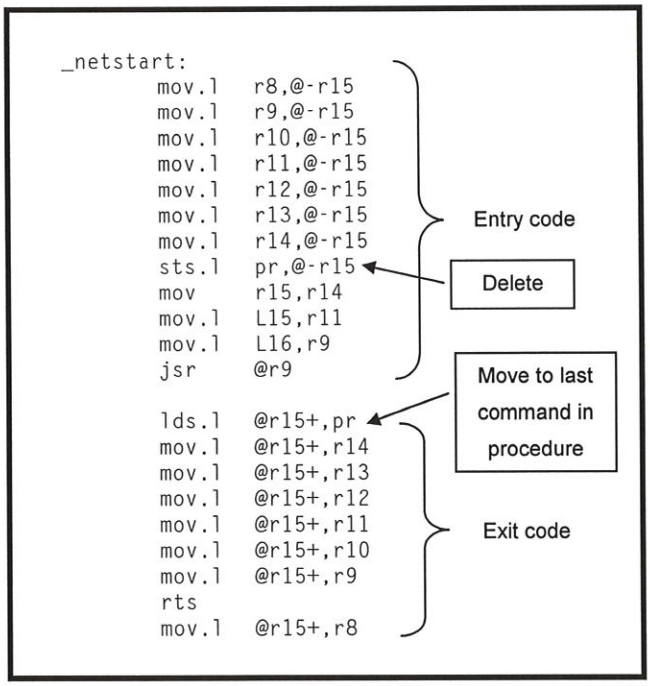


FIGURE 16: Process State Diagram.

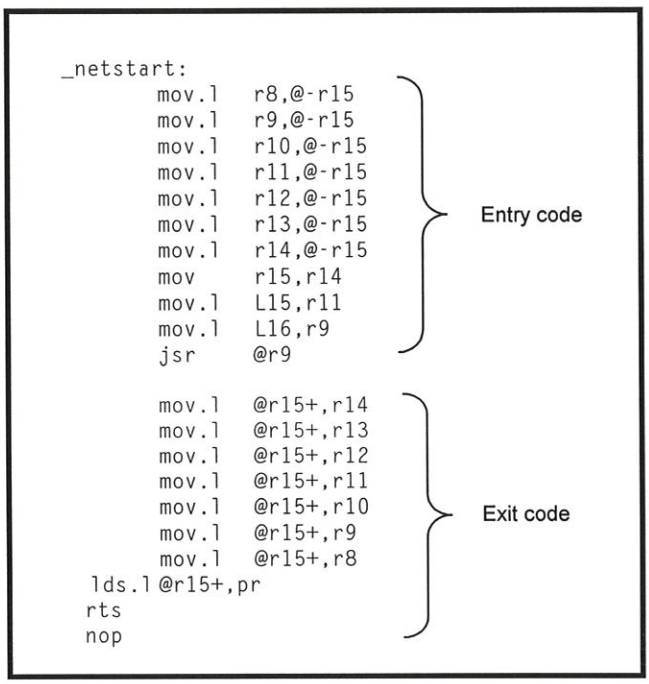
At this point is important to mention a slight fix required in assembler code for prcoedures that are to run as processes. Usually when a subroutine jump is compiled in assembler, the return address is pushed onto the stack. However, the SH7032 has a special Procedure Return (PR) register. The C compiler for the processor uses this register to store the address to jump to after the subroutine call returns. This is fine in normal procedure calls, but does not function correctly when context switching is required. This is because when a process becomes current for the first time, the return address is saved in the PR register. In this case the address saved is that of the *resched* procedure. When the process finishes this address will be returned to rather than the required *userret* procedure in *proc.c*. To solve this problem the code in *switcher.s* enters the correct return address in the PR register but the entry and exit code in the process procedure must be modified so that the correct PR value is not overwritten. To do this, process procedures must be compiled to assembler code, edited and then further compiled to object code. Listing 1 shows an example of the original assembler code produced when compiled, while Listing 2 shows the required changes. Note that

editing at the end of the procedure is only required for a process that returns (i.e. does not run in an infinite loop).

LISTING 1: Assembler code produced by compiler.



LISTING 2: Assembler code with required changes.



As well as process control, the OS provides quite sophisticated memory management. The memory available for use by the OS in this project was the 256Kbyte of SRAM, 1Mbyte of DRAM and 8Kbyte of internal RAM. The compiled code is downloaded into the SRAM during development. It was decided that the internal RAM would be used as stack space as it provides fast access, the DRAM used for dynamically allocated space and uninitialised global data while the free space in the SRAM would be used for pre-initialised global variables. This organisation was created through the use of the linker command file *sr.cmd* which may be seen in Listing 3.

LISTING 3: SR.CMD linker command file.

```
ENTRY(_start)
OUTPUT_FORMAT(srec)
MEMORY {
  dram : ORIGIN = 0x1000000, LENGTH = 1024K
  sram : ORIGIN = 0xA000000, LENGTH = 64K
  ram  : ORIGIN = 0xF000000, LENGTH = 8K
  }
SECTIONS {
  .vects 0xA002000 : { *(.vects) } > sram
  .text : { *(.text) ; } > sram
  .mdata : { _data = . ; *(.data) ; _edata = . ; } > sram
  .bss 0x1000000 (NOLOAD) : { _bss = . ; *(.bss) *(COMMON) ; _ebss = . ; } >
        dram
  .stack 0xF001FFC : { _stack = . ; *(.stack) } > ram
}
```

This file specifies that the linked file format should be srec, which is used by the MONITOR debugging program provided on the evaluation board EPROM. As well, the size and start address of the three memory areas are defined and assigned a name. It then specifies in which of these areas each section of the code should be placed. The first section is *vects* which is specified in the file *vects.c* and contains user defined interrupt vectors (including those for the clock and ethernet ISRs). Note that some of these vectors are overwritten by the MONITOR vectors in *startup.s*. The next defined section is *text* which contains the actual program code. The *mdata* section consists of the initialised global variables while the *bss* section consists of the uninitialised global variables. The NOLOAD option is specified for this section which means that data is not downloaded to this section. This is firstly because there is no valid data to download and secondly because the DRAM in not usable until it is configured by *start.s*. The *bss* and *ebss* values are however, used in *start.s* to zero out this section as mentioned previously. The initial value of the stack pointer is set to the top of the internal RAM and is also used in *start.s*.

With the memory configured as above, the OS needed to provide management of the free DRAM used for dynamically allocated heap space as well as the internal RAM stack space. The system calls *getheap*, *freeheap*, *getstack* and *freestack* were written to achieve this. The OS maintains a list of free blocks of heap and stack memory. *Getheap* and *getstack* both search their corresponding free block lists and use the first available block that is big enough. *Getheap* searches the space from lowest address to highest, while getstack searches from highest to lowest as the stack always grows downwards. *Freeheap* and *freestack* return allocated memory to its corresponding free list and also merges any adjacent free blocks. Both *getstack* and *freestack* are used automatically during creation and killing of processes respectively, while *getheap* and *freeheap* need to be called from a user process.

Higher level memory management was also implemented to prevent the system from deadlocking. Deadlock can occur if one or more processes are allocated all the available heap space. This is highly possible in a networking system where packet buffers are continuously allocated. In this situation a process that needs heap space to process these buffers is unable to run and the system deadlocks. To prevent this the memory space is partitioned into buffer pools and a process is allocated a memory partition in which it may use as much memory as it requires. The system calls required to implement this appear in *syscall.c*. *Poolinit*, *mkpool* and *mark* are used to initialise the buffer pools. *Getbuf* returns a free buffer from a specified buffer pool while *freebuf* returns a buffer to its correct buffer pool.

The OS provides simple but powerful process communication and coordination through the use of semaphores implemented in the system calls *wait* and *signal* found in *syscall.c*. A process that calls *wait* on a semaphore s, decrements s and then returns immediately if the value of s is greater than or equal to zero. If not, the process' state is changed to WAITING and the process ID is added to a queue of waiting processes associated with the semaphore s. A *signal* call on a semaphore s, increments the value of the semaphore and makes the first waiting process, if one exists, ready to run. System calls *screate* and *sdelete* are provided so that processes may dynamically create, use and delete semaphores from the total number provided by the system.

The process data structure mentioned previously also contains a defined space for messages to be stored that form the basis of a simple message passing mechanism between processes. The system call *send* allows a message to be sent to a process specified by a process ID. The system call *receive* checks the calling processes data structure for a message and returns immediately if one exists. If not, the processes' state is set to RECEIVING and the process is made ineligible for scheduling. Once a message is received, the process state is set to READY and the process is again able to be scheduled. Two variants of receiving a message are supplied by the system calls *recvclr* and *recvtim*. *Recvclr* checks for a message for the calling process. Whether or not one exists the call returns immediately. *Recvtim* behaves much like *receive* however after a specified time out period if no message is received the process state is set to READY and made eligible for scheduling.

A higher level message passing capability is also implemented in this OS. Communication ports are not process specific and provide the capability of storing a number of one word messages [4, p.242]. A port is implemented using a FIFO queue and two semaphores. The semaphores are used to block processes that attempt to add a message to a port when it is full and processes that attempt to retrieve a message from an empty port. Thus a processes that sends a message to a port using the system call *psend* returns immediately unless the specified port is already full in which case it remains blocked until at least one message is retrieved. Similarly a process using the system call *preceive* returns immediately with the message at the head of the port queue or is blocked on an empty queue until a message arrives. The system call *pdelete* deletes a port and frees any waiting messages and associated blocked processes, while *preset* frees messages and processes but leaves the port available for use again. These system calls may again be found in *syscall.c*.

Now that the OS structure has been described, its startup initialisation may be described. As mentioned earlier, the code in *start.s* calls the procedure *main* which is the start of the operating system proper. *Main* firstly writes a startup message over the serial port to the development computer using a routine provided by the MONITOR debugging program. *Sysinit* is called which initialises all aspects of the operating system including major system variables, free stack and heap memory lists, process table entries, semaphores, process ready list, buffer pool management and the real

time clock. As well a process table entry is made for a null process. A null process is required as when there are no user processes available to switch to, the scheduler still needs to perform a context switch. After *sysinit* returns, the RTC is running but interrupts are still masked. The system outputs free memory space details to the development PC over the serial port and then enables the system interrupts. At this stage, *main* becomes the null process. The null process enters an endless FOR loop after creating and starting the process *netstart* which initialises and starts the networking software.

4.3 Device Driver

The device driver for the Ethernet controller consists of four main routines. *Ethinit* is used to initialise the DP83902A and Ethernet data structures. *Ethwrite* and *ethread* are used by high level routines to write and read packets to and from the network, whilst *ethint* is the ISR that is called whenever the DP83902A completes reception or transmission of an Ethernet packet. The Ethernet controller has an associated data structure that stores the physical and broadcast addresses associated with the controller, the number of the network interface it belongs to and a pointer to an output packet queue. This structure is defined in *ethernet.h*.

Ethinit firstly allocates space for the output queue and assigns the physical and broadcast addresses as well as outputting them to the development PC over the serial port. The DP83902A is then initialised for use. This involves configuring a number of the DP83902A control registers, setting the physical address so that packets may be filtered, resetting the interrupt status register and putting the device into start mode so that reception and transmission may be performed.

To write a packet over the Ethernet, *ethwrite* is called with a specified packet buffer and length of this packet. *Ethwrite* firstly ensures that the packet length is within the allowable range. If the packet is too big it is discarded and if it is too small it is zero padded to the minimum length. Following this, the physical address of the Ethernet controller is copied into the physical address portion of the packet to be sent. The packet is placed in the output queue for the controller or discarded if the queue is full. The procedure *ethxmit* is then called after disabling interrupts. *Ethxmit* checks whether the DP83902A is currently transmitting a packet. If it is, *ethxmit* returns, as

when the current transmission is completed, the ISR will automatically retrieve the next packet from the output queue to send. However if no transmission is currently taking place, one must be initiated. To do this, a packet is retrieved from the output queue and DMA'ed to the local packet memory. The DP83902 is then given the address of the start of the packet in local memory and the transmit command is written to the command register.

Ethread is not explicitly called by a higher level procedure. Instead *Ethint* calls it when a packet is received. Once the packet is read *Ethread* sends the packet to a higher level protocol. The packet is transferred upwards through the protocol software until it reaches an application that is waiting for received data. *Ethread* commands the DP83902A to transfer a packet via DMA from the local memory to system memory. Once the packet is transferred, *ni_in* is called which demultiplexes the packet depending on its protocol type and sends it to the appropriate network input procedure.

Ethint is called by the Ethernet interrupt that occurs when either a packet has been successfully transmitted or received or when the local receiving buffer has overflowed. A loop is entered in which all pending interrupts of the DP83902A are serviced. The ISR firstly checks whether the receive buffer has overflowed. If it has all buffers that have been received are transferred to the system using *ethread.* The DP83902A is reset and if a packet transmission was pending this is started. If there is no buffer overflow, all received packets in local memory are transferred to the system memory. Following this, if a packet has been successfully transmitted, *ethxmit* is called to initiate the transmission of another packet from the output queue.

4.4 Network Protocol Stack

As mentioned earlier, the device driver and ARP software both lie in the data-link layer of the network protocol stack model. The ARP software developed in this system is based around a table containing a number of *arpentry* data structures defined in *arp.h.* These entries are able to contain a physical address and corresponding logical address as well as the state of the entry – if it is valid, invalid, pending translation etc. The file *arp.c* contains a number of procedures that are used to implement the Address Resolution Protocol. *Arpfind* returns an ARP entry for the

specified logical address or returns nothing if an entry does not exist. *Arpalloc* iterates through the table and allocates a free entry for a specified logical address. *Arpsend* broadcasts an ARP request packet over the network given a logical address with an unknown hardware address. When a ARP protocol packet is received and demultiplexed by *ni_in*, it is sent to *arp_in*. If the packet is a request for a translation of a logical address matching that of the system, a translation response is sent using *ethwrite*. However, if the packet is a translation response, and the translation request has not timed out, the translation is added to the ARP table. After this any packets that have been queued for transmission to the previously unknown physical network address are sent by *arpqsend*. The ARP software also includes a timer procedure *arptimer*, that iterates through the ARP table and removes any entries that have reached an age threshold. This ensures that only up-to-date entries remain. *Arpdq* is used to delete any queued packets that are associated with an ARP entry that is deleted. The *arptimer* procedure is called by the process *slowtimer* once a second.

Slowtimer is an endless loop that operates by putting itself to sleep for one second. After it is woken up, it calculates the exact time for which it was asleep using the system call *gettime* that returns the number of seconds for which the system has been running. The *arptimer* and *ipftimer* (discussed later) procedures are then called with the exact sleep time as an argument.

The IP layer in this system is based around the *ipproc* process found in *ipproc.c*. This process loops endlessly. It firstly retrieves a received packet from either the Ethernet interface or the local pseudo interface using the procedure *ipgetp*. The local pseudo interface is implemented similarly to a physical network interface, however packets sent and received through this interface are transferred to and from higher level software on the same system. This greatly simplifies the IP process [6, p.31]. *Ipgetp* chooses the interface to extract the next packet from in round robin order. It returns the packet and the number of the interface on which the packet was received. After discarding any invalid packets, a normal IP implementation would use a complex routing algorithm to determine the next network hop for the packet (this can include sending the packet to higher level software through the local interface). However, as mentioned earlier since this system will not be used as a gateway, it is possible to simplify the routing algorithm to merely check for four conditions. These conditions

are determined by whether the source of the packet is local or non-local and whether the destination is local or non-local.

If both the source and destination is non-local, the packet requires gateway routing and hence the packet is discarded. If the source is non-local and destination is local, the procedure *ipputp* is used to send the packet to the local interface. If both the source and destination are local, a similar call to *ipputp* is made. Lastly, if the source is local and the destination is non-local, *netmatch* is used to determine whether the destination is on the same subnet as the Ethernet interface. If it is, *ipputp* is used to send the packet directly to the destination over the Ethernet interface. If the destination is on a different subnet, *ipputp* is used to send the packet to a gateway so that it may be routed correctly.

The IP code also provides for fragmentation of packets if they are longer than the Maximum Transfer Unit (MTU) size for the Ethernet interface. Reassembly of incoming fragmented packets is also implemented. The IP output procedure *ipputp* fragments IP packets that are to long and uses the procedure *ipfsend* to send these fragments over the Ethernet interface.

Ipreass.h defines the data structures used for reassembling incoming packets. A table of fragment queues is implemented. When a packet is sent from *ipproc* using *ipputp* it is eventually sent to the procedure *netwrite*. *Netwrite* will write the packet to the Ethernet interface if this is specified. If the packet destination is local, the procedure *local_out* is called. Depending on the protocol of the packet received, *local_out* will either send it to an ICMP or TCP input procedure. Firstly however, *ipreass* is called to handle reassembly of fragmented packets. If the fragment received does not complete a packet it is added to a fragment queue using *ipfadd*. However, if the packet is completed, *ipfjoin*, *ipfhcopy* and *ipfcons* are called which return a complete packet. *Ipftimer* is called from the system timer process *slowtimer*. This procedure iterates through the fragment queues updating each fragment's associated time-to-live field and deleting any expired fragments.

As mentioned earlier the ICMP implementation for this system is greatly simplified. A received ICMP packet is sent by the *local_out* procedure to *icmp_in*. *Icmp_in*

discards invalid packets or any packets that are not echo requests. For an echo request, the packet type is changed to echo response and the source and destination addresses are switched. This packet is then sent using the procedure *ipsend.*

Unfortunately, at this stage of development and with time at a premium, the DP83902A Ethernet Controller was broken during testing. The ADS0 output used for the transfer of data to and from local buffer memory was shorted to 5 volts. A replacement was unable to be obtained in time due to large lead times from the supplier. Because of this setback, no further testing of the software could be achieved. Instead, a simple two process system was developed to demonstrate the use of the system, the OS functionality and the interface to the toaster.

4.5 Application Layer

The application layer software used for demonstration purposes consisted of two processes. One process, *input*, was used to provide an interface to the user using the development PC connected via the serial port. Options are provided to the user to check the status of the toaster, start the toaster for a specified amount of time and to halt the cooking of toast. Input from the user was transferred to the *toast* process using *send* where it was used to perform the desired actions. Results were sent back to the input process again using *send*. The *toast* process was started from the *main* procedure instead of *netstart*. The *toast* process in turn started the *input* process.

5.0 SYSTEM OPERATION

5.1 Status

Unfortunately, as mentioned in the previous chapter, the system is not completely developed. This is due to the replacement problems encountered with the DP83902A Ethernet Controller. Further work is required on the testing of the developed network connection and software as well as the implementation of the TCP and application layer software.

A large amount of work and research has been performed on what can only be considered a complex thesis project. This, together with the incomplete state of the project, provides an excellent opportunity for this topic to be pursued further. A continuing thesis project could aim to complete the software development and testing and fully achieve the goals specified in this document. Alternatively, the research and design presented here could be used as a basis for development of a new system.

5.2 Usage

The source code is compiled using the GNU C compiler provided with the SH7032 evaluation board. Use the *make* utility together with the makefiles provided on the source code disk at the back of this document. Typing *make clean* will remove any previously compiled object files and temporary files. *Make all* will compile the C code into object files. Following this, *make ass* will assemble the process files in the process directory into assembler code. After editing these as described in the previous chapter, typing *make sr* will compile the assembler files and link all the object files. This will result in a binary file *toast.sr* that is to be downloaded to the system.

To download the produced file, run the HINT communication program supplied with the SH7032 evaluation board. By default, HINT communicates with the system using COM2. After pressing the evaluation board reset button, a message produced by the CMON program should appear on the HINT screen. Type *l:toast.sr* to download the compiled software. After it has downloaded type *g* to start the software running.

As the software is presently configured a number of messages should appear on the HINT screen describing the amounts of memory available. The user should then be presented with the output from the *input* process.

Connection of the toaster is simple. The 5 pin single in-line plug should be connected to the power testpoints at the rear of the system board. The 10pin IDC socket should be connected to the 10pin polarised plug at the rear left of the system board. The mains power should be connected and turned on. The power to the toaster is not switched by the SSR until the output from the system board is high.

6.0 CONCLUSION

6.1 Outcomes

Despite the incomplete status of the project, the work performed on this project has resulted in a number of successful outcomes. These are provided in list form below.

- A thorough investigation of the origins of the concept of an embedded Web server was performed.
- A thorough investigation of all enabling technologies related to this project was performed.
- The hardware required for a prototype embedded Web server was designed and constructed.
- A relatively complex, yet compact, Operating System was developed for use in the system.
- A device driver for the DP83902A Ethernet Controller was developed.
- The ARP, IP and ICMP sections of the TCP/IP protocol stack were developed.
- The use of the system at its present stage was successfully demonstrated by interfacing it to a household toaster.
- A base for the further development of this powerful yet complex system has been established.

6.2 Future Issues

During development of this project a number of future alterations and additions were conceived. Firstly, the size of the developed prototype was prohibitively large for embedding into existing electronic appliances. The physical size would easily be

reduced by using surface mount components. As mentioned before, once the prototype was thoroughly tested both the evaluation board EPROM and SRAM could be eliminated and replaced by an EPROM containing the developed software. The DRAM and internal RAM would then be the only required memory. Designing a complete system board, rather than using the evaluation board would reduce the size, as in the final system, only the SH7032 from the evaluation board is required.

Perhaps the biggest size reduction would come from reducing the component count of the system. The selection of an Ethernet controller that is able to be interrupted during data transfers would enable the Bus Master Architecture described earlier to be implemented. Because of this, the local buffer memory and data port chips would no longer be necessary. This would remove approximately eight chips from the system. This would also largely reduce the complexity of the system thereby more closely meeting the specific goal of a *simple* embedded Web server.

A further development would be to improve the hardware interfacing to appliances. Rather than a simple parallel I/O interface, a device such as an EPLD could be used to provide interfacing to devices more complex than a toaster.

With regard to the software developed, a number of additions could be made to the network protocol support. Network security would become an issue if this device was to be connected on a network for long periods of time. (You don't want people all around the world cooking your toast!) Access to the Web server would need to be restricted to authorised users. This could be performed by simple user name and password checking or by a variety of networking security protocols that have been developed specifically to solve this type of problem. A File Transfer Protocol (FTP) could be implemented so that files could be transferred to the system [25, Ch.12]. With a large amount of custom programming, it would be possible to download a compiled process. This process could for example, control the appliance into which the system is embedded. Thus the use of the system could be software changeable.

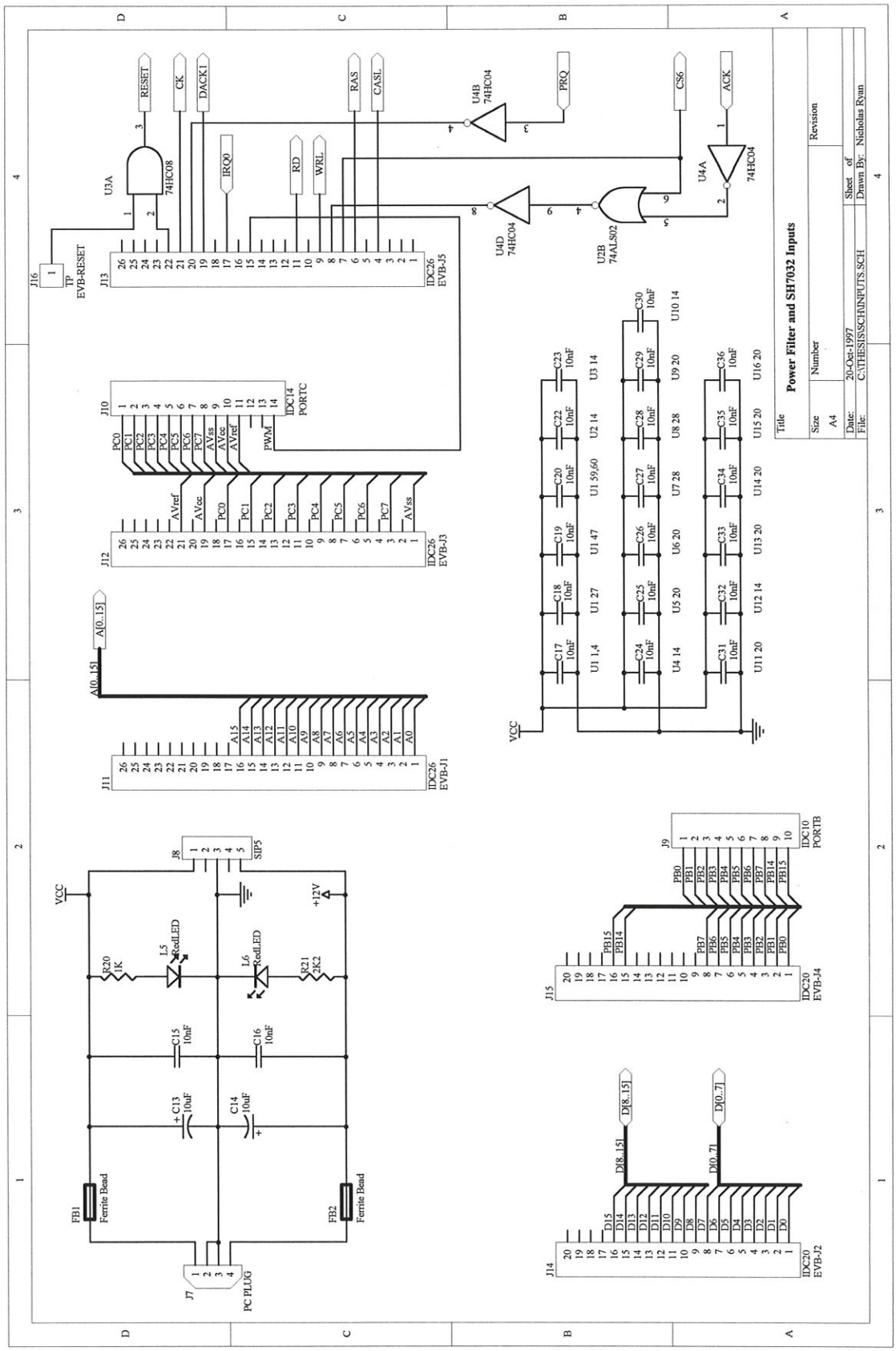
A further useful addition would be to implement the Simple Network Time Protocol (SNTP) that is able to calculate an exact time of day by communicating with time

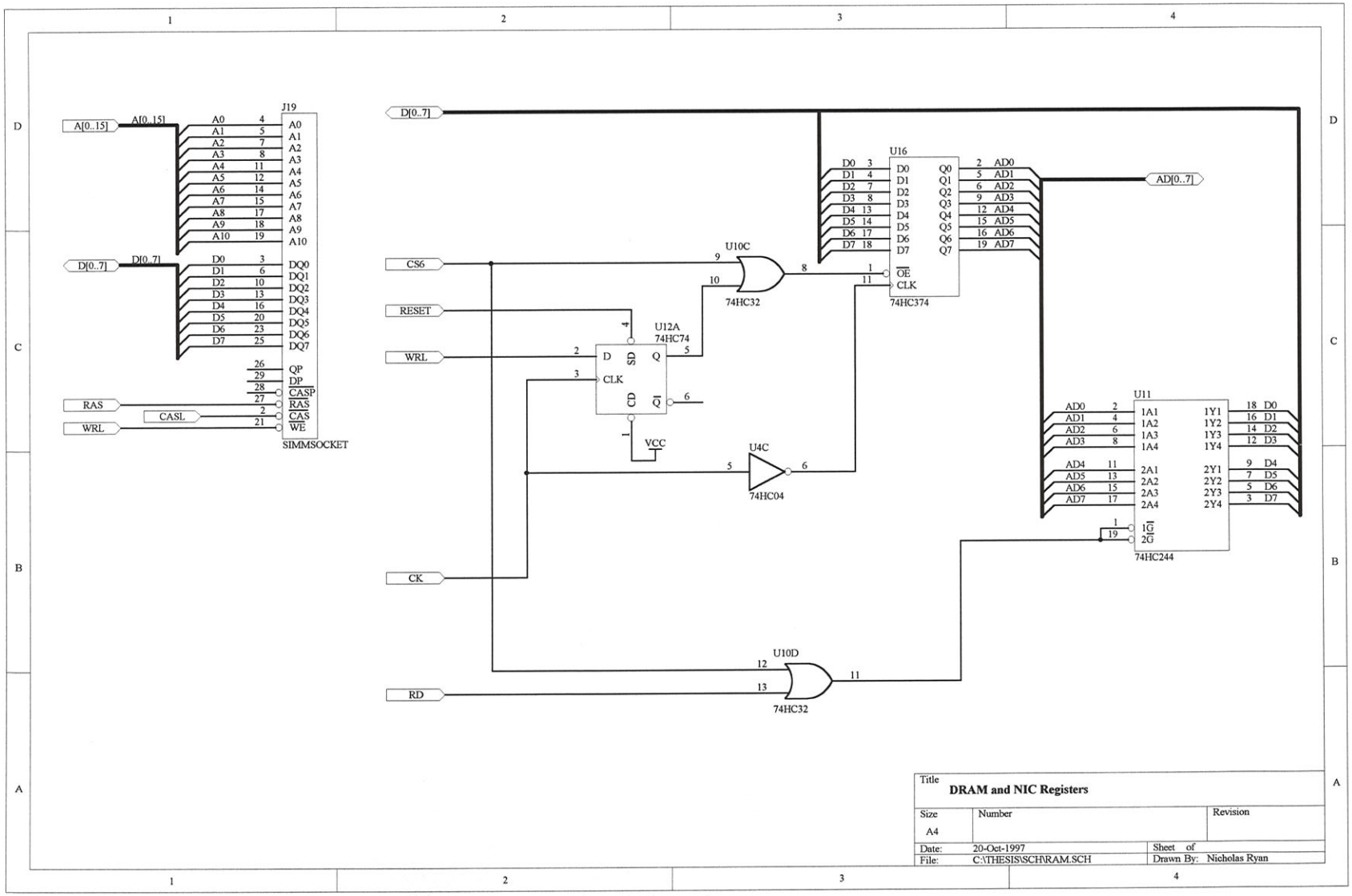
servers on the Internet [25, Ch. 10]. This would enable the system to be used in appliances that contain wakeup alarms such as sprinkler systems and VCRs.

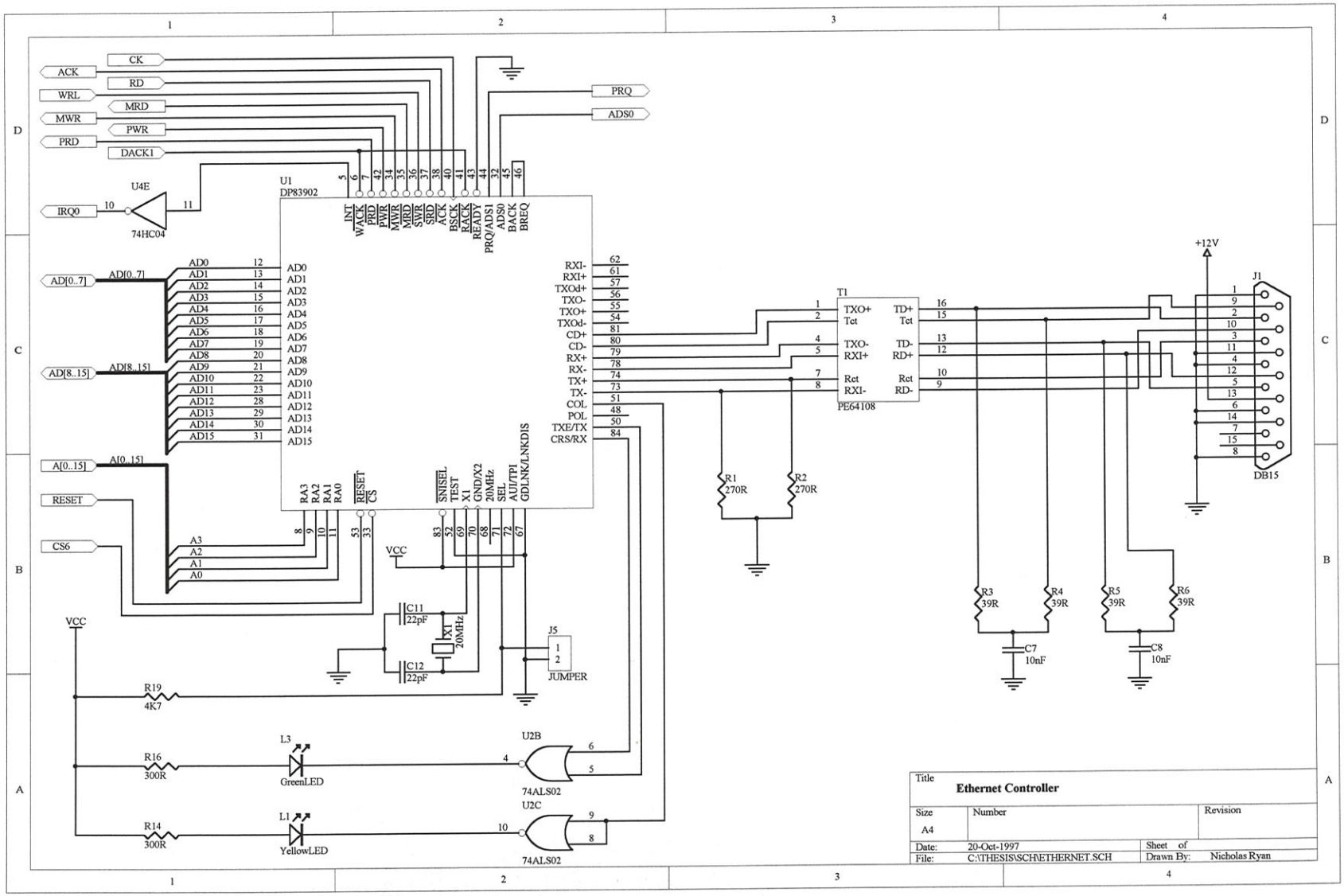
This thesis saw the partial development of an interface that allows control of, and communication with, electronic devices through a WWW based front-end. It uses a number of well established protocols and architectures to guarantee compatibility and widespread use. It also incorporates emerging technologies and concepts to produce a powerful, innovative product that promises to be ideal for the future globally-connected world.

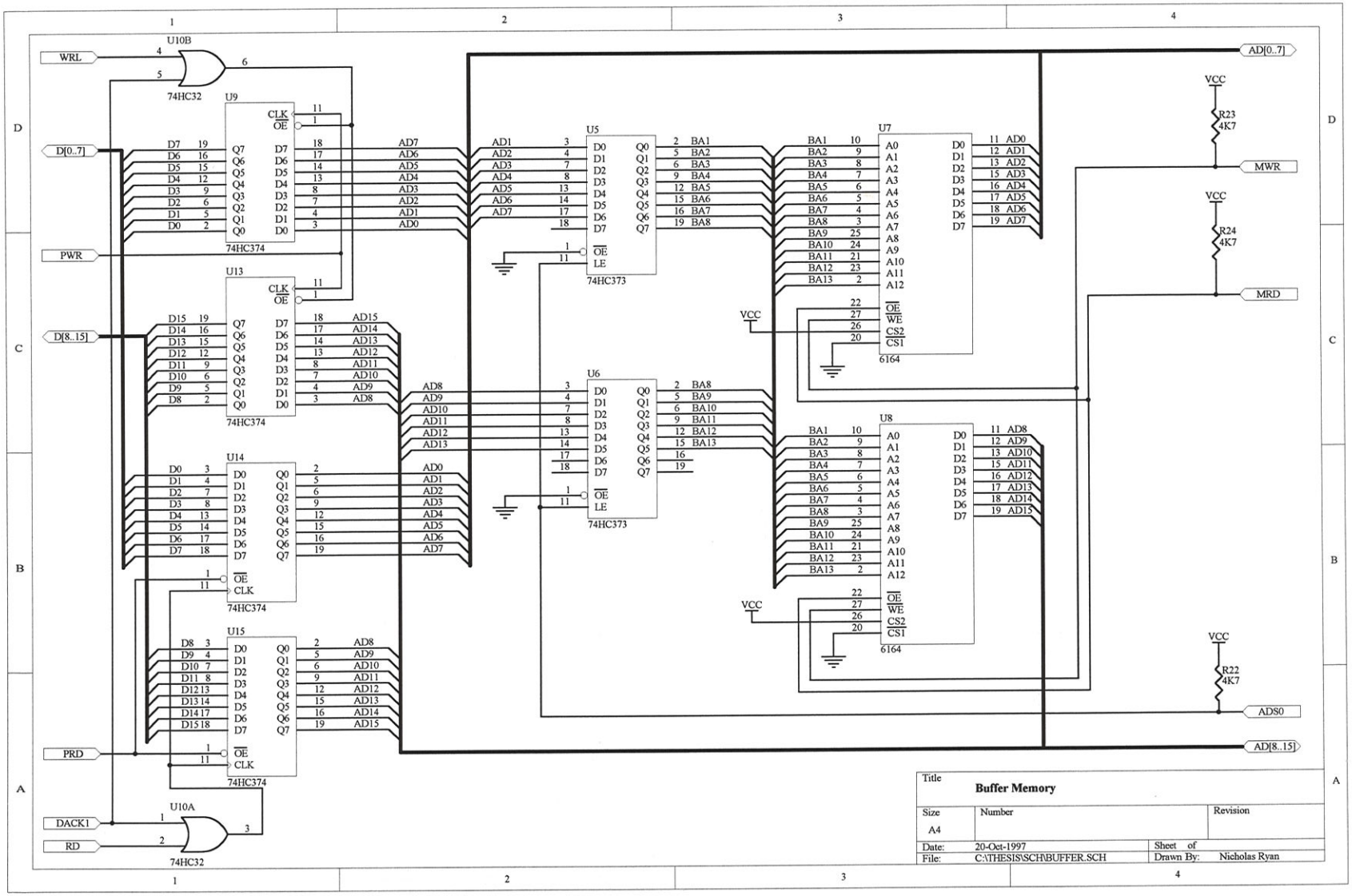
FIGURE 17: The Future? (PC Magazine, 7 January 1997. Page 392)

APPENDIX A : Circuit Schematics

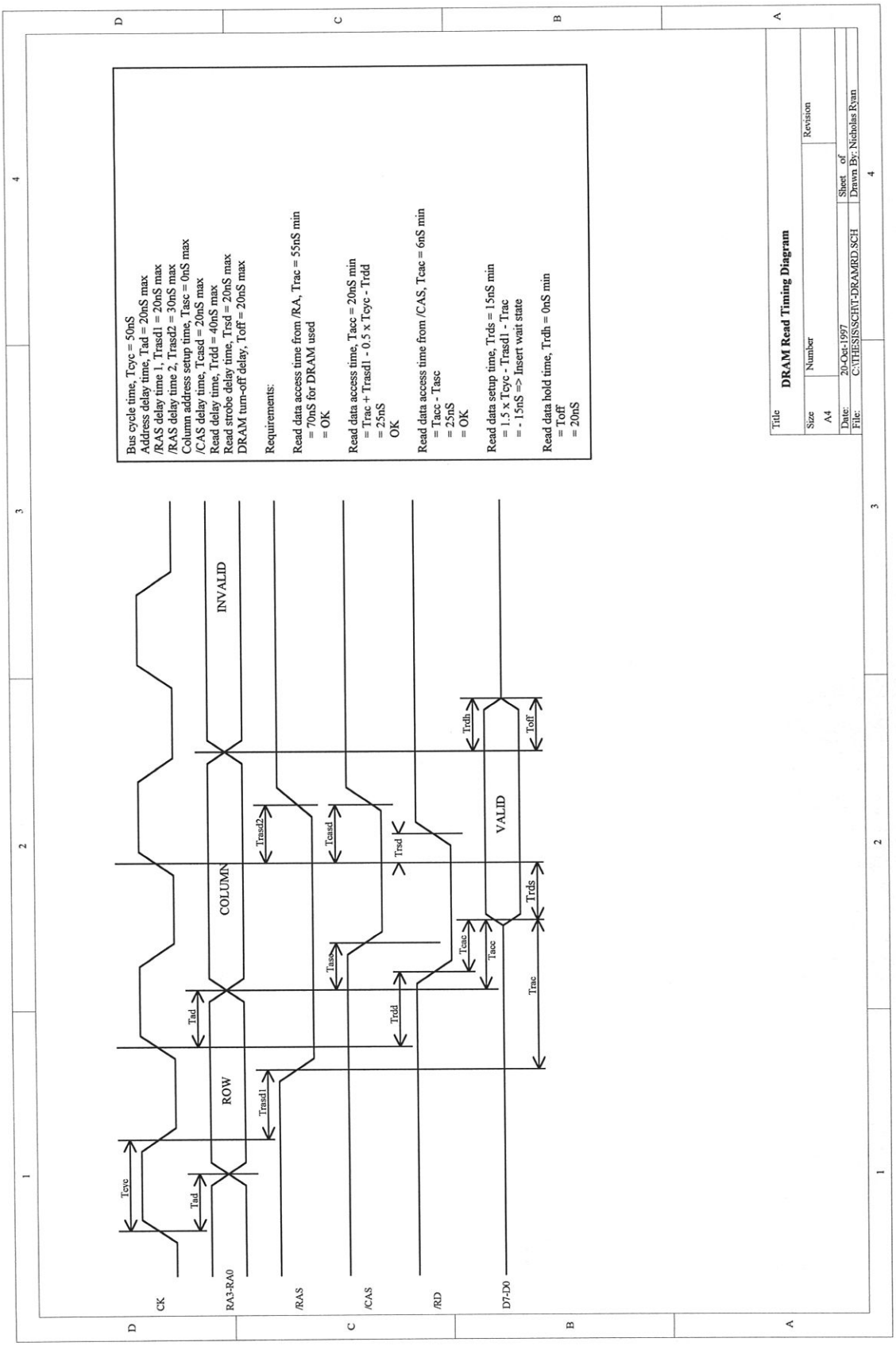


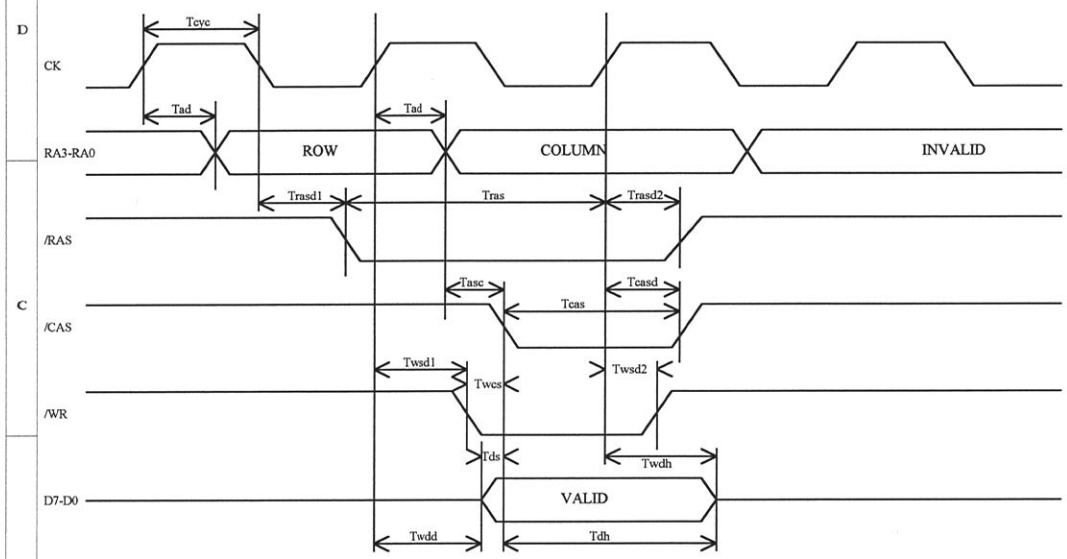

A[0..15]
A[0..15]
J19
SIMMSOCKET
D[0..7]
D[0..7]
RAS
CASL
WRL
D[0..7]
CS6
RESET
WRL
CK
RD
U10C
74HC32
U12A
74HC74
VCC
U4C
74HC04
U10D
74HC32
U16
74HC374
AD[0..7]
U11
74HC244
Title
DRAM and NIC Registers
Size
A4
Number
Revision
Date: 20-Oct-1997
File: C:\THESIS\SCH\RAM.SCH
Sheet of
Drawn By: Nicholas Ryan


CK
ACK
RD
WRL
MRD
MWR
PWR
PRD
DACK1
PRQ
ADS0
U4E
IRQ0
74HC04
U1
DP83902
INT
W/ACK
PRD
PWR
MWR
MRD
SWR
SRD
ACK
BSCK
RACK
READY
PRQ/ADS1
ADS0
BACK
BREQ
AD[0..7]
AD[8..15]
A[0..15]
RESET
CS6
AD0
AD1
AD2
AD3
AD4
AD5
AD6
AD7
AD8
AD9
AD10
AD11
AD12
AD13
AD14
AD15
A3
A2
A1
A0
RA3
RA2
RA1
RA0
RESET
CS
SNISEL
TEST
X1
GND/X2
20MHz
SEL
AUI/TPI
GDLNK/LNKDIS
RXI-
RXI+
TXOd+
TXO-
TXO+
TXOd-
CD+
CD-
RX+
RX-
TX+
TX-
COL
POL
TXE/TX
CRS/RX
T1
TXO+
Tct
TXO-
RXI+
Rct
RXI-
TD+
Tct
TD-
RD+
Rct
RD-
PE64108
R1
270R
R2
270R
R3
39R
R4
39R
R5
39R
R6
39R
C7
10nF
C8
10nF
+12V
J1
DB15
VCC
C11
22pF
C12
22pF
X1
20MHz
J5
JUMPER
R19
4K7
L3
GreenLED
R16
300R
L1
YellowLED
R14
300R
U2B
74ALS02
U2C
74ALS02
Title
Ethernet Controller
Size
A4
Number
Revision
Date:
20-Oct-1997
Sheet of
File:
C:\THESIS\SCH\ETHERNET.SCH
Drawn By:
Nicholas Ryan


U10B
74HC32
WRL
U9
74HC374
D[0..7]
PWR
U13
74HC374
D[8..15]
U14
74HC374
U15
74HC374
PRD
U10A
74HC32
DACK1
RD
U5
74HC373
U6
74HC373
U7
6164
U8
6164
VCC
R23 4K7
R24 4K7
R22 4K7
MWR
MRD
ADS0
AD[0..7]
AD[8..15]
Title
Buffer Memory
Size
A4
Number
Revision
Date: 20-Oct-1997
File: C:\THESIS\SCH\BUFFER.SCH
Sheet of
Drawn By: Nicholas Ryan

APPENDIX B : Timing Diagrams





Bus cycle time, Tcyc = 50nS
Address delay time, Tad = 20nS max
/RAS delay time 1, Trasd1 = 20nS max
/RAS delay time 2, Trasd2 = 30nS max
Column address setup time, Tasc = 0nS max
/CAS delay time, Tcasd = 20nS max
Write strobe delay 1, Twsd1 = 20nS max
Write strobe delay 2, Twsd2 = 20nS max
Write command setup time, Twcs = 0nS min
Write data delay time, Twdd = 20nS max
Write data hold time, Twdh = 0nS min

Requirements:

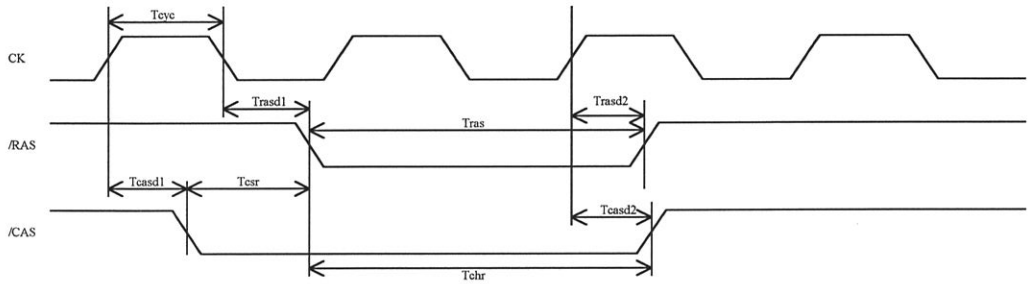
/RAS pulse width, Tras = 70nS min
= 1.5 x Tcyc + Trasd2 - Trasd1
= 85nS
OK

/CAS pulse width, Tcas = 20nS min
= Tcyc + Tcasd - Tad - Tasc
= 50nS
OK

Data setup time, Tds = 0nS min
= Tad + Tasc - Twdd2
= 0nS
OK

Data hold time, Tdh = 15nS min

Title: **DRAM Write Timing Diagram**		
Size: A4	Number:	Revision:
Date: 20-Oct-1997		Sheet of
File: C:\THESIS\SCH\T-DRAMWR.SCH		Drawn By: Nicholas Ryan



Bus cycle time, Tcyc = 50nS
/RAS delay time 1, Trasd1 = 20nS max
/RAS delay time 2, Trasd2 = 30nS max
/CAS delay time 1, Tcasd1 = 20nS max
/CAS delay time 2, Tcasd2 = 20nS max

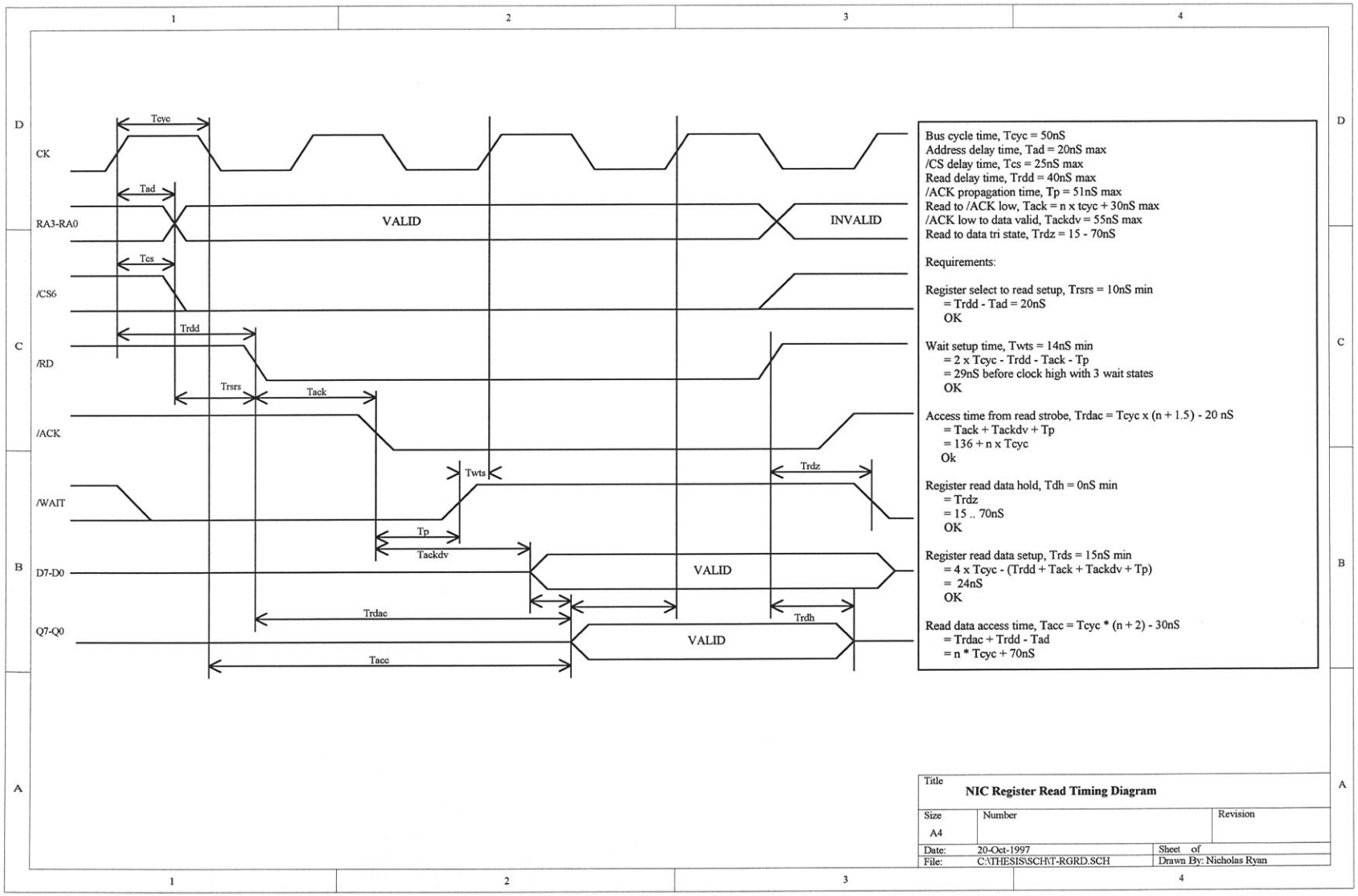
Requirements:

/CAS refresh setup time, Tcsr = 5nS min
= 10nS (From Sh7032)
OK

/CAS hold time for refresh, Tchr = 30nS min
= 1.5 x Tcyc - Trasd1 + Tcasd2
= 75nS
OK

/RAS pulse width, Tras 70nS min
= 1.5 x Tcyc + Trasd1 - Trasd2
= 65nS => Insert a wait state

Title	**/CAS-Before-/RAS DRAM Refresh Timing Diagram**	
Size A4	Number	Revision
Date: 20-Oct-1997		Sheet of
File: C:\THESIS\SCH\T-DRAMRF.SCH		Drawn By: Nicholas Ryan



Bus cycle time, Tcyc = 50nS
Address delay time, Tad = 20nS max
/CS delay time, Tcs = 25nS max
Read delay time, Trdd = 40nS max
/ACK propagation time, Tp = 51nS max
Read to /ACK low, Tack = n x tcyc + 30nS max
/ACK low to data valid, Tackdv = 55nS max
Read to data tri state, Trdz = 15 - 70nS

Requirements:

Register select to read setup, Trsrs = 10nS min
= Trdd - Tad = 20nS
OK

Wait setup time, Twts = 14nS min
= 2 x Tcyc - Trdd - Tack - Tp
= 29nS before clock high with 3 wait states
OK

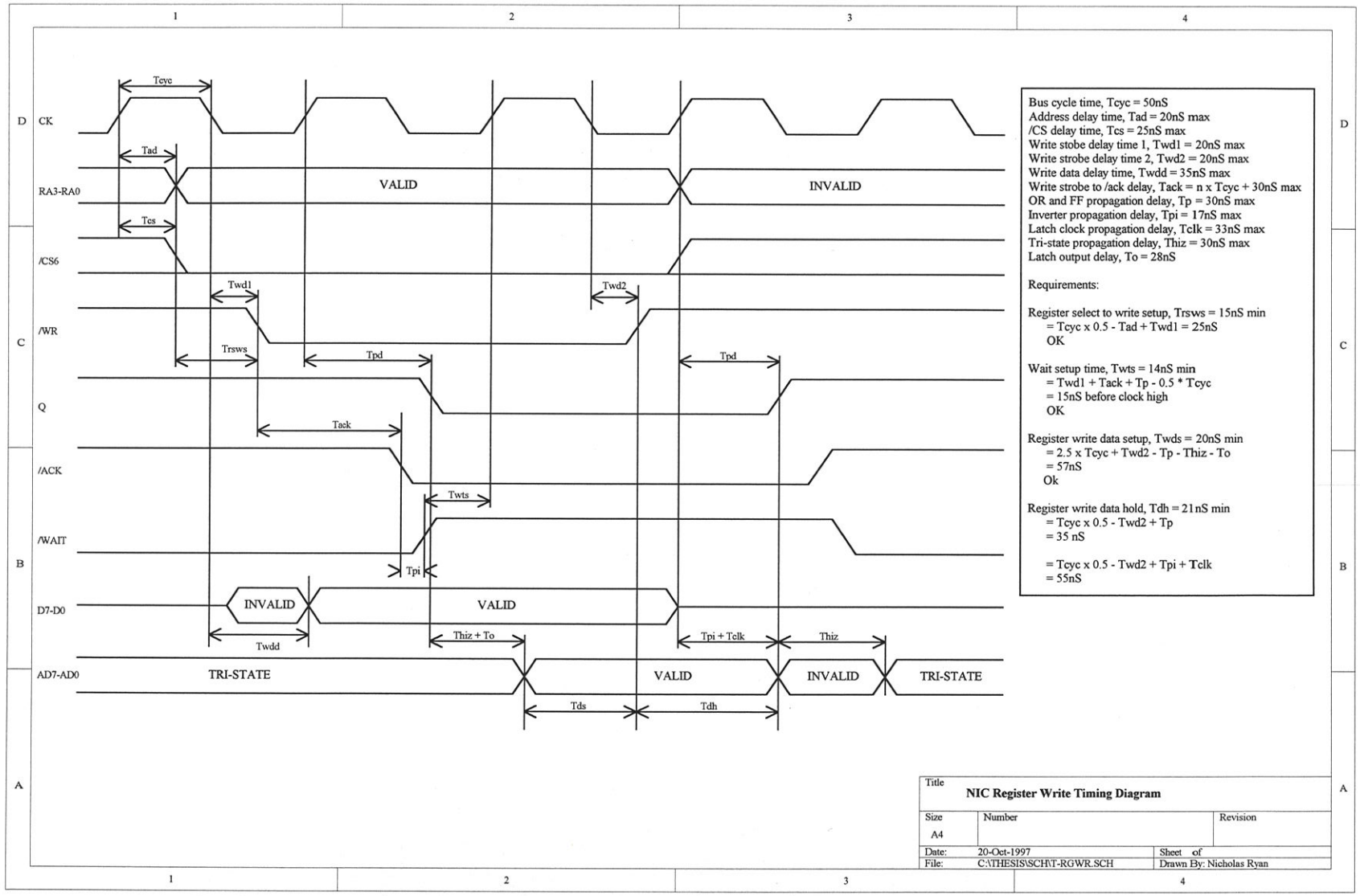
Access time from read strobe, Trdac = Tcyc x (n + 1.5) - 20 nS
= Tack + Tackdv + Tp
= 136 + n x Tcyc
Ok

Register read data hold, Tdh = 0nS min
= Trdz
= 15 .. 70nS
OK

Register read data setup, Trds = 15nS min
= 4 x Tcyc - (Trdd + Tack + Tackdv + Tp)
= 24nS
OK

Read data access time, Tacc = Tcyc * (n + 2) - 30nS
= Trdac + Trdd - Tad
= n * Tcyc + 70nS

Title: **NIC Register Read Timing Diagram**		
Size: A4	Number:	Revision:
Date: 20-Oct-1997		Sheet of
File: C:\THESIS\SCH\T-RGRD.SCH		Drawn By: Nicholas Ryan



Bus cycle time, Tcyc = 50nS
Address delay time, Tad = 20nS max
/CS delay time, Tcs = 25nS max
Write stobe delay time 1, Twd1 = 20nS max
Write strobe delay time 2, Twd2 = 20nS max
Write data delay time, Twdd = 35nS max
Write strobe to /ack delay, Tack = n x Tcyc + 30nS max
OR and FF propagation delay, Tp = 30nS max
Inverter propagation delay, Tpi = 17nS max
Latch clock propagation delay, Tclk = 33nS max
Tri-state propagation delay, Thiz = 30nS max
Latch output delay, To = 28nS

Requirements:

Register select to write setup, Trsws = 15nS min
= Tcyc x 0.5 - Tad + Twd1 = 25nS
OK

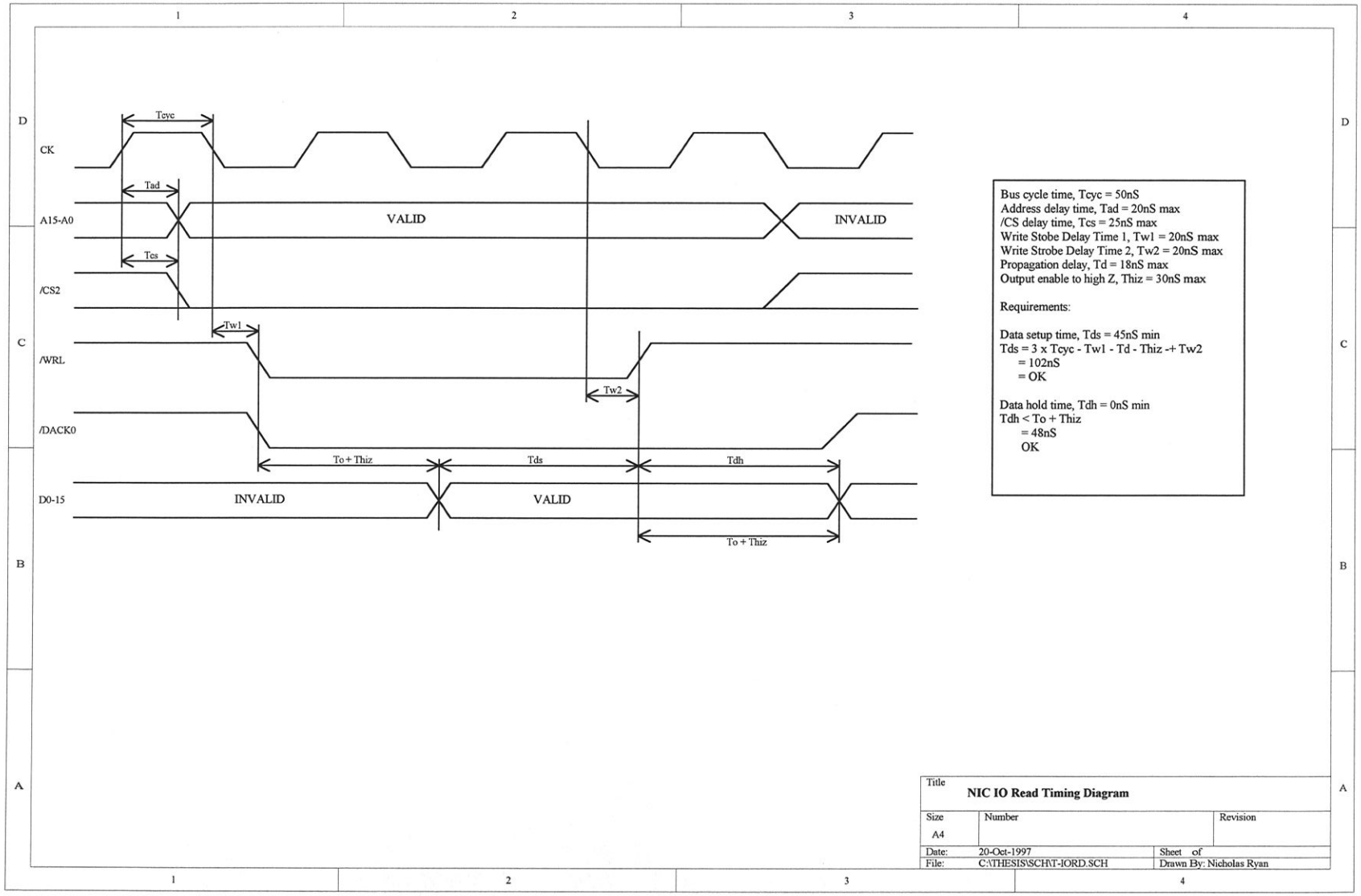
Wait setup time, Twts = 14nS min
= Twd1 + Tack + Tp - 0.5 * Tcyc
= 15nS before clock high
OK

Register write data setup, Twds = 20nS min
= 2.5 x Tcyc + Twd2 - Tp - Thiz - To
= 57nS
Ok

Register write data hold, Tdh = 21nS min
= Tcyc x 0.5 - Twd2 + Tp
= 35 nS

= Tcyc x 0.5 - Twd2 + Tpi + Tclk
= 55nS

Title: **NIC Register Write Timing Diagram**		
Size: A4	Number:	Revision:
Date: 20-Oct-1997		Sheet of
File: C:\THESIS\SCH\T-RGWR.SCH		Drawn By: Nicholas Ryan



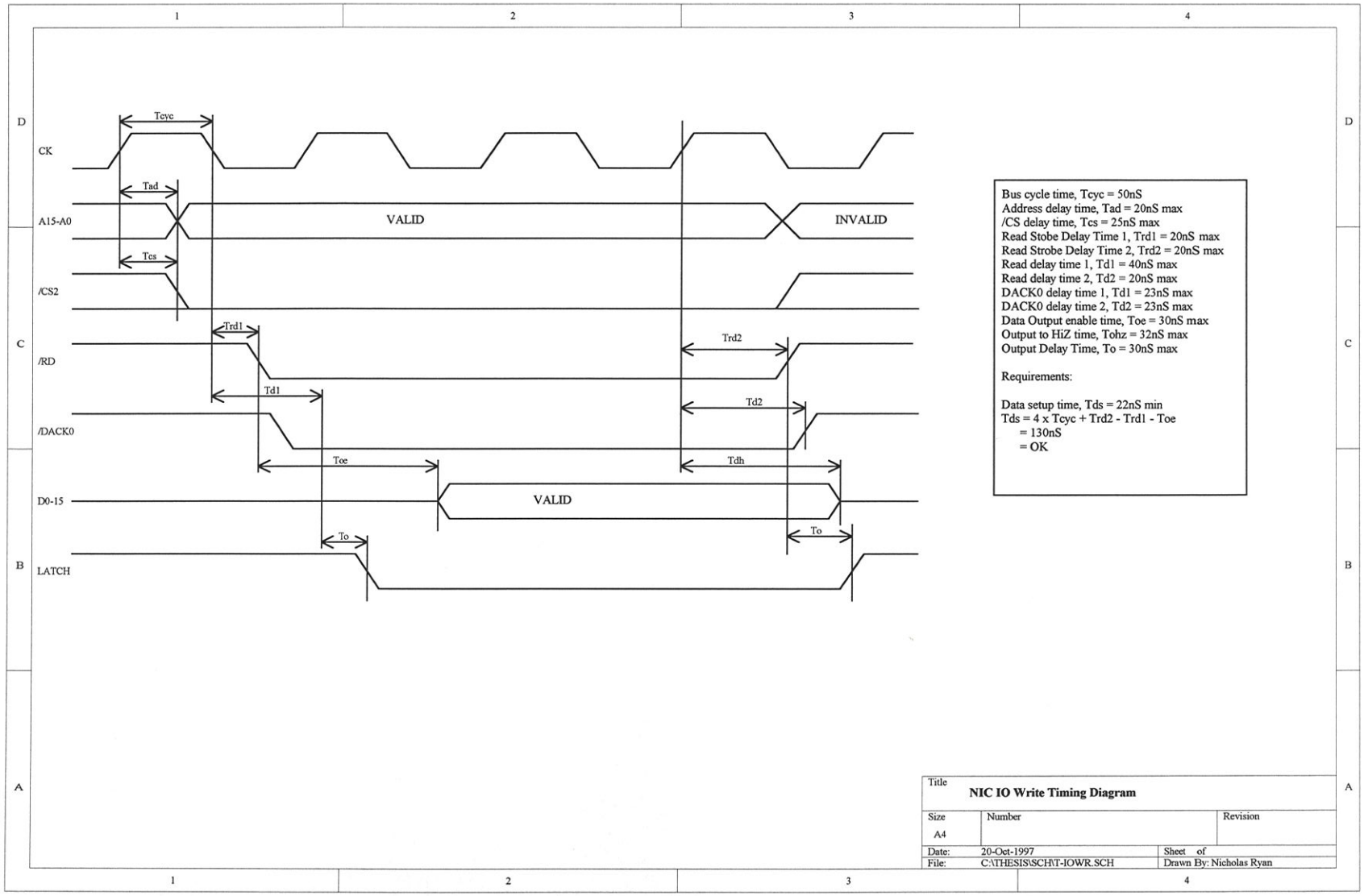
Bus cycle time, Tcyc = 50nS
Address delay time, Tad = 20nS max
/CS delay time, Tcs = 25nS max
Write Stobe Delay Time 1, Tw1 = 20nS max
Write Strobe Delay Time 2, Tw2 = 20nS max
Propagation delay, Td = 18nS max
Output enable to high Z, Thiz = 30nS max

Requirements:

Data setup time, Tds = 45nS min
Tds = 3 x Tcyc - Tw1 - Td - Thiz -+ Tw2
= 102nS
= OK

Data hold time, Tdh = 0nS min
Tdh < To + Thiz
= 48nS
OK

Title	**NIC IO Read Timing Diagram**		
Size A4	Number		Revision
Date:	20-Oct-1997	Sheet of	
File:	C:\THESIS\SCH\T-IORD.SCH	Drawn By: Nicholas Ryan	



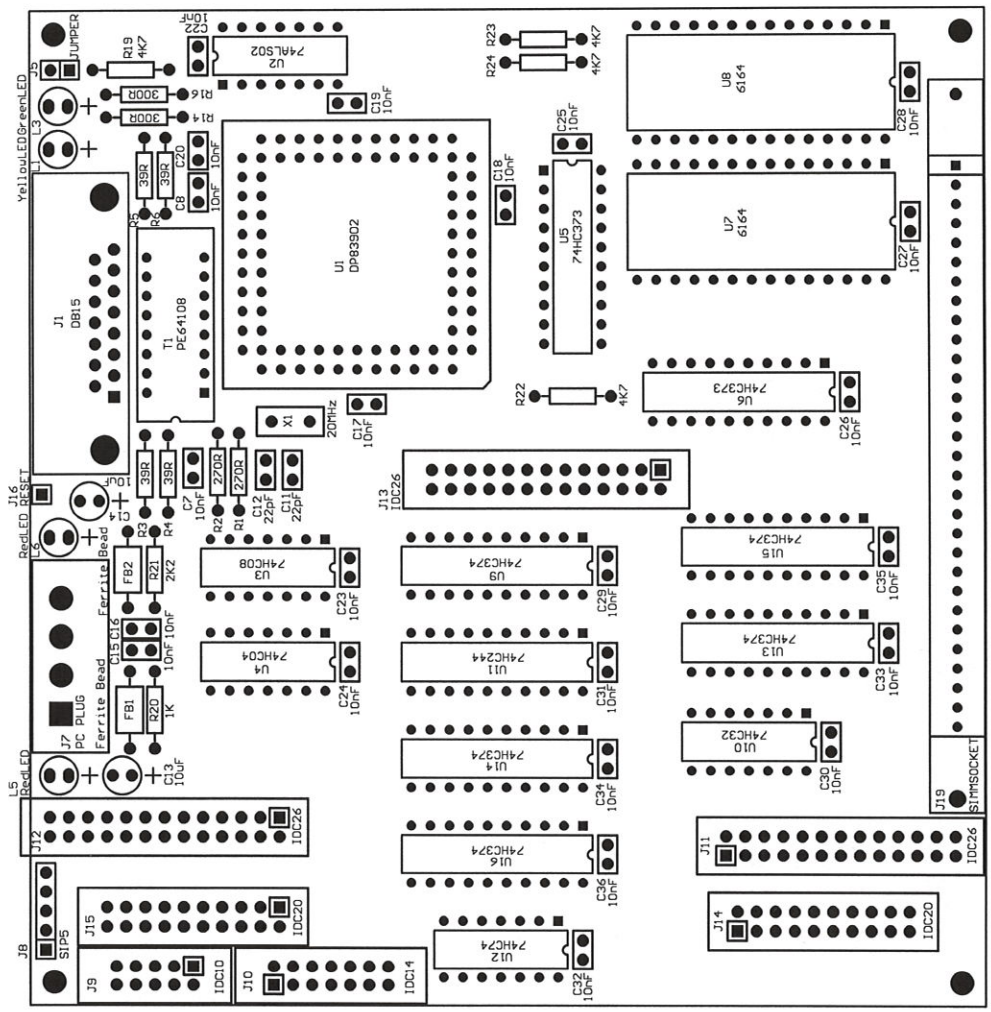
Bus cycle time, Tcyc = 50nS
Address delay time, Tad = 20nS max
/CS delay time, Tcs = 25nS max
Read Stobe Delay Time 1, Trd1 = 20nS max
Read Strobe Delay Time 2, Trd2 = 20nS max
Read delay time 1, Td1 = 40nS max
Read delay time 2, Td2 = 20nS max
DACK0 delay time 1, Td1 = 23nS max
DACK0 delay time 2, Td2 = 23nS max
Data Output enable time, Toe = 30nS max
Output to HiZ time, Tohz = 32nS max
Output Delay Time, To = 30nS max

Requirements:

Data setup time, Tds = 22nS min
Tds = 4 x Tcyc + Trd2 - Trd1 - Toe
= 130nS
= OK

Title	**NIC IO Write Timing Diagram**	
Size A4	Number	Revision
Date: 20-Oct-1997		Sheet of
File: C:\THESIS\SCH\T-IOWR.SCH		Drawn By: Nicholas Ryan

APPENDIX C : PCB Layout



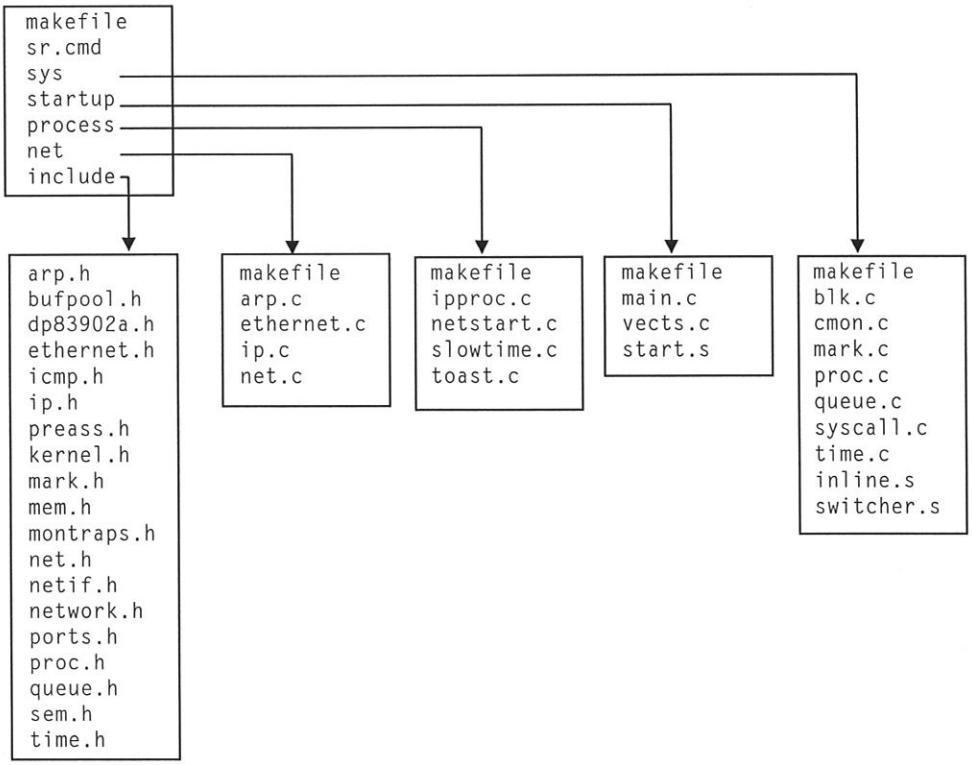
APPENDIX D : Bill of Materials

Quantity	Reference	Type	Description
23	C7, C8, C15 - C20, C22 - C36	Cap	10nF, 100V, Mon. Cer.
2	C11, C12	Cap	22pF, 100V, Mon. Cer
2	C13, C14	Cap	10uF, 63V, Electro.
2	FB1, FB2	Ind	Axial Ferrite Bead
1	J1	Conn	DB15 socket, R/A mount
1	J5	Conn	2 Pin Header
1	J7	Conn	PC Plug, R/A mount
1	J8	Conn	5 Pin Header
1	J9	Conn	IDC10 Boxed Header
1	J10	Conn	IDC14 Boxed Header
3	J11, J12, J13	Conn	IDC26 PCB Transition
2	J14, J15	Conn	IDC20 PCB Transition
1	J19	Conn	30pin SIMM Socket, Vert.
1	L1	LED	Yellow, 3mm
1	L3	LED	Green, 3mm
2	L5, L6	LED	Red, 3mm
2	R1, R2	Res	270R, 5%
4	R3, R4, R5, R6	Res	39R, 5%
2	R14, R16	Res	300R, 5%
1	R20	Res	1K, 5%
1	R21	Res	2K2, 5%
4	R19, R22, R23, R24	Res	4K7, 5%
1	T1	Trans	PE64108 Pulse Transformer
1	U1	IC	DP83902A NIC, 84pin PLCC
1	U2	IC	74ALS02
1	U3	IC	74HC08
1	U4	IC	74HC04
2	U5, U6	IC	74HC373
2	U7, U8	IC	6164 8K x 8 SRAM
1	U10	IC	74HC32
1	U11	IC	74HC244
1	U12	IC	74HC74
5	U9, U13, U14, U15, U16	IC	74HC374
2	X1	XTAL	20.000000MHz

Miscellaneous Parts	
1	84pin PLCC socket
5	14pin DIP socket
8	20pin DIP socket
2	28pin DIP socket
1	1MB 70nS DRAM SIMM
3	IDC26 socket
2	IDC20 socket
2	IDC20 Boxed header
3	IDC26 Boxed header
1	1m x 26way IDC cable

APPENDIX E : Software Hierarchy

The source code developed for this project may be found on the accompanying 3 ½ inch floppy disk included at the back of this thesis. The file structure follows the hierarchy shown below.



REFERENCES

[1] Apple Computer, Inc. 1997 *MessagePad 2000 Datasheet* [Online]. Available http://www.newton.apple.com/product_info/devices/MP2000/MP2000ds.html

[2] Benham, D. 1991 *10Base-T Solutions: Network Interface Adapter, System Brief 119* [Online]. Available http://www.national.com/ms/SB/SB-119.pdf

[3] Casio Computer Co., Ltd., *Press Release* [Online]. Available www.casiohpc.com/low/pressrelease.html

[4] Comer, D. 1984 *Operating System Design: The XINU Approach*, Prentice Hall, New Jersey.

[5] Comer, D. 1987 *Operating System Design – Volume II: Internetworking with XINU*, Prentice Hall, New Jersey.

[6] Comer, D.E. & Stevens, D.L. 1994 *Internetworking with TCP/IP, vol. 2*, Prentice Hall, New Jersey.

[7] Fielding, R., UC Irvine, Gettys, J., Mogul, J., DEC, Frystyk, H. Berners-Lee, T. & MIT/LCS 1997 *Hypertext Transfer Protocol, RFC: 2068* [Online]. Available http://andrew2.andrew.cmu.edu/rfc/rfc2069.html

[8] Hitachi America Ltd. 1996 *Hitachi Single-Chip RISC Microcomputer SH7032 and SH7034 Hardware Manual* [Online]. Available http://www.halsp.hitachi.com/tech_prod/h_micron/1_sh/1_sh1/h1101/pdf/h1101.pdf

[9] Hitachi America Ltd. 1997 *SH-1 Low-Cost Evaluation Board US7032EVB1 User's Manual*, Hitachi America Ltd.

[10] Hitachi America Ltd. 1996 *SuperH RISC Engine SH-1/SH-2 Programming Manual* [Online]. Available http://www.halsp.hitachi.com/tech_prod /h_micon/1_sh/2_sh2/h12p0/pdf/h12p0.pdf

[11] Horowitz, P. & Hill, W. 1993 *The Art of Electronics*, Cambridge University Press, New York.

[12] Information Sciences Institute 1981 *Internet Protocol, RFC: 791* [Online]. Available http://andrew2.andrew.cmu.edu/rfc/rfc791.html

[13] Information Sciences Institute 1981 *Transmission Control Protocol, RFC: 793* [Online]. Available http://andrew2.andrew.cmu.edu/rfc/rfc793.html

[14] Intel Corporation 1991 *1M Dynamic RAM with Fast Page Mode Datasheet*, Intel Corporation.

[15] National Semiconductor Corporation 1993 *Writing Drivers for the DP8390 NIC Family of Ethernet Controllers, Application Note 874* [Online]. Available http://www.national.com/an/AN/AN-874.pdf

[16] National Semiconductor Corporation 1995 *DP83902A ST-NIC Serial Network Interface Controller for Twisted Pair* [Online]. Available http://www. national.com/ds/DP/DP/DP83902A.pdf

[17] Newsbytes News Network 1997 *Embedded Conference – Java Perks Coffee Over Web* [Online]. Available newsgroup: biz.clarinet.

[18] O'Connell, M 1995 *Java: The Inside Story* [Online]. Available http://www. sun.com/sunworldonline/swol-07-1995/swol-07-java.html

[19] Pacific Softworks Inc. 1997 *Managing Network Devices with Web Browsers* [Online]. Available http://www.pacificsw.com/Background.html

[20] QNX Operating Systems Ltd. 1996 *QNX Operating System: System Architecture*, Ontario.

[21] Runkel, M. A. 1997 *Ethernet Frequently Asked Questions (FAQ)* [Online]. Available http://www.ots.utexas.edu/ethernet/enet-faqs/ethernet-faq

[22] Schofield, M.J. 1996 *Controller Area Network – Background Information* [Online]. Available http://www.can-cia.de/caninfo.html

[23] SEEQ Technology Inc, 1992 *8023A Manchester Code Converter Datasheet*, SEEQ Technology Inc.

[24] Socolofsky, T., Kale, C. & Spider Systems Ltd. 1991 *A TCP/IP Tutorial, RFC: 1180* [Online]. Available http://andrew2.andrew.cmu.edu/rfc/rfc1180.html

[25] Stevens, W.R. 1990 *UNIX Network Programming*, Prentice-Hall, New Jersey.

[26] Valentino, R. 1997 *comp.sys.ibm.pc.hardware.* Frequently Asked Questions (FAQ) Part 4/5* [Online]. Available http://zebra.herston.uq.edu.au/tec/pcfaq/part4.htm

[27] Von Voros, J. 1993 *Architectural Choices for Network Performance, Application Note 873* [Online]. Available http://www.national.com/an/AN/AN-873.pdf

[28] W3 Consortium 1997 *WWW Project History* [Online] Available www.pku.edu.cn\on_line\w3html\History.html

[29] Wakeman, L. 1993 *The Design and Operation of a Low Cost, 8-Bit PC-XT Compatible Ethernet Adapter Using the DP83902, Application Note 942* [Online]. Available http://www.national.com/an/AN/AN-942.pdf

[30] Wilson, B. 1994 *Loopback Diagnostics Using the DP8390/901/902/905, Application Note 937* [Online]. Available http://www.national.com/an/AN/AN-937.pdf