DESIGN AND DEVELOPMENT OF A REMOTE RECONFIGURABLE INTERNET EMBEDDED I/O CONTROLLER

By

Grant Phillips

submitted in complete fulfillment of the degree:

MAGISTER TECHNOLOGIAE:
ELECTRICAL ENGINEERING

in the

Faculty of Electrical and Mechanical Engineering,
at the
Port Elizabeth Technikon

Promoter:
Frank Adlam

January 2003

I, Grant Phillips, hereby declare that:

- the work done in this thesis is my own;
- all sources used or referred to have been documented and recognized; and
- this thesis has not been previously submitted in full or partial fulfilment of the requirements for an equivalent or higher qualification at any other educational institution.

Signature **Date: 14/01/2003**

ACKNOWLEDGEMENTS

- I would like to thank Jeremy Bentham for the knowledge he shared with me via emails and through his book. The work he has done in the field of embedded Internet systems was the driving force behind my project.

- My sincere appreciation to the Altera Corporation for donating a MAX7000S development kit. This was extremely helpful in learning about CPLDs and reduced my developing time drastically.

- Sincere thanks to my mentor, Frank Adlam, for his support, motivation and patience during the development of this project.

- My deepest thanks also go to my family and fiancé for their support and encouragement throughout this project.

- Lastly I would like to thank the Lord Jesus Christ for giving me the strength and dedication to finish the project to the best of my ability.

ABSTRACT

The use of embedded Internet systems is growing rapidly in the manufacturing sector. These systems allow the monitoring and controlling of plant machinery and manufactured items from a remote location via a standard Web interface.

In a manufacturing environment, it is inevitable that long running processes will require support for dynamic reconfiguration because, for example, machines may fail, services may be moved or withdrawn and user requirements may change. In such an environment it is essential that the operation and architecture of such processes can be modified to reflect such changes.

This research project will present methods and ideas for establishing a reconfigurable remote system by using standard 8-bit microcontrollers and reconfigurable hardware. It will allow a manufacturing process to be modified and changed within minutes without even having to be physically present at the location where the process is running.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AIM	Avalanche-Induced Migration
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuits
BCD	Binary-Coded Decimal
BOR	Brown-out Reset
BST	Boundary-scan Test
CGI	Common Gateway Interface
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex PLD
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense and Multiple Access With Collision Detection
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DNS	Domain Name Service
EEPROM	Electrically Erasable Programmable Read-Only Memory
EGI	Embedded Gateway Interface
ESD	Electrostatic Discharge
FIFO	First-In/First-Out
FPLA	Field Programmable Logic Array
FPLS	Field Programmable Logic Sequence
FTP	File Transfer Protocol
GAL	Generic Array Logic
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
I^2C	Inter-IC
ICD	In-Circuit Debugger
ICMP	Internet Control Message Protocol
ICSP	In-Circuit Serial Programming

IDE	Integrated Development Environment
IP	Internet Protocol
ISA	Industry Standard Architecture
ISO	International Standards Organization
ISOC	Internet Society
ISP	In-System Programming
JTAG	Joint Action Test Group
Kb	Kilobyte
LAB	Logic Array Block
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
LSB	Least Significant Bit
LSI	Large-Scale Integration
LVP	Low-Voltage Programming
MAC	Media Access Control
MAX	Multiple Array Matrix
MCU	Microcontroller Unit
MOS	Metal-Oxide Semiconductor
MSB	Most Significant Byte
MSI	Medium-Scale Integration
NFS	Network File System
NIC	Network Interface Controller
OSI	Open System Interconnection
OST	Oscillator Start-up Timer
PAL	Programmable Array Logic
PCB	Printed Circuit Board
PIA	Programmable Interconnection Array
PLD	Programmable Logic Devices
POR	Power-On Reset
PWM	Pulse Width Modulation
PWRT	Power-Up Timer
RAM	Random Access Memory
RARP	Reverse Address Resolution Protocol

RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTC	Real Time Clock
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SPI	Serial Peripheral Interface
SSI	Small-Scale Integrations
SSP	Synchronous Serial Port
TAP	Test Access Port
TCP	Transport Control Protocol
TTL	Transistor Transistor Logic
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
USART	Universal Synchronous Asynchronous Receiver Transmitter
UTP	Untwisted Pair
WAN	Wide Area Network
WDT	Watchdog Timer
WWW	World Wide Web

CHAPTER 1
INTRODUCTION

Industry analysts see embedded Internet systems as poised for rapid growth in the manufacturing sector in the next few years. Embedded Internet systems allow users to communicate directly with items such as plant, machinery and manufactured components delivered to site. This communication may range from passive remote monitoring to direct control or resetting of the remote system. Although it seems that these systems are almost perfect, they do however lack one important component: reconfigurability.

1.1 PROBLEM STATEMENT

The rapid growth of embedded Internet applications such as data logging and controlling requires remote online maintenance and system upgrade capability to enable flexible system re-configurations. This leads to the following sub-problems in current embedded Internet systems:

- To modify the controlling software on the remote application, the system must physically be disconnected from the network to reprogram the device in the field.
- Modifying the Input/Output (I/O) interfacing hardware of the system normally leads to the redesign of the I/O hardware printed circuit board (PCB).
- Specialised technical staff often has to travel long distances to do maintenance and system upgrades on remote systems.
- Existing systems are limited in terms of flexibility.

1.2 HYPOTHESIS

A remote configurable/programmable embedded Internet system will be developed to allow flexible system configurations in industry. The system can be divided into the following main components (refer to Figure 1.1):

- An embedded Internet application board with the following components:
- ▪ Communications processor for connecting the embedded system to the Internet and for managing the TCP/IP stack.

- I/O processor which is the main device controlling the input and output of the embedded system.
- Reconfigurable hardware which can be adapted to the needs of the user for the specific application.
- A programming processor which will be the heart of the whole reconfigurability aspect of the embedded Internet application.
- Embedded server software to allow data communication between the PC and the remote embedded system via Ethernet.
- Client software for a standard PC to allow the following functions:
- Transfer new program files for I/O processors, the web server and the reconfigurable hardware to the remote system.
- Instruct the programming processor to reprogram the I/O processor, the web server and reconfigurable hardware.
- Allow the user to send and receive data from the remote Internet application and thereby also to control the application.

1.3 DELIMITATION OF PROJECT

- Computer literate users will operate the client software and should have the necessary technical background of the system used in the particular remote application.

- Programming files for the particular reprogrammable/reconfigurable components will be developed using third party development tools. The client software will only act as a medium to transfer compiled programs to the remote Internet application as indicated in Figure 1.1 on the next page.

- The programming routines implemented in the programming processor for the reprogrammable/reconfigurable components, will be limited to a specific manufacturer for this project.

- TCP/IP is the main network protocol that will be investigated in this project, although the implementation of other protocols may be used.

1.4 OUTLINE OF THE SYSTEM DEVELOPED

Figure 1.1 illustrates a system diagram indicating the project outline.

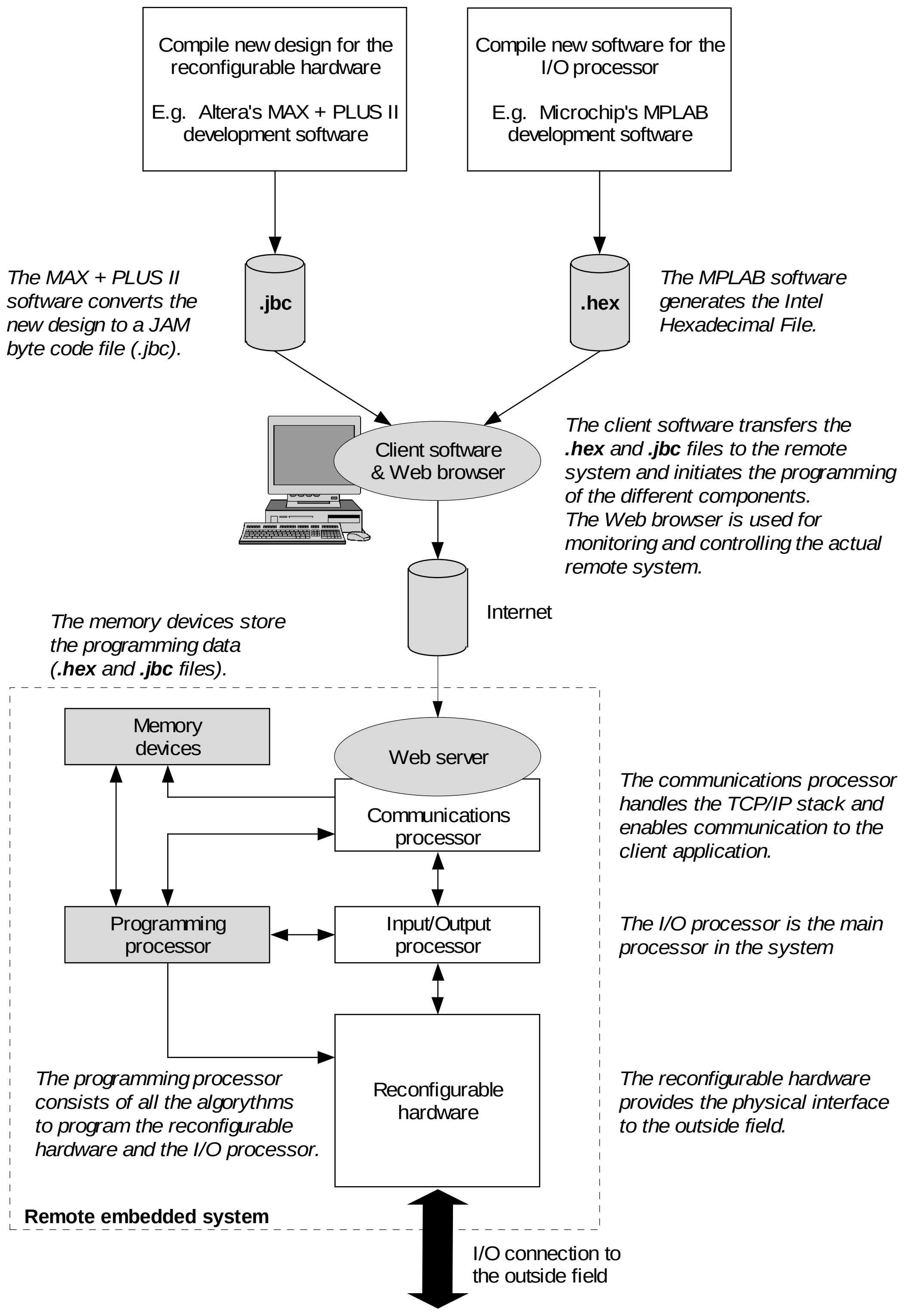

Compile new design for the reconfigurable hardware
E.g. Altera's MAX + PLUS II development software
Compile new software for the I/O processor
E.g. Microchip's MPLAB development software
The MAX + PLUS II software converts the new design to a JAM byte code file (.jbc).
.jbc
.hex
The MPLAB software generates the Intel Hexadecimal File.
Client software & Web browser
*The client software transfers the **.hex** and **.jbc** files to the remote system and initiates the programming of the different components. The Web browser is used for monitoring and controlling the actual remote system.*
Internet
*The memory devices store the programming data (**.hex** and **.jbc** files).*
Memory devices
Web server
Communications processor
The communications processor handles the TCP/IP stack and enables communication to the client application.
Programming processor
Input/Output processor
The I/O processor is the main processor in the system
Reconfigurable hardware
The programming processor consists of all the algorythms to program the reconfigurable hardware and the I/O processor.
The reconfigurable hardware provides the physical interface to the outside field.
Remote embedded system
I/O connection to the outside field

Figure 1.1 The project outline.

1.5 SIGNIFICANCE OF THE PROJECT

The remote configurable/programmable embedded Internet system will be more flexible than similar existing systems. This is advantageous, for example, in the scenario where a manufacturer produces a product for a period of time and then needs to change the system configuration in order to use the same system to produce another product. Instead of sending a maintenance team to physically change the hardware at the remote system, the user will now be able to do this remotely from a PC via the Internet. It also reduces maintenance cost and time significantly, especially if the system is located in a different town where travelling becomes a factor.

Embedded software for the I/O processor and reconfigurable hardware can also be changed remotely without having to remove the processor or memory from the system board and thus reduces downtime.

1.6 REVIEW OF RELEVANT LITERATURE

The I/O processor must be able to be programmed while it is still in the circuit in order to make the remote Internet application flexible. In-System Programming (ISP) is a technique where a programmable device is programmed after the device is placed in a circuit board. In-Circuit Serial Programming (ICSP) is an enhanced ISP technique and uses only two I/O pins to serially input and output data which makes ICSP easy to use and less intrusive on the normal operation of the microcontroller unit (MCU).

ICSP can be used in a variety of ways:

- *Reduce cost of field upgrades*
- *Add unique ID code to your system during manufacturing*
- *Calibrate your system in the field*
- *Customize and configure your system in the field*

[Microchip: In-circuit serial programming guide, 2001]

Some kind of hardware must be implemented to make the interface to the outside field flexible. Programmable logic components are devices which are able to implement a wide variety of logic functions, both combinational and sequential. The actual logic function implemented is determined by the user, using some form of design entry software to

specify the state of the internal programmable points. The internal programmable points, effectively are, switches which can be selected to be either closed or open.
[Seals et al, 1998]

In-system programmability in Altera's programmable logic devices (PLDs) makes prototyping easy during design development, streamlines production, increases design flexibility and allows quick and efficient in-field upgrades. ISP uses the IEEE Std. 1149.1 Joint Action Test Group (JTAG) test port, which allows devices to be programmed and the PCB to be functionally tested in a single manufacturing step.

The Jam programming and test language, compatible with all current PLDs that offer ISP, provides a software-level standard for in-system programming by remaining vendor-independent, which results in small file sizes and reduces programming times. Designers, using the Jam language to implement ISP, can lengthen the life and enhance the quality and flexibility of the end product
[Altera: Introduction to ISP, 1999].

Until now, the Internet has almost exclusively served the computer market. With the introduction of IPv6 (a new Internet communication standard), the Internet is likely to serve very different needs in construction, each with a new set of requirements. Nomadic computing, through a variety of network attachments including radio frequency wireless networks and infrared systems, will make on-site communication more viable. But perhaps more significantly will be the possibility of remote device controls – more specifically embedded systems. Remote access is enabling cost savings in development, commissioning, use, and maintenance and is also the major driving force for transferring Internet mass technologies from the office and commercial sectors to industrial applications [Finch, 1998].

A certain protocol must be used to establish communication between the remote server and the client application software. TCP/IP is a family of protocols used for computer communications. Often a TCP/IP user does not use the TCP/IP protocol itself, but some other protocol from the family. This protocol family includes the following:

- *IP (Internet Protocol)*
- *ARP (Address Resolution Protocol)*

- *ICMP (Internet Control Message Protocol)*
- *UDP (User Datagram Protocol)*
- *TCP (Transport Control Protocol)*
- *RIP (Routing Information Protocol)*
- *SMTP (Simple Mail Transfer Protocol)*

[Arnett, 1994]

1.7 ORGANIZATION OF THESIS

The chapters are organized logically as far as possible. The first two chapters will provide all the background information needed to fully understand the project. Thereafter the chapters will introduce the different project building blocks individually and leads to the last chapter which covers the systems test results and conclusions. The different building blocks for the project are as follows:

- The Ethernet interface
- A Web server
- The Input/Output controller
- Reconfigurable hardware
- The Embedded programming interface
- The Software environment

Here is a summary of the layout for the rest of the thesis:

Chapter 2: A background of the origin of computer networks. Detailed description of the different network layers is also given which is of vital importance to understand the rest of the thesis.

Chapter 3: To start explaining the building blocks of the project. This chapter builds further on the knowledge of basic networks and examines the possibility of using embedded controllers to implement Ethernet connectivity, highlighting some possible limitations and solutions.

Chapter 4: Gives a detailed explanation of how an embedded controller is implemented as the web server in this system.

Chapter 5: Explains the operation of the actual input/output controlling processor and inter-processor protocols that have to be implemented. The in-circuit programming process for this processor will also be covered.

Chapter 6: Introduces re-configurable hardware in general, and complex PLDs (CPLDs) more specifically. Altera’s product range is then introduced and reasons for choosing this vendor is highlighted. In-circuit programming in CPLDs is explained, followed by a detailed description of the reconfigurable hardware interface.

Chapter 7: Covers a detailed description and study of the heart of the whole system – the actual programming processor.

Chapter 8: Explains how the client software transfers new programming files to the remote system and gives instructions to program the different components of the system. It also shows how any web browser can be used to access the remote embedded web server.

Chapter 9: Indicates the test results after the whole system was completed by testing it with three different applications which were developed and programmed from a remote location. Some future development areas will also be highlighted and the thesis will end with a conclusion of the whole project.

CHAPTER 2
THE INTERNET AND TCP/IP

2.1 OVERVIEW

The chapter starts with a brief history of the Internet and explains the whole concept of internetworking. The rest of the chapter is dedicated to a description of some of the relevant protocols in the TCP/IP suite which is crucial for understanding the networking section of the project.

2.2 INTERNETWORKING EVOLUTION

Twenty years ago state of the art data communications involved transmitting command and data at 2400 bits per second or less (Wilder, 1999, p.1). This was done using analog telephone lines with a remote host to process the data using an application program. To send data to another terminal (computer) in the same city (or sometimes even the same building) meant that the data first had to be sent to a remote host and then only relayed to the other terminal. The process was termed terminal-to-terminal communication. Problems started arising due to the fact that the two terminals might have been in the same city, but then the host was in another city, causing data flow to be very slow. The matter was worsened by the fact that the communications protocols were proprietary and meant that a terminal manufactured by “brand A” could only communicate with another “brand A” terminal.

Then, in the early 1970s, Bob Medcalfe invented Ethernet which was a local area network (LAN) technology (Wilder, 1999, p.1). This technology eliminated the need for a host in a terminal-to-terminal environment and raised the speed of data transfer by a 5000 times. A few minor modifications were made to this LAN technology and was made into a standard called IEEE 802.2/IEEE 802.3, or as it is known today: Ethernet.

Other LAN technologies, like token ring, were also developed and standardised after Ethernet, but were proprietary technologies and meant that two different LAN technologies could not interwork over a wide area network (WAN), i.e. company A’s token ring network could not communicate with company B’s Ethernet network.

It was only in the late 1970s that the non-proprietary transport control protocol/internet protocol (TCP/IP) was invented by Vinton Cerf and for the first time could dissimilar LAN and WAN protocols interwork. TCP/IP was developed and used in the Advanced Research Projects Agency Network (ARPANET) and made it possible for the different topologies of the ARPANET to interwork. ARPANET was later split into two parts: the first was intended for military use and was called MILNET, and the second was the Internet as it is known today (Dulaney et al., 1995, p.4).

The International Standards Organization (ISO) attempted to standardise an internetworking protocol to replace TCP/IP, but failed. Their work, however, was documented as the Open System Interconnection (OSI) reference model and is still used today throughout the communications industry. The model is described by seven layers: Application, Presentation, Session, Transport, Network, Data link and Physical (Blackwell, 1995, p.258).

2.3 INTERNET

Wilder defines Internetworking as *“the connection of multiple networks.”* The Internet is the connection of multiple diverse networks with different hardware technologies. The protocol use to construct the original Internet backbone is called TCP/IP. A non-profit company called Internet Society (ISOC) standardized all the protocols used by the Internet, which are collectively described as the TCP/IP protocol suite.

Many government agencies, colleges and companies are connected worldwide by the Internet. The most used upper layer programs of the Internet are the following:

- World Wide Web (WWW) browsers: To search data stored on the Internet, e.g. Internet Explorer & Netscape.
- Electronic Mail (email): To exchange memos and letters with individuals or companies, e.g. Outlook Express.
- File transfer: To exchange large files, e.g. CuteFTP, WSFTP.
- Remote login: To allow a user to log into an application located at a different computer.
- Remote procedure call: To execute remote procedures.

Many companies use the TCP/IP protocol suite to connect the different networks of their own company. This implementation of the TCP/IP protocol suite in a private network is referred to as *internets*, or more specific, as *intranets*. Their specifications, however, are the same as for the Internet.

2.4 TCP/IP OVERVIEW

The generic term "TCP/IP" means anything and everything related to the specific protocols of transmission control protocol (TCP) and Internet protocol (IP). It can include other protocols, applications, and even the network medium. A sample of these protocols are: UDP, ARP, and ICMP, while a sample of these applications are: TELNET, FTP, and rcp. The more accurate term for TCP/IP would be "internet technology". Any network that uses internet technology is called an "internet" (Bentham, 2000, p.73).

2.4.1 BASIC STRUCTURE

Figure 2.1 indicates the conceptual layering of the TCP/IP protocol suite compared to the OSI reference model. These layers are depicted in the logical structure (Figure 2.2) of the layered protocols inside a computer on an internet. Any computer that can communicate using internet technology has such a logical structure. It is this logical structure that determines the behaviour of the computer on the internet.

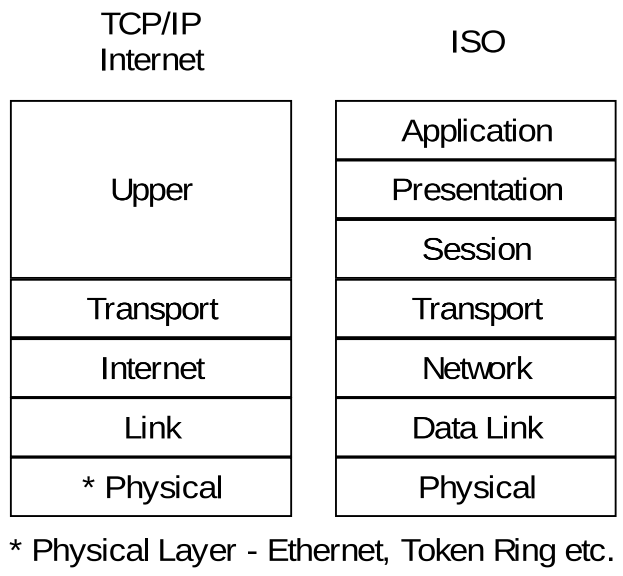


Figure 2.1 Conceptual layering of the Internet protocols (Adapted from Wilder, 1999, p.7).

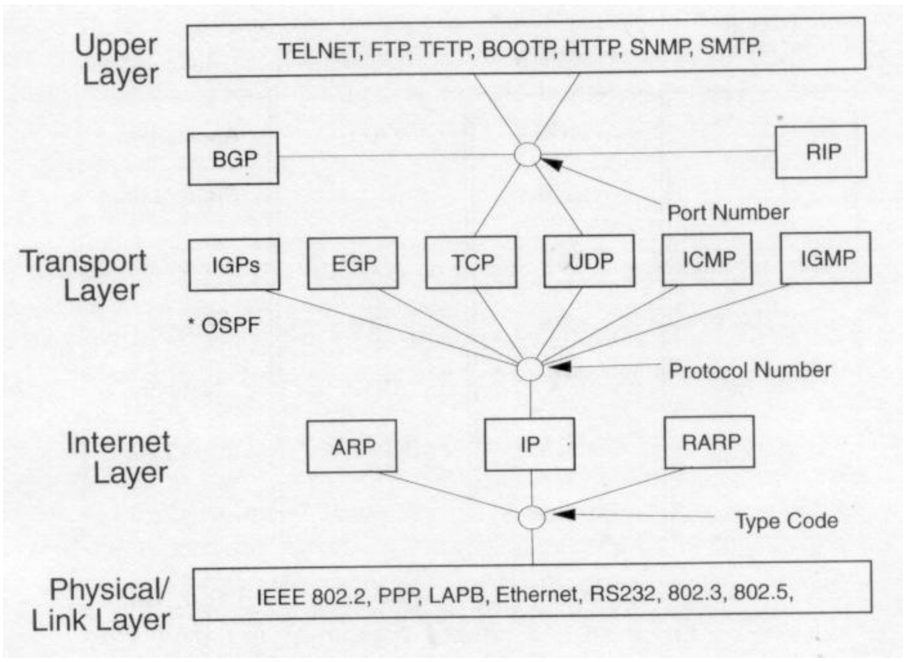


Figure 2.2 Internet architecture and protocols (Wilder, 1999, p.8).

The Internet provides three sets of services (Socolofsky et al., 2001, p.1). At the lowest level is the IP, which is a connectionless delivery service. The next level is the transport layer service and includes services like TCP, UDP, ICMP, etc. which uses the IP service. The highest level is the upper layer service and usually consists of the FTP (file browsing), HTTP (information browsing), etc. services, which uses the services of the transport layer. Theoretically the layered structure of the services permits research and development on one layer without affecting the others.

The physical/link layer envelops the IP layer header and data. If the physical layer is an Ethernet LAN, the IP layer places its message (datagram) in the Ethernet frame data field. The transport layer places its message (segment) in the IP data field. The application layer places its data in the transport layer data field. To indicate what combination of protocols in the TCP/IP suite is needed to transfer data, certain fields in each layer will indicate which service is needed in the next layer.

The upper layer protocols are divided into two groups – those that provide a utility function to the Internet and those that provide a service directly to the user (Wilder, 1999, p.8). Examples of those that provide a direct user service are the following:

- Hypertext transfer protocol (HTTP): Allows the user to receive information from the World Wide Web (WWW).
- Simple message transfer protocol (SMTP): Provides the user with email capabilities.
- File transfer protocol (FTP): Provides a service of reliable file transfers.
- TELNET: Provides remote logon capability.

Those that provide a utility function are:

- Simple network management protocol (SNMP): Provides network management information.
- Domain name service (DNS): To allow the using of names instead of Internet addresses.
- Address resolution protocol (ARP): Provides a link layer address given an IP address.
- Reverse address resolution protocol (RARP): Provides an IP address given a link layer address.

2.4.2 TERMINOLOGY

The name of a unit of data that flows through an internet is dependent upon where it exists in the protocol stack. In summary: if it is on an Ethernet it is called an Ethernet frame; if it is between the Ethernet driver and the IP module it is called a IP packet; if it is between the IP module and the UDP module it is called a UDP datagram; if it is between the IP module and the TCP module it is called a TCP segment; and if it is in a network application it is called a application message. A driver is defined as software that communicates directly with the network interface hardware and a module as software that communicates with a driver, with network applications, or with another module (Socolofsky et al., 2001, p.3).

2.4.3 FLOW OF DATA

For an application that uses TCP, data passes between the application and the TCP module. For applications that use UDP, data passes between the application and the UDP module. FTP is a typical application that uses TCP. Its protocol stack in this example is

FTP/TCP/IP/ENET. SNMP (Simple Network Management Protocol) is an application that uses UDP. Its protocol stack in this example is SNMP/UDP/IP/ENET.

The TCP module, UDP module, and the Ethernet driver are n-to-1 multiplexers. As multiplexers they switch many inputs to one output. They are also 1-to-n de-multiplexers. As de-multiplexers they switch one input to many outputs according to the type field in the protocol header. This is illustrated in Figure 2.3.

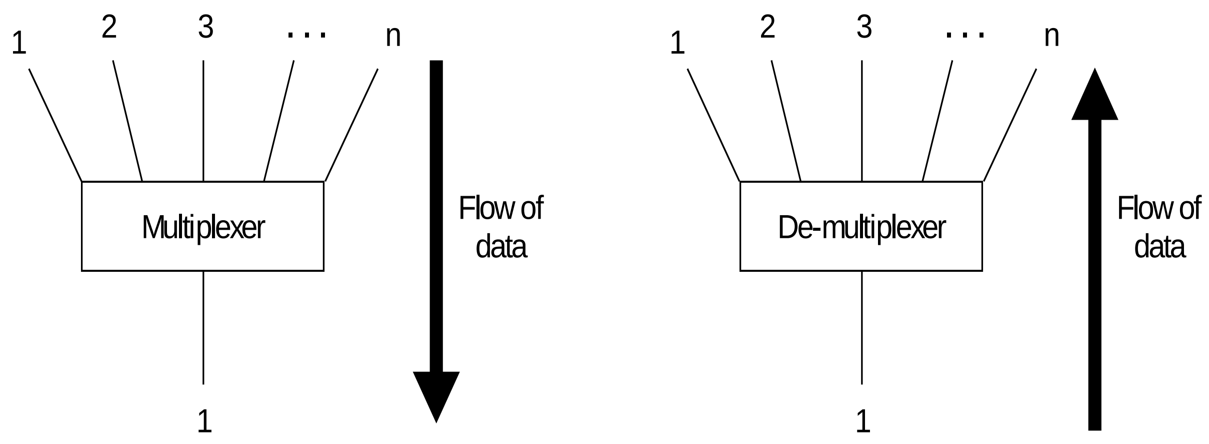


Figure 2.3 n-to-1 Multiplexer and 1-to-n de-multiplexer.

If an Ethernet frame comes up into the Ethernet driver off the network, the packet can be passed upwards to either the ARP module or to the IP module. The value of the type field in the Ethernet frame determines whether the Ethernet frame is passed to the ARP or the IP module.

If an IP packet comes up into IP, the unit of data is passed upwards to either TCP or UDP, as determined by the value of the protocol field in the IP header. If the UDP datagram comes up into UDP, the application message is passed upwards to the network application based on the value of the port field in the UDP header. If the TCP message comes up into TCP, the application message is passed upwards to the network application based on the value of the port field in the TCP header.

For downwards multiplexing there is only the one downward path; each protocol module adds its header information so the packet can be de-multiplexed at the destination computer. Data passing out from the applications through either TCP or UDP converges on the IP module and is sent downwards through the lower network interface driver.

Although internet technology supports many different network media, Ethernet is used for all examples because it is the most common physical network used under IP and will effectively also be the network that will be used for the project. The computer in Figure 2.2 has a single Ethernet connection. The 6-byte Ethernet address is unique for each interface on an Ethernet and is located at the lower interface of the Ethernet driver.

The computer also has a 4-byte IP address. This address is located at the lower interface to the IP module. The IP address must be unique for an internet. A running computer always knows its own IP address and Ethernet address.

2.4.4 IP CREATES A SINGLE LOGICAL NETWORK

The IP module is central to the success of internet technology (Arnett et al., 1995, p.20). Each module or driver adds its header to the message as the message passes down through the protocol stack. Each module or driver strips the corresponding header from the message as the message climbs the protocol stack up towards the application. The IP header contains the IP address, which builds a single logical network from multiple physical networks. This interconnection of physical networks is the source of the name: internet. A set of interconnected physical networks that limit the range of an IP packet is called an "internet" (Socolofsky et al., 2001, p.8).

2.4.5 PHYSICAL NETWORK INDEPENDENCE

IP hides the underlying network hardware from the network applications. If a new physical network has to be developed, it can be put into service by implementing a new driver that connects to the internet underneath IP. Thus, the network applications remain intact and are not vulnerable to changes in hardware technology.

2.5 ETHERNET

Ethernet uses Carrier Sense and Multiple Access with Collision Detection (CSMA/CD). CSMA/CD means that all devices communicate on a single medium, that only one can transmit at a time, and that they can all receive simultaneously (Socolofsky et al., 2001,

p.9). If 2 devices try to transmit at the same instant, the transmit collision is detected, and both devices wait a random (but short) period before trying to transmit again.
An Ethernet frame contains the destination address, source address, type field, data, and cyclic redundancy check (CRC) as indicated in Figure 2.4.

Destination address	Source address	Type	Data	CRC
6 bytes	6 bytes	2 bytes	46-1500 bytes	4 bytes

Figure 2.4 Ethernet frame (Adapted from Arnett et al., 1995, p.53).

2.5.1 DESTINATION AND SOURCE ADDRESSES

These 6-byte values identify the recipients and sender of the frame and are generally known as media access control (MAC) addresses (Arnett et al., 1995, p.56). Each network adaptor (controller) has its 6-byte address burned into a memory device at manufacture, but it is the responsibility of the networking software to copy this value into the appropriate field of the network packet. Every controller then listens for Ethernet frames with their destination address. All devices also listen for Ethernet frames with a wild-card destination address of "FF-FF-FF-FF-FF-FF" (in hexadecimal), called a "broadcast" address.

2.5.2 TYPE/LENGTH FIELD

This field is used differently for different Ethernet standards. The one standard uses it to indicate the total number of bytes in the data field. Other standards use it as a protocol type field, indicating which protocol is being used in the data field.

2.5.3 DATA

This area contains user data in any format with the only restriction that the minimum size must be 46 bytes and the maximum 1500 bytes (Bentham, 2000, p.11). The minimum is necessary to ensure that the overall frame is at least 64 bytes. If the frame has less than 64 bytes, there might be a danger that frame collisions would not be detected. The protocols of the Internet layer (ARP, IP, RARP) are transmitted in this data field of the Ethernet frame.

2.5.4 CYCLIC REDUNDANCY CHECK

The network controller checks this value to determine if the frame is corrupted and then discards this frame. It is automatically appended by the Ethernet controller on transmit and checked on receive.

2.6 ARP

Address Resolution Protocol (ARP) is used to translate IP addresses to Ethernet addresses. The translation is done only for outgoing IP packets, because this is when the IP header and the Ethernet header are created.

The translation is performed with a table look-up. The table, called the ARP table, is stored in memory and contains a row for each computer (Socolofsky et al., 2001, p.11). There is a column for IP address and a column for Ethernet address. When translating an IP address to an Ethernet address, the table is searched for a matching IP address. Table 2.1 indicates a simplified ARP table.

IP address	**Ethernet address**
223.1.2.1	08-00-39-00-2F-C3
223.1.2.3	08-00-5A-21-A7-22
223.1.2.4	08-00-10-99-AC-54

Table 2.1 Example ARP table.

The human convention when writing out the 4-byte IP address is each byte in decimal and separating bytes with a period. When writing out the 6-byte Ethernet address, the conventions are each byte in hexadecimal and separating bytes with either a minus sign or a colon.

The ARP table is necessary because the IP address and Ethernet address are selected independently; an algorithm cannot be used to translate IP address to Ethernet address. The IP address is selected by the network manager based on the location of the computer on the internet. When the computer is moved to a different part of an internet, its IP address must be changed. The Ethernet address is selected by the manufacturer based on the Ethernet address space licensed by the manufacturer. When the Ethernet hardware interface board changes, the Ethernet address changes.

2.7 INTERNET PROTOCOL

The IP module is central to internet technology and the essence of IP is its route table (Arnett et al., 1995, p.61). IP uses this in-memory table to make all decisions about routing an IP packet. The content of the route table is defined by the network administrator and any mistakes in this table will lead to blockage of communication.

The route table is best understood by first having an overview of routing, then learning about IP network addresses, and then looking at the details.

2.7.1 DIRECT ROUTING

Figure 2.5 represents a tiny internet with 3 computers: A, B, and C. Each computer has the same TCP/IP protocol stack as in Figure 2.2. Each computer's Ethernet interface has its own Ethernet address. Each computer has an IP address assigned to the IP interface by the network manager, who also has assigned an IP network number to the Ethernet.

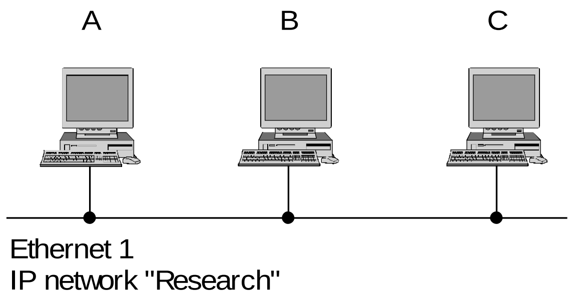


Figure 2.5 One IP network.

When A sends an IP packet to B, the IP header contains A's IP address as the source IP address, and the Ethernet header contains A's Ethernet address as the source Ethernet address. Also, the IP header contains B's IP address as the destination IP address and the Ethernet header contains B's Ethernet address as the destination Ethernet address. This is demonstrated in Table 2.2.

Address	Source	Destination
IP header	A	B
Ethernet header	A	B

Table 2.2 Addresses in an Ethernet frame for an IP packet from A to B.

When B's IP module receives the IP packet from A, it checks the destination IP address against its own, looking for a match, then it passes the datagram to the upper-level protocol.

This communication between A and B is termed direct routing (Socolofsky et al., 2001, p.16).

2.7.2 INDIRECT ROUTING

Figure 2.6 is a more realistic view of an internet. It is composed of 3 Ethernets and 3 IP networks connected by an IP-router called computer D. Each IP network has 4 computers; each computer has its own IP address and Ethernet address.

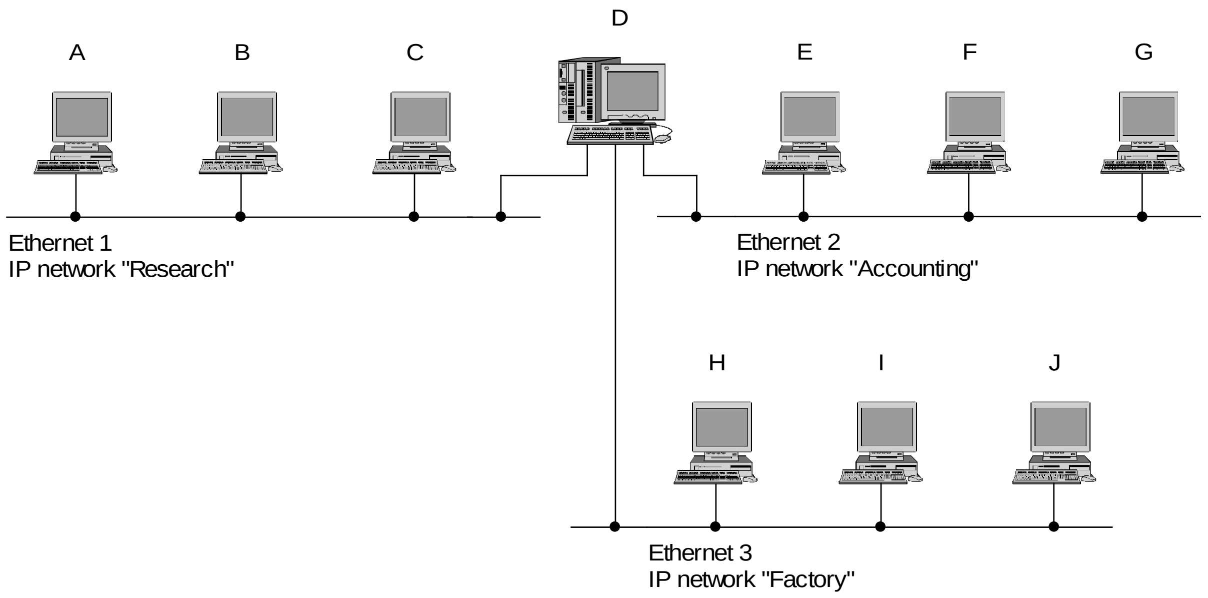


Figure 2.6 Three IP networks; one internet (Adapted from Socolofsky et al., 2001, p.17).

Except for computer D, each computer has a TCP/IP protocol stack like the one shown in Figure 2.2. Computer D is the IP-router. It is connected to all 3 networks and therefore has 3 IP addresses and 3 Ethernet addresses. Computer D has a TCP/IP protocol stack similar to that in Figure 2.2, except that it has 3 ARP modules and 3 Ethernet drivers. Note that computer D has only one IP module.

The network manager has assigned a unique number, called an IP network number, to each of the Ethernets. The IP network numbers are not shown in this diagram, just the network names.

When computer A sends an IP packet to computer B, the process is identical to direct routing. Any communication between computers located on a single IP network can be described using direct routing.

When computer D and A communicate, computer D and E communicate, and computer D and H communicate, it is direct communication. This is because each of these pairs of computers is on the same IP network.

However, when computer A communicates with a computer on the far side of the IP-router, communication is no longer direct. A must use D to forward the IP packet to the next IP network. This communication is called "indirect".

This routing of IP packets is done by IP modules and happens transparently to TCP, UDP, and the network applications.

If A sends an IP packet to E, the source IP address and the source Ethernet address are A's. The destination IP address is E's, but because A's IP module sends the IP packet to D for forwarding, the destination Ethernet address is D's, as indicated in Table 2.3.

Address	**Source**	**Destination**
IP header	A	E
Ethernet header	A	D

Table 2.3 Addresses in an Ethernet frame for an IP packet from A to E (before D).

D's IP module receives the IP packet and upon examining the destination IP address, detects that it is not its IP address, and sends the IP packet directly to E. This is indicated in Table 2.4.

Address	Source	Destination
IP header	A	E
Ethernet header	D	E

Table 2.4 Addresses in an Ethernet frame for an IP packet from A to E (after D).

2.7.3 IP ADDRESS

To identify an individual computer on the Internet, it must have a unique address. The current version of the Internet Protocol (IPv4) uses a 4-byte number, expressed in dotted decimal notation (e.g. 123.45.67.9). This address consists of 3 parts (Wilder, 1999, p.152):

- A network address, which uniquely identifies an organization.
- A subnet address, which identifies a subnet within the organization.
- A system address, which identifies a single node on that subnet.

So the address 123.45.67.8 has a class A network address of 123. However, if this node wants to contact another with the address 123.45.78.9, the knowledge that it is in the same organization is of little use. What the node really needs to know is whether the destination is on the same subnet. To do this, each node is equipped with a “subnet mask”; a logical AND with this value will eliminate the system address so that the rest of the address fields can be compared. If the node 123.45.67.8 has a subnet mask of 255.255.255.0, then it would detect that 123.45.78.9 was on a different subnet (123.45.67.8 AND 255.255.255.0 = 123.45.67.0 which is not equal to 123.45.78.0), whereas a mask value of 255.255.0.0 would suggest it is on the same subnet (123.45.67.8 AND 255.255.0.0 = 123.45.0.0 which is equal to 123.45.0.0).

2.7.4 NAMES

People refer to computers by names, not numbers. A computer called alpha might have the IP address of 223.1.2.1. For small networks, this name-to-address translation data is often kept on each computer in the "hosts" file. For larger networks, this translation data file is stored on a server and accessed across the network when needed. A few lines from that file might look like this:

```
223.1.2.1    alpha
223.1.2.2    beta
223.1.2.3    gamma
223.1.2.4    delta
223.1.3.2    epsilon
223.1.4.2    iota
```

2.8 USER DATAGRAM PROTOCOL

UDP is one of the two main protocols to reside on top of IP. It offers service to the user's network applications. Example network applications that use UDP are: Network File System (NFS) and SNMP. The service is little more than an interface to IP.

UDP is a connectionless datagram delivery service that does not guarantee delivery. UDP does not maintain an end-to-end connection with the remote UDP module. It merely pushes the datagram out on the net and accepts incoming datagrams off the net.

UDP adds two values to what is provided by IP (Socolofsky et al., 2001, p.27). One is the multiplexing of information between applications based on port number. The other is a checksum to check the integrity of the data.

2.9 TRANSMISSION CONTROL PROTOCOL

TCP provides a different service than UDP. TCP offers a connection-oriented byte stream, instead of a connectionless datagram delivery service. TCP guarantees delivery, whereas UDP does not.

TCP is used by network applications that require guaranteed delivery and cannot be bothered with doing time-outs and retransmissions. The two most typical network applications that use TCP are FTP and TELNET. TCP's greater capability is not without cost: it requires more central processing unit (CPU) and network bandwidth (Arnett et al., 1995, p.69). The internals of the TCP module are much more complicated than those in a UDP module.

Similar to UDP, network applications connect to TCP ports. Well-defined port numbers are dedicated to specific applications. For instance, the TELNET server uses port number 23. The TELNET client can find the server simply by connecting to port 23 of TCP on the specified computer.

When the application first starts using TCP, the TCP module on the client's computer and the TCP module on the server's computer start communicating with each other. These two end-point TCP modules contain state information that defines a virtual circuit. This virtual circuit consumes resources in both TCP end-points. The virtual circuit is full duplex. Data can go in both directions simultaneously. The application writes data to the TCP port, the data traverses the network and is read by the application at the far end.

TCP packetizes the byte stream at will. It does not retain the boundaries between writes. For example, if an application does 5 writes to the TCP port, the application at the far end might do 10 reads to get all the data or it might get all the data with a single read. There is no correlation between the number and size of writes at one end to the number and size of reads at the other end.

TCP is a sliding window protocol with time-out and retransmits. Outgoing data must be acknowledged by the far-end TCP. Acknowledgements can be piggybacked on data. Both receiving ends can flow control the far end, thus preventing a buffer overrun.

As with all sliding window protocols, the protocol has a window size. The window size determines the amount of data that can be transmitted before an acknowledgement is required. For TCP, this amount is not a number of TCP segments but a number of bytes.

2.10 TELNET

TELNET provides a remote login capability on TCP. The operation and appearance is similar to keyboard dialing through a telephone switch (Arnett et al., 1195, p.74). On the command line the user types "telnet delta" and receives a login prompt from the computer called "delta".

TELNET works well. It is an old application and has widespread interoperability. Implementations of TELNET usually work between different operating systems.

2.11 FILE TRANSFER PROTOCOL

File Transfer Protocol (FTP), as old as TELNET, also uses TCP and has widespread interoperability (rfc). The operation and appearance is as if a user TELNETed to the remote computer, but instead of typing usual commands, the user has to make do with a short list of commands for directory listings and the like. FTP commands allow the copying of files between computers.

2.12 HYPERTEXT TRANSFER PROTOCOL

Hypertext transfer protocol (HTTP) is used to transfer document files between computers over the Internet. The document files contain tagged words that are used as pointers to other information, or lists of other information. The Hypertext Markup Language (HTML) is used to construct the payload message transported by HTTP.

Basically it is a upper layer application that uses the well known TCP port 80 to create a connection between a client and server (Wilder, 1999, p.345). A client simply sends a request message to a HTTP server, which sends a response message to the client and closes the connection.

2.13 CONCLUSION

Networking protocols and their general operation was explained without any in-depth detail. The purpose for this chapter was to provide a background into networking to be able to understand the rest of the thesis.

Chapter 3 will build on the knowledge obtained about certain protocols in this chapter to establish an Ethernet connection via an 8-bit microcontroller.

CHAPTER 3
EMBEDDED ETHERNET

3.1 OVERVIEW

In his book “TCP/IP Lean: Web Servers for Embedded Systems”, Jeremy Bentham describes techniques to implement a small web server on a PIC microcontroller. The only limitation is that it can only be used on a LAN using a serial (SLIP) link. He went further by creating methods to implement this same system to allow Ethernet communication. These hardware methods and how they fit into the entire project will be discussed in this chapter. The references for this whole chapter are: “TCP/IP Lean: Web Servers for Embedded Systems (J. Bentham)” and the “PICDEM.net User’s Guide (Microchip)”.

3.2 HARDWARE

This section will explain the hardware requirements to implement Ethernet on an embedded system.

3.2.1 ETHERNET INTERFACE

As explained in Chapter 2, Ethernet is designed to allow a large number of terminals to be connected together, which basically means that one computer could send data to any of the other computers on that network. Currently the star topology is implemented by Ethernet where each computer is connected to a repeater and several repeaters again are connected together to form a LAN (Figure 3.1).

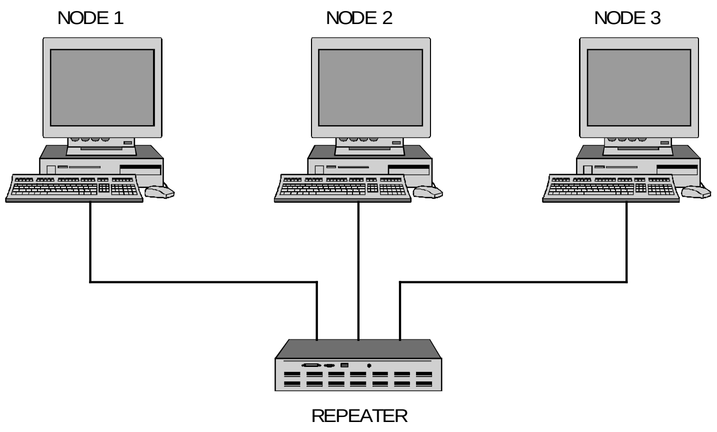


Figure 3.1 Star network topology.

Hardware on each computer arbitrates with other computers on the network so they can transmit data when necessary. Each computer (node) has its own 6-byte Ethernet address (MAC address) to enable it to filter out messages not intended for that node and hence to reduce the strain implied on the resources of that node. All the messages on the network have a destination MAC address and a source MAC address where one node wants to communicate to another node. This is known as unicast. Sometimes it might be necessary to send a message to all nodes on the network which is known then as broadcast.

The need to filter certain MAC addresses and the requirement to handle the 10Mbps data rate, dictates the use of Ethernet specific hardware. Currently there are no microcontrollers with built-in Ethernet interfaces and therefore external network interface hardware has to be used.

3.2.2 ETHERNET HARDWARE

The old style Industry Standard Architecture (ISA) computer networks cards need to buffer all incoming and outgoing messages in a packet buffer to avoid overloading the computer. Access to these cards via the ISA bus is more or less 1 MB per second, whereas the data rate at which messages enter and leave the network card is 10 MB per second.

Modern network interface cards need less buffering, since they can interface directly to the PC main memory over a fast 32-bit bus. This, however, is unsuitable for microcontrollers because microcontroller speeds are not fast enough for the restrictions of these cards and therefore it was decided to use an old style ISA buffered interface.

Figure 3.2 indicates a block diagram of the ISA network interface controller (NIC) that will be used in the project.

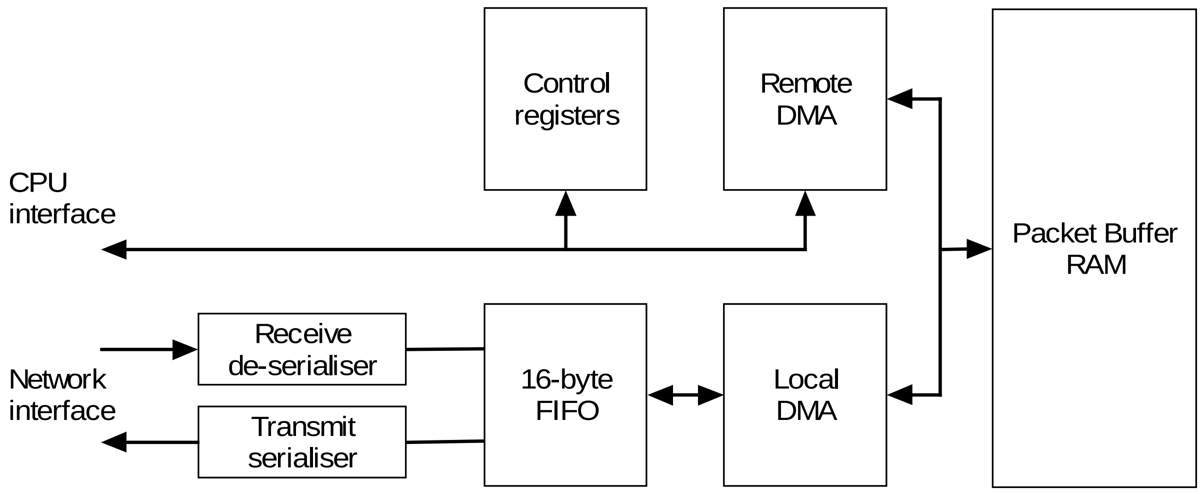


Figure 3.2 Ethernet hardware block diagram (Adapted from PICDEM.net User's guide, p.29).

- The CPU can read and write the control registers. These registers are organized in banks (register pages) so that a large number of registers can be accessed using only 4 address bits.
- The CPU cannot access the packet buffer random access memory (RAM) directly, but instead has to make use of the remote direct memory access (DMA) controller. The control registers are used to set up the desired packet buffer address and byte count, whereafter the CPU repeatedly reads or writes a data latch, to transfer a block of data to and from the packet buffer. The latch is not one of the control registers and therefore an extra line is needed, making the number of address lines a total of five (A0-4).
- Once the NIC is set up, it can now transfer data between the packet buffer and the network interface using a small first-in/first-out (FIFO) buffer and a second DMA channel. The NIC is intelligent enough so that it can receive several messages (Ethernet frames) without intervention from the host CPU.

The Realtek RTL8019AS was chosen as the NIC to be used in the project. It is a derivative of the well-known DP8390 NIC which was one of the first network controllers that was developed. Figure 3.3 shows the general connection between a Ethernet controller and a microcontroller.

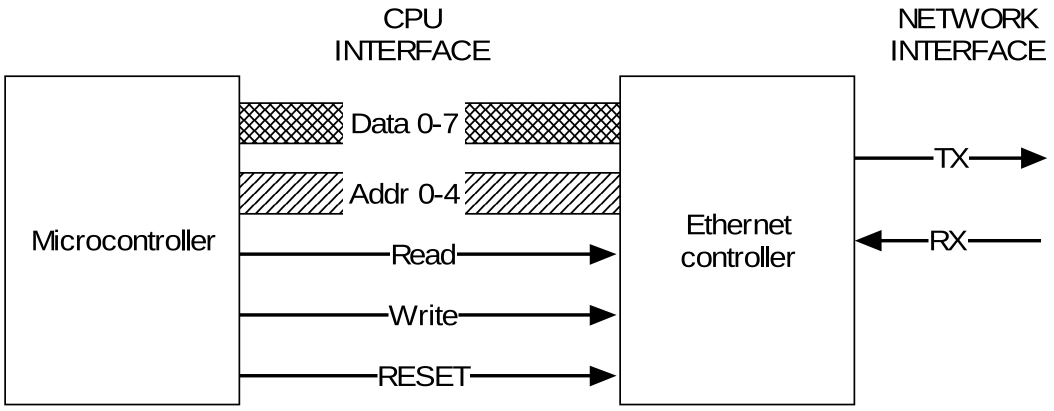


Figure 3.3 Ethernet controller interfaces (Adapted from PICDEM.net User's guide, p.30).

3.2.3 MICROCONTROLLER INTERFACE

In order to drive the RTL8019AS Ethernet controller, the microcontroller has to simulate the read and write cycles of a standard computer ISA bus. The steps for a single cycle read/write are as follows:

1. **Set the address.**
   The standard RTL8019AS controller has 20 address lines (A0-A19), but only 5 of these lines are used to access the registers that are needed in this application. The rest of the address lines are tied down to ground.

2. **Set the data lines.**
   In a read cycle these lines act as inputs to the microcontroller and in a write cycle they act as outputs from the microcontroller.

3. **Activate the Read or Write signals.**
   To activate a Read or Write signal, these lines have to be set LOW.

4. **If it is a Read cycle, fetch the data.**
   As soon as the Read signal goes LOW for a read cycle, the Ethernet controller drives the data bus and places the data at the specified address on the data bus.

5. **De-activate both Read and Write signals.**
   If it was a Read cycle, the Ethernet controller will stop driving the data bus. If it was a Write cycle the controller will latch the data that was placed on the data bus by the microcontroller.

6. **Unset the data lines.**
   The microcontroller output driver for the data bus should be disabled so that other devices can use the data bus, i.e. the microcontroller data port should be set as inputs.

Figure 3.4 indicates the signals involved in a read or write cycle.

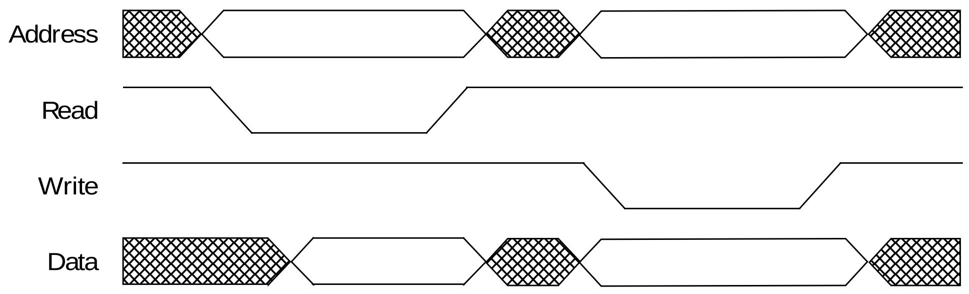


Figure 3.4 Ethernet controller access cycles (PICDEM.net User's guide, p.31).

The following I/O lines are needed on the microcontroller:

- One read output and 1 Write output.
- Five address outputs. These lines are only needed when the Ethernet controller is accessed and could be used for other functions otherwise.
- Eight bi-directional data lines, which are also only needed when the Ethernet controller is accessed.

Microchip's PIC16F877 was chosen as the microcontroller to interface to the RTL8019AS Ethernet controller for the following reasons:

- It has a total of 33 I/O pins in a 40-pin DIP package, making it very suitable to interface the RTL8019AS and then still have 18 I/O pins left to use for other parts of the project, as will be described in the following chapters.
- Microchip's "F" series microcontrollers contain FLASH based program memory which means that they can be reprogrammed without removing the device from the

board. This is a huge advantage, as this will reduce the developing time quite significantly.

- To make development even easier, the PIC16F877 consists of built-in debug capabilities.
- It contains built-in EEPROM data memory, which makes it convenient to save set-up data (such as the MAC address and IP address) internally and then not to worry about a power failure where such data might be lost.

Figure 3.5 indicates the actual connections of the RTL8019AS and PIC16F877 to establish the Ethernet interface for the project.

- D1 is used as the system LED and flashes if the system is in operation.
- The S1 pushbutton is used for resetting the Ethernet interface, while S2 is used for setting up the parameters (e.g. MAC and IP addresses) for the whole system.
- J3 acts as the in-circuit debugger (ICD) port for debugging and programming the PIC16F877 while still in the circuit.
- A power supply section is used to provide a regulated +5V for the whole system.
- A liquid crystal display (LCD) interface is included for use to display messages or information. Jumpers JP1 to JP8 determine to which source the LCD is connected. If the jumpers are in the position 1-3, then the LCD is connected to the I/O controller, and if it is in the
  1-2 position, it is connected to the Ethernet interface controller.
- A RS-232 interface is also provided to make development easier and could be used for debugging purposes. Jumpers JP4 and JP5 can disconnect the PIC16F877's RS-232 module from the RS-232 transceiver and is then connected in such a way to establish RS-232 communication between the Ethernet interface controller and the I/O controller.
- U4, the RTL8019AS, uses three LEDs to indicate if there are packets transmitted, received or if collisions occur. The physical connection to the Ethernet port (J4) is established using the FL1012 untwisted pair (UTP) transceiver.
- The STATUS0-3 and CMD0-1 nets on the PIC16F877 are used for command and status control between the Ethernet controller and the Programmer controller. These lines will be explained in detail in the following chapters.

Figure 3.5   The embedded Ethernet interface.

3.3 ETHERNET DRIVER

The following sections will describe the software drivers that are needed to access the RTL8019AS and hence to simulate the PC ISA bus. The drivers were written in the C language using the C-compiler from Custom Computer Services (www.ccsinfo.com).

3.3.1 NIC INITIALIZATION

The most fundamental part of the Ethernet device driver is to be able to read and write to the NIC. Figure 3.6 indicates how this is accomplished in the firmware.

```
/* Input a byte from a NIC register */
BYTE innic(int reg)
{
   BYTE b;

   DATA_FROM_NIC;
   NIC_ADDR = reg;
   NIC_IOR_ = 0;
   b = NIC_DATA;
   NIC_IOR_ = 1;
   return(b);
}

/* Output a byte to a NIC register */
void outnic(int reg, int b)
{
   NIC_ADDR = reg;
   NIC_DATA = b;
   DATA_TO_NIC;
   NIC_IOW_ = 0;
   delay_cycles(1);
   NIC_IOW_ = 1;
   DATA_FROM_NIC;
}
```

Figure 3.6 Read and write functions (PICDEM.net User's guide, p.34).

The **DATA_FROM** and **DATA_TO** macros are used to set the microcontroller data port direction registers (TRIS). To ensure the Read and Write signals remain LOW for the correct time, a single CPU cycle delay is added to each function.

Once the read and write functions are established, the NIC can be initialised using a series of **outnic** functions calls as indicated in Figure 3.7.

```
outnic(CMDR, 0x21);            /* Stop, DMA abort, page 0 */
delay_ms(2);                   /* ..wait to take effect */
outnic(DCR, DCRVAL);
outnic(RBCR0, 0);              /* Clear remote byte count */
outnic(RBCR1, 0);
outnic(RCR, 0x20);             /* Rx monitor mode */
outnic(TCR, 0x02);             /* Tx internal loopback */
```

Figure 3.7   NIC initialisation (PICDEM.net User's guide, p.34).

Note that Figure 3.7 doesn't show all the initialisation parameters needed. The only real important parameters are the following:

- **Ethernet MAC address**
  This 6-byte address must be supplied to the NIC so that it can filter out incoming packets not intended for this node.
- **Address filtering**
  The parameter enables or disables promiscuous mode. Enabling promiscuous mode means that the NIC will accept all packets it receives and will not filter them out, which is convenient for initial development and testing of the firmware.
- **RAM size**
  The RTL8019AS has a total of 16Kbytes of packet buffer RAM. When the NIC is used in 8-bit mode, it is suggested that only 8Kbytes should be used. Doing this will mean that there are 1.5Kbytes for the transmit buffer and 6.5K for the receive buffer.

3.3.2 ACCESSING THE PACKET BUFFER

The operation of a normal network adaptor is such that the host CPU will read the entire packet from the Ethernet controller into its own RAM and then process it from there. Since one packet is much larger than the RAM capacity of a microcontroller, it will be necessary to fetch and process the incoming packets in small chunks. The NIC's DMA controller is useful for this purpose, since it can accept any packet buffer address and byte count.
Figure 3.8 shows the methods to read and write data from the NIC's RAM area.

```
/* Set the 'remote DMA' address in the NIC's RAM to be accessed */
void setnic_addr(WORD addr)
{
   outnic(ISR, 0x40);                  /* Clear remote DMA interrupt flag */
   outnic(RSAR0, addr&0xff);           /* Data addr */
   outnic(RSAR1, addr>>8);
}

/* Get data from NIC's RAM into the given buffer */
void getnic_data(BYTE *data, int len)
{
   BYTE b;

   outnic(ISR, 0x40);                  /* Clear remote DMA interrupt flag */
   outnic(RBCR0, len);                 /* Byte count */
   outnic(RBCR1, 0);
   outnic(CMDR, 0x0a);                 /* Start, DMA remote read */
   while (len--)                       /* Get bytes */
   {
      b = innic(DATAPORT);
      *data++ = b;
   }
}

/* Put the given data into the NIC's RAM */
void putnic_data(BYTE *data, int len)
{
   len += len & 1;                     /* Round length up to an even value */
   outnic(ISR, 0x40);                  /* Clear remote DMA interrupt flag */
   outnic(RBCR0, len);                 /* Byte count */
   outnic(RBCR1, 0);
   outnic(CMDR, 0x12);                 /* Start, DMA remote write */
   while (len--)                       /* O/P bytes */
      outnic(DATAPORT, *data++);
   len = 255;                          /* Done: must ensure DMA complete */
   while (len && (innic(ISR)&0x40)==0)
      len--;
}
```

Figure 3.8 Reading and writing data from RTL8019AS RAM (PICDEM.net User's guide, p.36).

3.3.3 PACKET RECEPTION

Packet analysis basically involves the following:

- Checking for over-runs of the packet buffer
- Detecting that one or more packets have been received
- Checking the error status of the packet
- Establishing the start address of the packet in the buffer
- Freeing up the buffer RAM used by the packet

Each Ethernet packet has its own header containing the 6-byte destination and source addresses (known as the MAC addresses). The structure to define these headers is shown in Figure 3.9.

```
#define MACLEN         6

typedef struct {                    // Ethernet frame header
   BYTE dest[MACLEN];               //     Dest & srce MAC addresses
   BYTE srce[MACLEN];
   WORD pcol;                       //     Protocol
} ETHERHEADER;
```

Figure 3.9 Ethernet header structure (PICDEM.net User's guide, p.37).

The total frame size, including the header and a 4-byte CRC, is between 64 and 1518 bytes, so the actual length of data is between 46 and 1500 bytes. If less than the minimum, the data is padded to fit.

The NIC also adds its own hardware-specific header on the front of the Ethernet packet, so that the length and error status are known (Figure 3.10).

```
typedef struct {                     // NIC hardware packet header
   BYTE stat;                        //     Error status
   BYTE next;                        //     Pointer to next block
   WORD len;                         //     Length of this frame incl. CRC
} NICHEADER;
```

Figure 3.10 NIC hardware-specific header (PICDEM.net User's guide, p.37).

As a result, the frame in the NIC buffer RAM has two headers in front of the data and to simplify this, the driver copies them into local RAM.

```
typedef struct {                                        // NIC and Ethernet headers combined
   NICHEADER nic;
   ETHERHEADER eth;
} NICETHERHEADER;

NICETHERHEADER nicin;                                   // Buffer for incoming NIC & Ether hdrs

/* Get packet into buffer, return length (excl CRC), or 0 if none available */
WORD get_ether()
{
   WORD len=0, curr;
   . . .
   getnic_data((BYTE *)&nicin, sizeof(nicin));
```

```
    len = nicin.nic.len;                              /* Take length from stored header */
    . . .
    len -= MACLEN+MACLEN+2+CRCLEN;
    . . .
    return(len);                                      /* Return length excl. CRC */
}
```

Figure 3.11 Receiving an Ethernet frame (PICDEM.net User's guide, p.38).

3.3.4 PACKET ANALYSIS

Due to the shortage of local on-chip RAM in the microcontroller, the incoming packet must be analysed as it is in the NIC buffer RAM. The packet data is then fetched a byte at a time while the checksum is computed for IP verification. Figure 3.12 shows some of the packet analysis functions.

```
/* Get a byte from network buffer; if end, set flag */
BYTE getch_net(void)
{
    BYTE b=0;

    atend = rxout>=rxin;
    if (!atend)
    {
        b = getnic_byte();
        rxout++;
        check_byte(b);
    }
    return(b);
}

BYTE ungot_byte;
BOOL ungot;

/* Get an incoming byte value, return 0 if end of message */
BOOL get_byte(BYTE &b)
{
    if (ungot)
        b = ungot_byte;
    else
        b = getch_net();
    ungot = 0;
    return(!atend);
}

/* Unget (push back) an incoming byte value */
void unget_byte(BYTE &b)
{
    ungot_byte = b;
    ungot = 1;
}

/* Get an incoming word value, return 0 if end of message */
BOOL get_word(WORD &w)
{
    BYTE hi, lo;
```

```
   hi = getch_net();
   lo = getch_net();
   w = ((WORD)hi<<8) | (WORD)lo;
   return(!atend);
}

/* Match an incoming byte value, return 0 not matched, or end of message */
BOOL match_word(WORD w)
{
   WORD inw;

   return(get_word(inw) && inw==w);
}

/* Skip an incoming word value, return 0 if end of message */
BOOL skip_word(void)
{
   getch_net();
   getch_net();
   return(!atend);
}
```

Figure 3.12   Packet analysis functions (PICDEM.net User's guide, p.39).

The **Unget()** function is included so that one byte can be pushed back into the NIC buffer RAM. The **get_**, **match_**, and **skip_** functions are used for the actual packet analysis.

3.3.5 PACKET TRANSMISSION

The packet transmission driver has to do the following:

- Write the Ethernet header (destination & source addresses, and type of protocol) into the NIC packet buffer.
- Write the packet data into the buffer.
- Set the length of the packet in the NIC registers.
- Start the NIC state machine.

The NIC will automatically retry the transmission if it fails due to a collision, but the transmission could still fail if the network is heavily loaded. The packet transmission drivers will not take action in this event, since it is the upper layers (TCP) that initiate a retry. Figure 3.13 shows two of the packet transmission functions.

```
#define TXBUFFLEN 64
BYTE txbuff[TXBUFFLEN];                    // Tx buffer
int txin, txout;

/* Put a byte into the network buffer */
void putch_net(BYTE b)
{
    if (txin < TXBUFFLEN)
        txbuff[txin++] = b;
    check_byte(b);
}

/* Send Ethernet packet given payload len */
void put_ether(void *data, WORD dlen)
{
    outnic(ISR, 0x0a);                /* Clear interrupt flags */
    setnic_addr(TXSTART<<8);
    putnic_data(nicin.eth.srce, MACLEN);
    putnic_data(myeth, MACLEN);
    swapw(nicin.eth.pcol);
    putnic_data(&nicin.eth.pcol, 2);
    putnic_data(data, dlen);
}
```

Figure 3.13 Packet transmission functions (PICDEM.net User's guide, p.40).

3.4 PROTOCOL DRIVERS

The drivers for the ARP, IP, ICMP, TCP and HTTP protocols are not included in this chapter as they basically remain the same as described in Bentham's book "TCP/IP Lean: Web Servers for Embedded Systems". Changes to these protocols will be indicated in the relevant chapters that follow.

3.5 CONCLUSION

This chapter explained the hardware implementation of the RTL8019AS Ethernet controller in an embedded system to establish Ethernet communication with a PIC microcontroller. Software drivers for this interface have also been described briefly.

The next chapter will deal with one of the upper layer protocols: HTTP. This will then be used to create an embedded Web server using the same PIC microcontroller that is used for interfacing with the RTL8019AS.

CHAPTER 4
EMBEDDED WEB SERVER

4.1 OVERVIEW

This chapter takes a look at HTTP and how it was implemented in the project. The HTML language will also be briefly discussed. The project makes use of its own read-only memory (ROM) file system, which is explained with an example. Only the relevant sections of the Web server driver are discussed.

4.2 HYPERTEXT TRANSFER PROTOCOL

HTTP is the primary protocol used to distribute information on the Web (Stanek, 1996, p.15). HTTP is a powerful and fast protocol that allows for easy exchange of files.

To achieve high speed and versatility, HTTP is defined as a connectionless and stateless protocol. This means that the client and server do not maintain a connection or state information related to the connection. When clients connect to the server, they make a request, get a response, then disconnect. Because a connection is not maintained, no system resources are used after the transaction is completed. Consequently, HTTP servers are only limited by active connections and can generally service hundreds of transactions with low system overhead (Stanek, 1996, p.16). The only drawback to connectionless protocols is that when the same client requests additional data, the connection must be re-established. To Web users this means a delay whenever additional data is requested.

HTTP is also a stateless protocol. Servers using stateless protocols maintain no information about completed transactions and processes. When a client breaks a connection with a server running a stateless protocol, there is no data that has to be cleaned up or logged. By not tracking state information, there is less overhead on the server and the server can generally handle transactions swiftly (Stanek, 1996, p.17).

The fact that HTTP is connectionless and stateless makes it possible to use a PIC microcontroller as a Web server, seeing that the controller will not be occupied constantly.

4.2.1 GET METHOD

To fetch a Web document, the browser (Internet Explorer 5.0, Netscape etc.) opens a TCP connection to server port 80, then uses the HTTP protocol to send a request. The request and response are one or more lines of text, each terminated by the newline characters. If the request is successful, the information (document text, graphical data) is then sent down the same connection, which is then close on completion. HTTP commands are called methods. The one used to fetch documents is the GET method.

4.2.2 REQUEST

A basic request consists of the following:

- The keyword GET
- The filename
- A protocol identifier
- The newline character

e.g.

```
GET /index.htm HTTP/1.0<CR><LF>
```

Generally a browser only receives data (Web pages) from the Web server, but it is also possible to send data back to the server. In this case the request stays exactly as above, but the ‘?’ character is added together with some tag specific characters.

e.g.

```
GET /index.htm?Q.x=10&Q.y=10 HTTP/1.0
```

This example demonstrates what the request will look like if the user clicks on an image, on the Web page, with the tag name ‘Q’. The x and y coordinates are passed to the server in this case.

Following the GET method is an optional header containing further browser-specific details, such as its configuration and the document format it can accept (Figure 4.1).

```
GET /index.htm HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.5 [en] (Win95; I)
Pragma: no-cache
Host: 10.1.1.11
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, /image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```

Figure 4.1 An example of a GET method with the optional header.

The optional header is simply ignored in this mini Web server application since this data is not necessary for a simple Web page transfer.

4.2.3 RESPONSE

The server replies with a response line containing the HTTP version, status code and description, such as

```
HTTP/1.0 200 OK
```

or an error code as shown below.

```
HTTP/1.0 404 Not Found
```

The server may then send optional header information in a similar format to the request headers, but once again, the header will be ignored in this case seeing that it is not necessary and will only take up more resources on the PIC microcontroller.

Immediately after the response line from the server, the server sends a blank line followed by the entity body. The entity body is simply the text of graphic data that was requested by the client. The request/response process using the HTTP protocol is demonstrated in Figure 4.2.

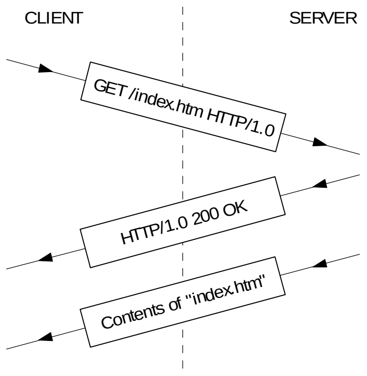


Figure 4.2 An example of a HTTP request/response process.

4.3 HYPERTEXT MARKUP LANGUAGE

HTML is the most commonly used language for designing Web pages (Stanek, 1996, p18). HTML's popularity stems in large part from its ease of use and friendliness. With HTML, the user can quickly and easily create Web documents and make them available to a wide audience. HTML also enables the user to control many of the layout aspects for text and images (e.g. size, position, etc.).

By default, all text in an HTML document is intended for display on the browser screen. HTML **tags** contain information that affects the text presentation and allows the insertion of extra information into the text, such as graphic images and navigational links. The tag consists of left and right angle brackets ('<' and '>') enclosing text that is generally insensitive to case. Figure 4.3 demonstrates a simple HTML document and the resulting Web page.

```html
<html>
<head>
<title>New Page 1</title>
</head>
<body bgcolor="#00FFFF">
<p align="center"><b><font face="Arial" size="7" color="#000000">Hi There!!!</font></b></p>
<p align="center"><img border="0" src="bd06784_.gif" width="264" height="243"></p>
</body>
</html>
```

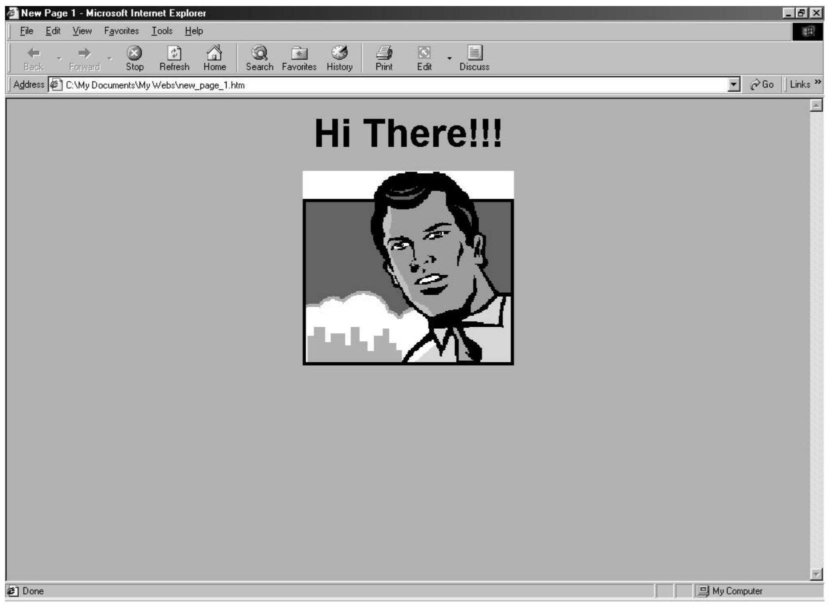

New Page 1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address C:\My Documents\My Webs\new_page_1.htm
Go Links
Hi There!!!
Done
My Computer

Figure 4.3 Example of a HTML document.

4.4 DYNAMIC WEB PAGES

A classic Web server provides Web pages without alteration and is essentially just a file server. Modern servers can alter Web pages on the fly or create the pages from scratch each time they are requested. The umbrella term for this facility is common gateway interface (CGI), which, because of the large amount of string manipulation involved, is usually implemented on a powerful system running a language, such as Perl, that is well suited for the job (Bentham, 2000, p.237).

Because of the PIC microcontroller's lack of RAM and difficulty with string manipulation, it will not be suitable for implementing CGI. Instead another method called embedded gateway interface (EGI) will be used. This method was developed by Jeremy Bentham, but was changed and simplified for the purposes of this project

4.4.1 DYNAMIC REQUEST USING EGI

The EGI will be invoked every time a client requests a document with an ".egi" extension. The server will then do the following steps:

- Look up the document name with the matching ".egi" extension and if not found, an error message will be displayed.
- Start transmitting the contents of the document byte by byte using the HTTP protocol.
- When an EGI variable substitution character ('@' or '#') is detected, the value of the particular variable name following that will be requested from the I/O controller. Transmission is temporarily halted.
- On receiving the value from the I/O controller, the EGI character and the variable name is removed from the document and the value of the variable is transmitted.
- Transmission then continues as normally, until the next EGI character is detected.

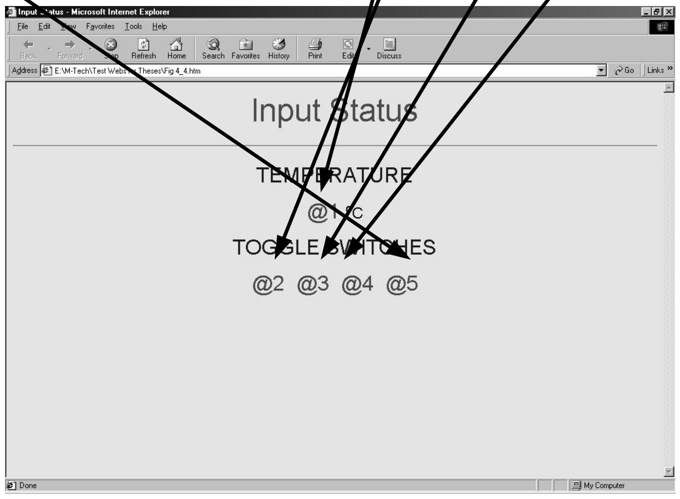
Figure 4.4 demonstrates a simple Web page utilising EGI to read the temperature and values of four toggle switches from the remote embedded system. The first line:

```
<html><meta http-equiv="refresh" content="3">
```

causes the Web page to be refreshed every 3 seconds to update the values for the temperature and the toggle switches.

```
<html><meta http-equiv="refresh" content="3">
<title>Input Status</title>
<body bgcolor="#CCFF99">
<p align="center"><font face="Arial" size="7" color="#FF0000">Input Status</font></p>
<hr>
<p align="center"><font face="Arial" color="#0000FF" size="6">TEMPERATURE</font></p>
<p align="center"><font face="Arial" size="6" color="#FF0000">@1</font><font face="Arial"
color="#0000FF" size="6">
</font><font face="Arial" size="5"><sup>o</sup>C</font></p>
<p align="center"><font face="Arial" color="#0000FF" size="6">TOGGLE SWITCHES</font></p>
<p align="center"><font face="Arial" size="6" color="#FF0000">@2&nbsp; @3&nbsp;@4&nbsp;
@5</font></p>
</body>
</html>
```

Viewed as a normal Web page:


Input Status - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address E:\M-Tech\Test Webs for Theses\Fig 4_4.htm
Go Links
Done
My Computer

Input Status

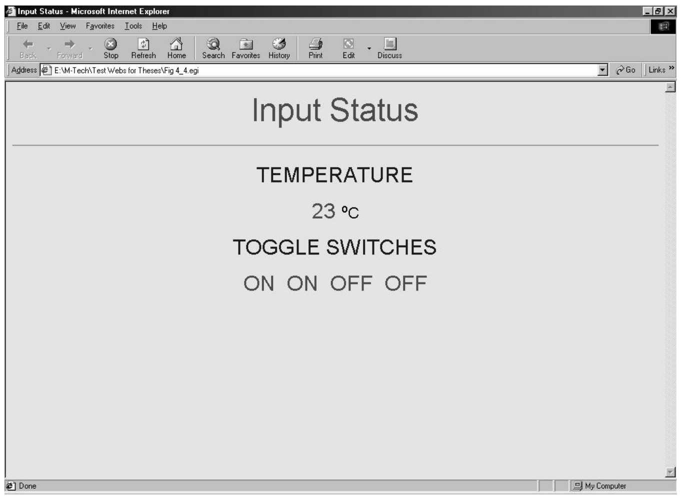
TEMPERATURE

@1 oC

TOGGLE SWITCHES

@2 @3 @4 @5

Viewed after being connected to the embedded system:


Input Status - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address E:\M-Tech\Test Webs for Theses\Fig 4_4.egi
Go Links
Done
My Computer

Input Status

TEMPERATURE

23 oC

TOGGLE SWITCHES

ON ON OFF OFF

Figure 4.4 An example to demonstrate dynamic Web page requests using EGI.

4.4.2 DYNAMIC OUTPUT

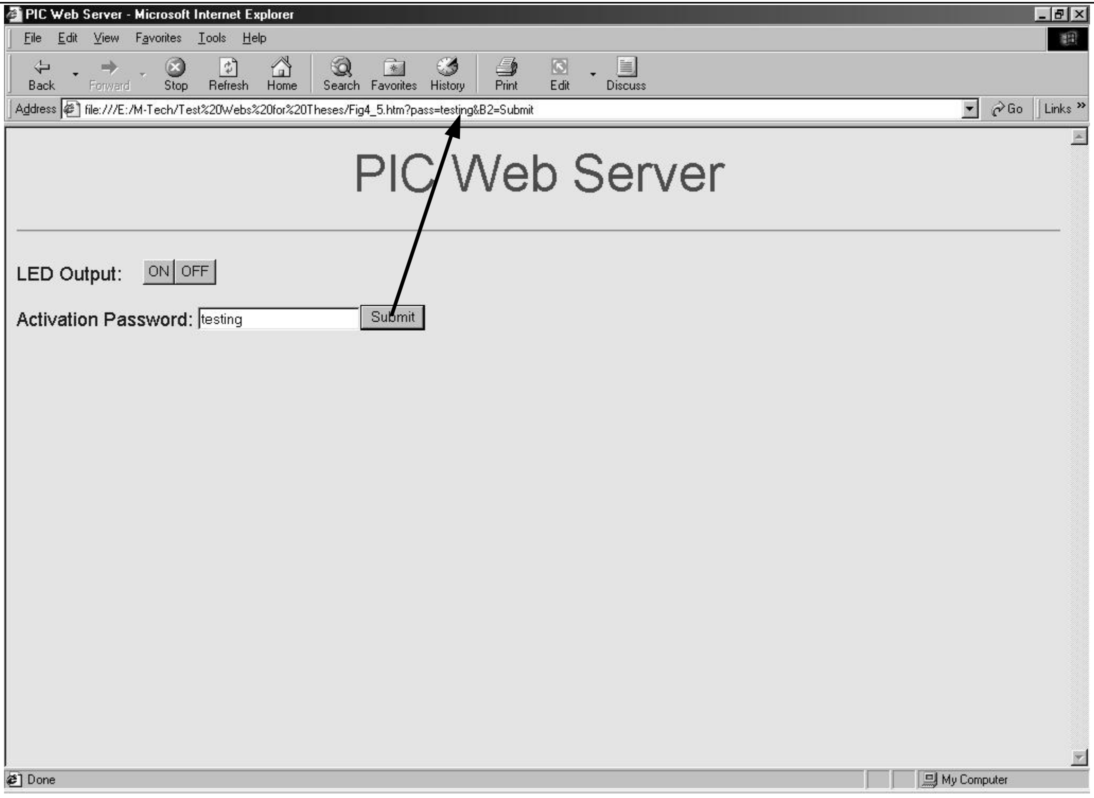
When a client needs to send data to a server, Web page form controls can be used. This will cause the Web page request to have ‘?’ character at the end of the line (as described in section 4.2.2). In this case, it is not necessary for the Web page to have an “.egi” extension and it could be just a normal “.htm” or “.html” document with form controls on it. As soon as the server recognizes the ‘?’ character in the request line, it will:

- evaluate the rest of the data following the ‘?’ character.
- send this data to the I/O controller so that it can make changes to the system accordingly.
- transmit the normal response along with the Web page contents.

Figure 4.5 gives an example of such a transaction and is continued on the next to show the results.

```
<html>
<head>
<title>PIC Web Server</title>
</head>
<body bgcolor="#CCFF99">
<p align="center"><font face="Arial" size="7" color="#FF0000">PIC Web Server</font>
</p>
<hr>
<form action="Fig4_5.htm">
  <p align="left"><font size="4" face="Arial" color="#0000FF">
  LED Output:&nbsp;&nbsp;&nbsp;&nbsp;</font><input type="submit" value="ON" name="B1"><input
type="submit" value="OFF" name="B1"></p>
</form>
<form action="Fig4_5.htm">
  <p align="left"><font face="Arial" color="#0000FF" size="4">Activation
  Password: </font> <input type="text" name="pass" size="20"><input type="submit" value="Submit"
name="B2"></p>
</form>
<p align="left">&nbsp;</p>
</body>
</html>
```

Figure 4.5 An example to demonstrate dynamic Web pages using form controls.


PIC Web Server - Microsoft Internet Explorer

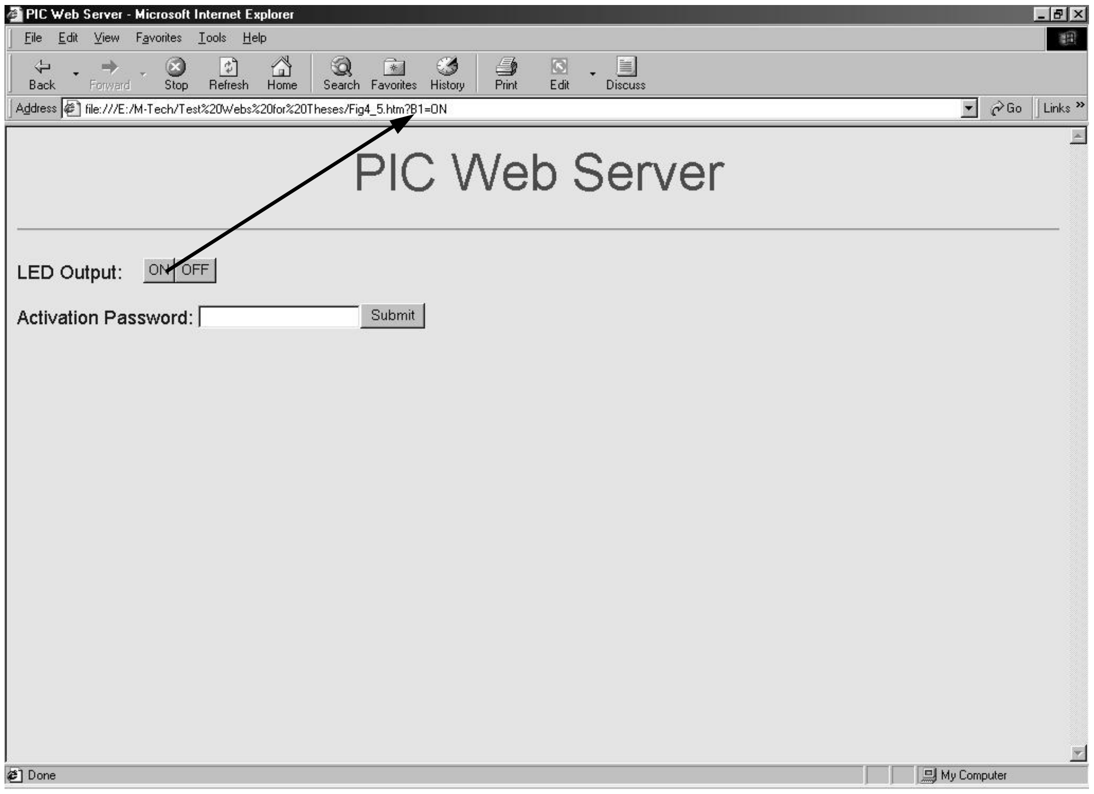
Address: file:///E:/M-Tech/Test%20Webs%20for%20Theses/Fig4_5.htm?pass=testing&B2=Submit

PIC Web Server

LED Output: ON OFF

Activation Password: testing Submit

File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Go Links
Done
My Computer


PIC Web Server - Microsoft Internet Explorer

Address: file:///E:/M-Tech/Test%20Webs%20for%20Theses/Fig4_5.htm?B1=ON

PIC Web Server

LED Output: ON OFF

Activation Password: Submit

File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Go Links
Done
My Computer

Figure 4.5 continued.

4.5 ROM FILE SYSTEM

HTML documents, images and other files are normally stored on the server on hard disks. If a client requests information on Web pages from the server, the documents are simply read from the hard disk and transferred using the HTTP protocol.

Hard disks are, however, not usually found in 8-bit embedded systems because of their complexity. This means that some other kind of memory device is needed to store the information documents for the server. This project makes use of a single 32Kb inter-IC (I^2C) serial EEPROM (24LC256) and basically simulates a hard disk system.

A client will need to find a file in the system using the filename as an identifier. This implies a file directory of some sort, preferably in a block at the start of the ROM. This will allow the client to search the directory using sequential read cycles until the desired file is found and will minimize the number of address select cycles that will have to be emitted. The elements required in the directory are:

- The length of the file in bytes
- A pointer to the start of the file contents in ROM
- A TCP checksum for the file
- Flags to enable EGI variable substitution
- A filename (lowercase, 8.3 format)

This results in a directory structure in the firmware as indicated in Figure 4.6.

```
#define ROM_FNAMELEN    12              /* Maximum filename size */

typedef struct                          /* Filename block structure */
{
    WORD len;                           /* Length of file in bytes */
    WORD start;                         /* Start address of file data in ROM */
    WORD check;                         /* TCP checksum of file */
    BYTE flags;                         /* Embedded Gateway Interface (EGI) flags */
    char name[ROM_FNAMELEN];            /* Lower-case filename with extension */
} ROM_FNAME;

/* Embedded Gateway Interface (EGI) flag values */
#define EGI_ATVARS        0x01          /* '@' variable substitution scheme */
#define EGI_HASHVARS      0x02          /* '#' and '|' boolean variables */
```

Figure 4.6 Directory structure in firmware.

The end of the directory area in ROM is identified by an entry with a dummy length value of 0xFFFF.

The following is also required for the file system:

- **Default page**
  The default page, "index.htm", must be easy to find, so it must be the first file in the ROM.
- **HTTP header**
  String manipulation is difficult on a PIC microcontroller because of the lack of RAM. So instead of identifying the file type and prefixing the file with the correct HTTP response header (including the content-type), the files are stored on the ROM with their headers already attached.
- **EGI flags**
  The EGI flags need to be set if EGI variable substitution is to be performed on the file as it is sent out.

Figure 4.7 illustrates a simplified file system in a ROM.

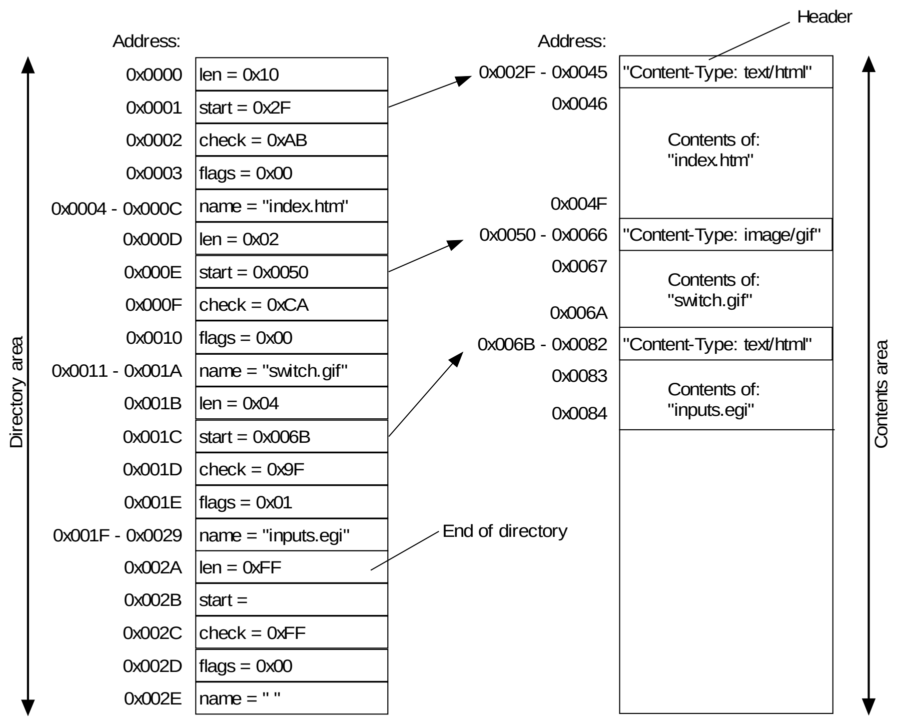


Figure 4.7 Example of a file system in a ROM.

4.6 WEB SERVER DRIVER

Figure 4.8 shows the section of the HTTP software driver that detects the GET method in the HTTP request from the client.

```
if (match_byte('G') && match_byte('E') && match_byte('T'))                                1
  {
     ret = 1;
     match_byte(' ');
     match_byte('/');                             // Start of filename
     DEBUG_PUTC(' ');
     memset(romdir.f.name, 0, ROM_FNAMELEN);
     for (i=0; i<ROM_FNAMELEN && get_byte(c) && c>' ' && c!='?'; i++)
     {                                            // Name terminated by space or '?'      2
        DEBUG_PUTC(c);
        romdir.f.name[i] = c;
     }                                            // If file found in ROM
     if (find_file())                                                                     3
     {                                            // ..check for form arguments
        DEBUG_PUTC('>');
        check_formargs();                                                                 4
     }
     else                                         // File not found, get index.htm
     {
        DEBUG_PUTC('?');                                                                  5
        romdir.f.name[0] = 0;
        find_file();
     }
     checkhi = checklo = 0;
     checkflag = 0;
     txin = IPHDR_LEN + TCPHDR_LEN;
     if (!fileidx)                                // No files at all in ROM - disaster!
     {
        setnic_addr((TXSTART<<8)+sizeof(ETHERHEADER)+IPHDR_LEN+TCPHDR_LEN);
        printf(putnic_checkbyte, HTTP_FAIL);
        tflags = TFIN+TACK;
        d_checkhi = checkhi;
        d_checklo = checklo;
        tcp_xmit();
     }
     else                                         // File found OK
     {
        open_file();                              // Start i2c transfer
        setnic_addr((TXSTART<<8)+sizeof(ETHERHEADER)+IPHDR_LEN+TCPHDR_LEN);                 6
        while (tx_file_byte())                    // Copy bytes from ROM to NIC
            ;
        close_file();
        tflags = TFIN+TPUSH+TACK;                 // Close connection when sent
        d_checkhi = checkhi;                      // Save checksum
        d_checklo = checklo;
        tcp_xmit();                               // Do header, transmit segment
     }
  }
```

Figure 4.8 A section of the HTTP driver that detects the GET method.

1. The if.. statement reads the characters from the HTTP protocol’s data area. It makes use of three **match_byte()** function calls to determine if the first three characters correspond with “GET” (the request line from the client).
2. The for.. loop then reads the rest of the characters directly after the GET method until it reaches a blank space or the ‘?’ character, which represents the end of the file name.
3. The ROM file system is then searched to see if the filename exists in the directory.
4. If the file is found, the **check_formarg()** function is called, which will simply just check each of the characters after the ‘?’ character in the request line to determine what the command to the I/O controller should be. The contents of the check_formargs() function will depend on the application self.
5. If the file is not found in ROM, then the default page “index.htm” is loaded.
6. Once the file is opened, the HTTP header for that particular file is transmitted. The **tx_file_byte()** function (Figure 4.9) will then transmit the contents of the file byte by byte.

```
BOOL tx_file_byte(void)
{
   int ret=0;
   BYTE b,c;

   if (romdir.f.len)                                 // Check if any bytes left to send
   {                                                                                        1
      b = i2c_read(1);                               // Get next byte from ROM
      if ((romdir.f.flags&EGI_ATVARS) && b=='@')                                            2
      {                                              // If '@' and EGI var substitution..
         b = i2c_read(1);                            // ..get 2nd byte                      3
         romdir.f.len--;
                   while(bit_test(PIR1, 5))          // clear input buffer
                             getch();
                   putchar('?');
                   putchar(b);
                   while(!bit_test(PIR1, 5));        // wait for response                   4
                   do
                   {
                             c = getch();
                             if (c != '~')
                                       printf(putnic_checkbyte, "%c", c);
                   } while(c != '~');
      }
      else                                           // Non-EGI byte; send out unmodified
         putnic_checkbyte(b);                                                               5
      romdir.f.len--;                                // Decrement length
      ret = 1;
   }
   return(ret);
}
```

Figure 4.9   The tx_file_byte() function.

1. The driver makes use of the CCS function **i2c_read()** to read the file from ROM.
2. For every byte it reads from ROM, it checks if the byte corresponds with the ‘@’ EGI character.
3. If the ‘@’ character is detected and the EGI flags for the particular file is set (section 4.5), then the next byte is read to determine the variable number that is requested.
4. The driver then makes use of a RS-232 protocol (will be explained in Chapter 5) to communicate with the I/O controller to request a particular variable’s value. The result is then transmitted as part of the contents of the document.
5. If the ‘@’ character is not detected, then the byte that was read from ROM is then just simply transmitted as it is using the **putnic_checkbyte()** function. The **putnic_checkbyte()** function updates the Ethernet packet’s checksum on the fly, i.e. as it adds data to the Ethernet packet.

4.7 CONCLUSION

The chapter explained all the building blocks for a Web server and how to implement them on an embedded system. Certain changes to standard HTML documents also need to be made to allow dynamic Web pages. These dynamic pages make embedded Ethernet applications very powerful, so these methods were explained in detail. The chapter then concluded with a brief explanation of certain sections of the HTTP driver.

Chapter 5 will cover the next section of the project: the Input/Output controller. The I/O controller will control all the inputs and outputs to the system and somehow it must be able to communicate with the Web server. This protocol will be explained along with a detailed description of the firmware on the I/O controller.

CHAPTER 5
THE INPUT/OUTPUT CONTROLLER

5.1 OVERVIEW

The purpose of this project is to be able to control and monitor processes. This controlling and monitoring is achieved by implementing a microcontroller of which the main purpose is to simply control its outputs and monitor its inputs. The microcontroller also needs to be totally dynamic towards software upgrades, application changes etc., and therefore needs to be able to be reprogrammed while it is still in the circuit.

This chapter will describe this input/output (I/O) controller interface in detail and will also explain the necessary firmware that is needed for this interface.

5.2 INPUT/OUTPUT CONTROLLER SELECTION CRITERIA

Microchip's PIC16F877 was chosen as the I/O controller for this project and the reasons for selecting it can be summarised as follows:

5.2.1 REPROGRAMMABILITY

The aim of the whole project is to be totally reconfigurable and therefore this is also the main selection criterion for a suitable microcontroller. PE Technikon has all the development tools for the PIC16F877 and that was probably the main reason for choosing this microcontroller.

Microchip makes use of an ICSP technique, which allows a microcontroller to be programmed after the device is placed in a circuit board (In-circuit serial programming guide, p.1). Because these devices then accommodate rapid code changes, they offer tremendous flexibility and reduce development time considerably.

5.2.2 PROGRAMMING INTERFACE

PIC16F877 microcontrollers can be programmed at a lower voltage (+5V) than the normal +12V required by conventional microcontrollers. This makes it very suitable for this application for the simple reason that all the devices on the board operate from a single +5V power supply and therefore a second +12V power rail will be unnecessary.

In low voltage programming mode, the PIC16F877 uses only 4 of the microcontroller's pins, as indicated in Table 5.1. Again this is a major advantage because there will be more pins available for the controlling or monitoring functions.

Pin Name	During programming		
	Function	Pin Type	Description
RB3	PGM	INPUT	Low voltage ICSP input
RB6	CLOCK	OUTPUT	Clock output
RB7	DATA	INPUT/OUTPUT	Data input/output
MCLR	$V_{TEST\ MODE}$	PROGRAMMING	Program mode select

Table 5.1 Programming pins description for the PIC16F877.

During normal operation, pins RB6 and RB7 can be used for other functions, e.g. input/output.

5.2.3 DEVELOPMENT SOFTWARE

Microchip has their own Integrated Development Environment (IDE) software package called MPLAB. The package is totally free and can be downloaded from www.microchip.com, which makes it very appealing for embedded software developers. MPLAB also allows the use of third-party c-compilers.

5.2.4 FEATURES AND PERIPHERALS

The PIC16F877 is rich with features and peripherals:

- It has a high performance reduced instruction set computer (RISC) CPU.
- Operating speed of DC up to 20Mhz.
- 8K x 14 words of FLASH Program Memory.
  368 x 8 bytes of Data Memory (RAM).
  256 x 8 bytes of EEPROM Data Memory.
- Interrupt capability (14 sources).
- 8 Level hardware stack.
- Power-on Reset (POR).
- Power-up Timer (PWRT) and Oscillator Start-up Timer (OST).
- Watchdog Timer (WDT) with its own on-chip RC oscillator.

- Power saving SLEEP mode.
- High sink/source current: 25mA.
- Low power consumption.
- 3 Timers.
- 2 Capture, Compare, pulse width modulation (PWM) modules.
- 10-bit multi-channel Analog-to-Digital converter.
- Synchronous Serial port (SSP) with Serial Peripheral Interface (SPI) and I^{2}C.
- Universal Synchronous Asynchronous Receiver Transmitter (USART) with 9-bit address detection.
- Brown-out detection circuitry for Brown-out Reset (BOR).

5.3 ANALOG INTERFACE

An analog interface is also added to the I/O controller to allow the measurement of analog values in real-time. This interface is indicated in Figure 5.1.

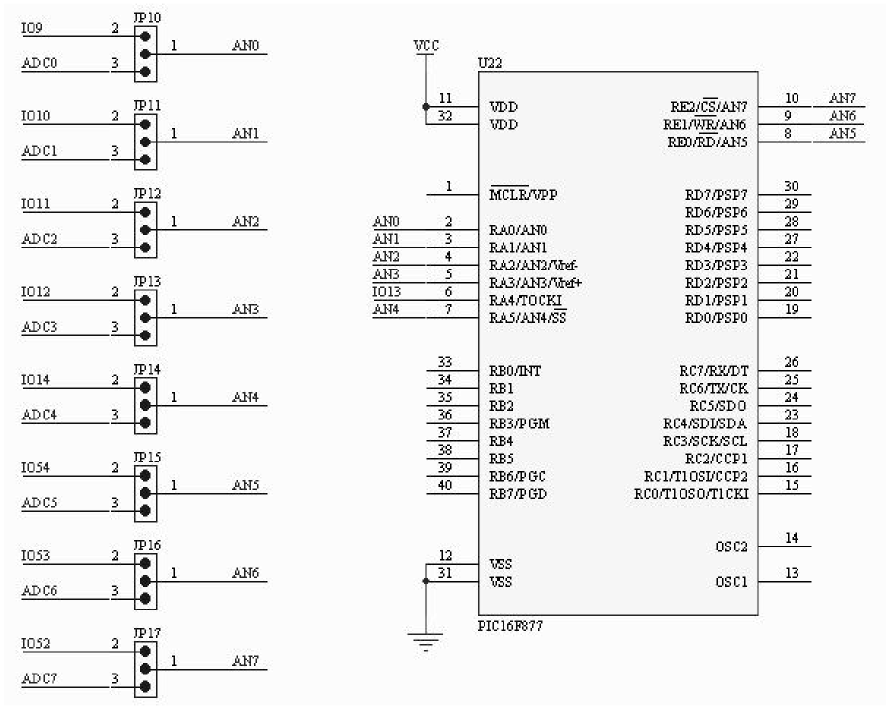


Figure 5.1 I/O controller analog interface.

The pins from port A (except RA4) and port E are used as the analog inputs to the microcontroller as shown in Table 5.2.

Pin	Description
RA0	Channel 0
RA1	Channel 1
RA2	Channel 2 OR negative reference voltage (V_{ref-})
RA3	Channel 3 OR positive reference voltage (V_{ref+})
RA5	Channel 4
RE0	Channel 5
RE1	Channel 6
RE2	Channel 7

Table 5.2 Analog pins configuration.

These pins can also be used as normal inputs or outputs through the use of jumpers JP10 to JP17. If a jumper is in the position 1-2, that particular microcontroller pin will be used as an input or output, and if it is in the 1-3 position it will be used as a pure analog input. Pins RA2 and RA3 can be used as normal analog inputs or as voltage reference inputs for the PIC16F877's analog-to-digital converter, e.g. if a analog sensor has an output voltage between +1.5V and +3V, then to make use of the full 10-bit conversion, the user must set the V_{ref-} pin to exactly +1.5V (maybe through the use of a potentiometer) and the V_{ref+} pin to +3V. Figure 5.2 shows an example for using the analog interface.

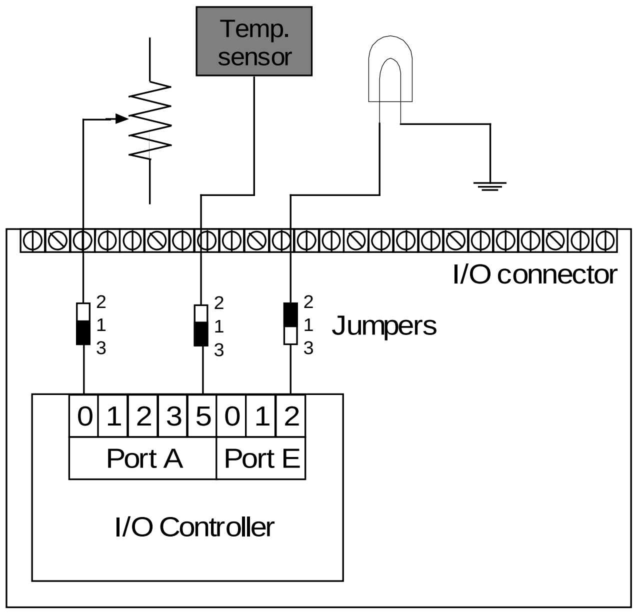


Figure 5.2 Analog interface example.

5.4 LIQUID CRYSTAL DISPLAY INTERFACE

The board has a connector for a LCD, which is mainly used by the Ethernet interface controller (as explained in Chapter 3). This LCD can, however, also be used by the I/O controller through the use of jumpers JP1 to JP8 and JP18 to JP24 as indicated in Figure 5.3.

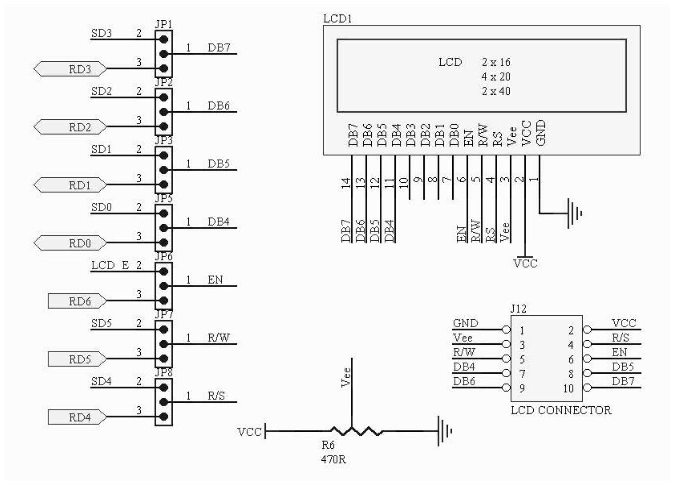


Figure 5.3 Liquid crystal display interface.

Potentiometer R6 is simply the contrast control for the LCD. The LCD's pins are routed to port D of the I/O controller if it is assigned to it. Table 5.3 summarizes the jumper settings for assigning the LCD.

JP1 to JP8	JP18 to JP24	Assigned to:	I/O controller pins RD0 to RD6
1-2	1-2	Ethernet interface controller	Used as I/O pins
1-2	1-3	Ethernet interface controller	Used for LCD, but LCD assigned to Ethernet interface controller.
1-3	1-2	I/O controller	Used as I/O pins, therefore LCD will not work even it is assigned.
1-3	1-3	I/O controller	Used for LCD

Table 5.3 Jumper settings for LCD assignment.

Often the user will need the LCD for the Ethernet interface, but will also need an additional LCD to be connected to the I/O controller. In this case an external LCD can be connected directly to the I/O connector pins IO30 to IO36 and then port D of the I/O controller can be programmed to drive the LCD (provided that jumpers JP18 to JP24 are in the 1-2 position). This is demonstrated in Figure 5.4.

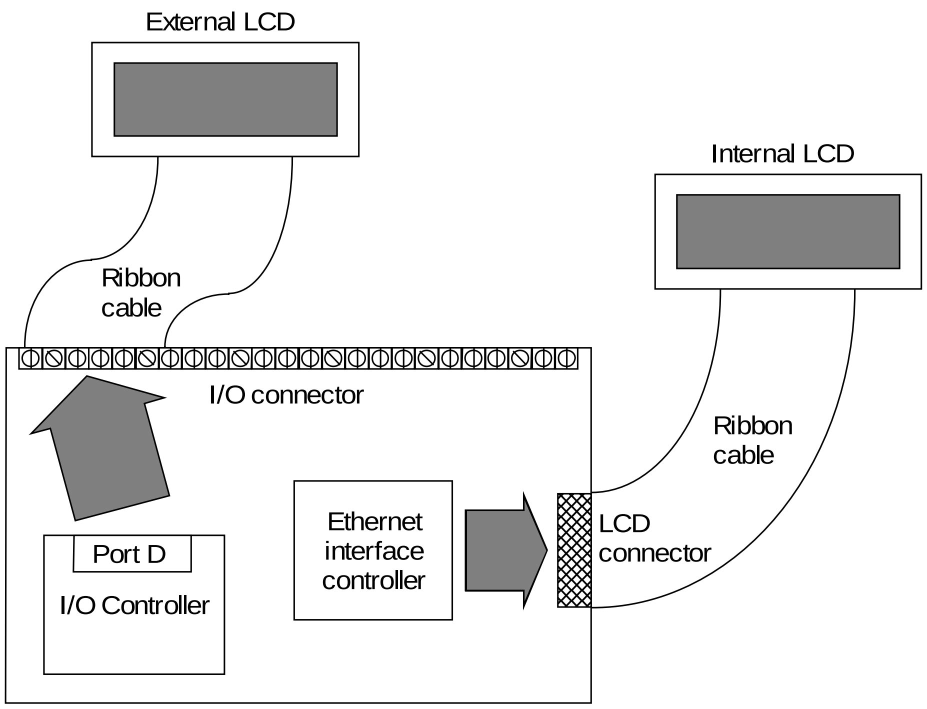


Figure 5.4 External LCD connection example.

5.5 I/O INTERFACE

The remaining port pins of the PIC16F877 can be used as general I/O pins. These pins are routed directly to the I/O connector and can be used to connect external peripherals, sensors and actuators to the board. The remaining pins are also routed to a reconfigurable hardware interface, which will be explained in the next chapter.

5.6 INFORMATION EXCHANGE PROTOCOL

The I/O controller operates independently from the Ethernet interface and uses most of its resources to fulfil a controlling or monitoring function. This process can, however, be coupled to an embedded Web page as described in Chapter 4, and therefore some kind of information exchange method between the Web server and the I/O controller needs to be established.

5.6.1 REQUIRED HARDWARE

To minimize the amount of pins used on the I/O controller, it was decided to use a USART link between the Web server and the I/O controller. This means that port pins RC6 and RC7 will not be available to the user for normal I/O functions. Figure 5.5 indicates the USART interface.

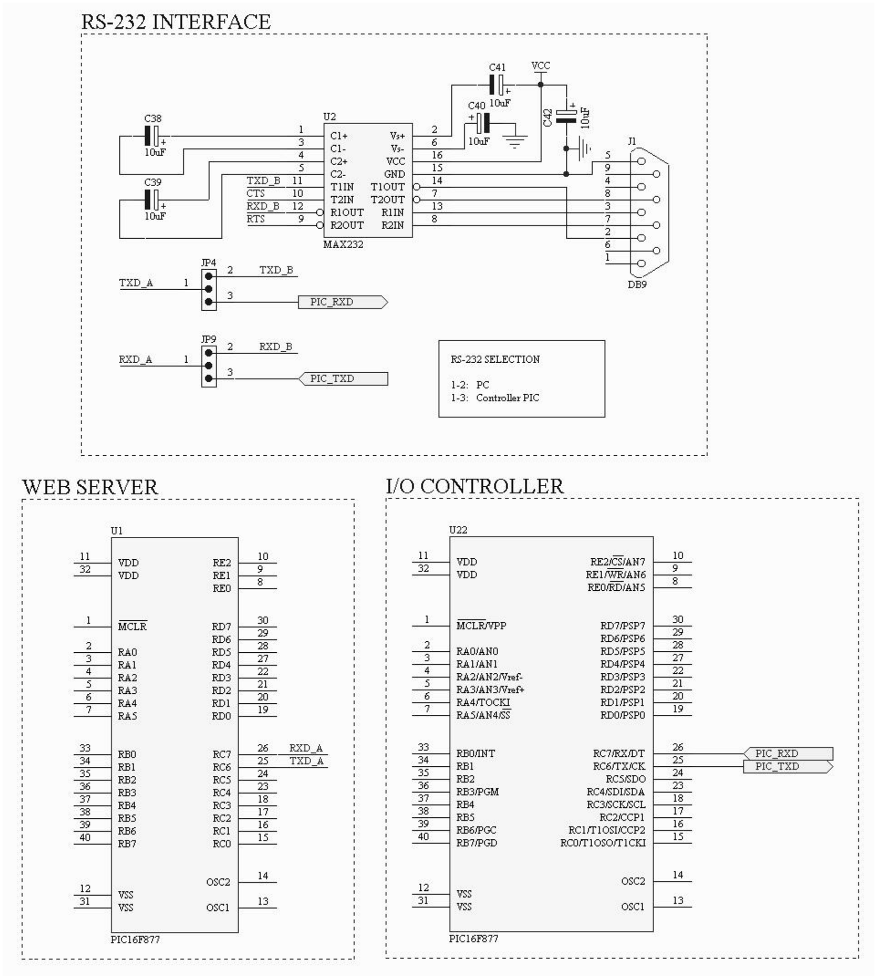


Figure 5.5 USART interface between the Web server and I/O controller

Jumpers JP4 and JP9 are used to select the function of the Web server's USART module. If these jumpers are in the 1-3 position, the Web server's RXD and TXD pins are used to communicate with the I/O controller. The jumpers could also be set to the 1-2 position, which will divert the Web server's USART module to a MAX232 transceiver to allow communication with a PC. The last option was added purely for development reasons and note must be taken that no communication can take place between the Web server and the I/O controller when making use of this feature.

5.6.2 PROTOCOL DEFINITION

The Web server uses the USART link for two reasons:

- To request data from the I/O controller, e.g. the value of the temperature variable.
- To issue a command to the I/O controller, e.g. switch pin RB6 on.

These commands and requests are built into the embedded Web pages as described in Chapter 4. It is the task of the Web server to issue the requests and the commands to the I/O controller using the following scheme:

<u>To send a request for a certain variable value:</u>

1. Clear the receive buffer.
2. Send the '?' character followed by the character describing the requested variable.
3. Wait for a response.
4. Read the incoming data until the '~' character is received.
5. Transmit this data as part of the Web page.

<u>To issue a command to the I/O controller:</u>

1. Send the '!' character.
2. Send the character describing the output to change.

5.6.3 PROTOCOL IMPLEMENTATION

The implementation of the protocol occurs at the HTTP level.

<u>For sending a request:</u>

When a client requests a Web page, the Web server uses the **tx_file_byte()** function to transfer the file byte for byte from the I^2C serial EEPROM as shown in Figure 5.6.

1. Read the next character from the file to transmit as a Web page.
2. If the character is a variable substitution EGI flag ('@'), then get ready to send a request to the I/O controller using the information exchange protocol.
3. Read the byte that describes the variable that needs to be accessed, e.g. 2 in the case of the HTML code: "the temperature is @2 degrees Celsius".
4. Clear the Web server's USART receive buffer to make sure that the data is received correctly.

5. Send the ‘?’ character followed by the character describing the requested variable.
6. Wait in a loop until a response is received from the I/O controller.
7. Read the data byte for byte as it is sent from the I/O controller until the STOP (‘~’) character is read. As the data is received, it must be transmitted as part of the Web page file.
8. If no data is requested, then send the character as part of a normal Web page.

```
/* Transmit a byte from the current i2c file to the NIC
** Return 0 when complete file is sent
** If file has EGI flag set, perform run-time variable substitution */

BOOL tx_file_byte(void)
{
   int ret=0;
   BYTE b,c;

   if (romdir.f.len)                                  // Check if any bytes left to send
   {
      b = i2c_read(1);                                // Get next byte from ROM                 1
      if ((romdir.f.flags&EGI_ATVARS) && b=='@')                                                2
      {                                               // If '@' and EGI var substitution..
         b = i2c_read(1);                             // ..get 2nd byte                         3
         romdir.f.len--;
         while(bit_test(PIR1, 5))                     // clear input buffer                     4
            getch();
         putchar('?');                                                                          5
         putchar(b);
         while(!bit_test(PIR1, 5));                   // wait for response                      6
         do
         {
            c = getch();
            if (c != '~')                                                                       7
               printf(putnic_checkbyte, "%c", c);     // transmit data as it is read
         } while(c != '~');                           // read data until STOP character ‘~’ is read
      }
      else                                            // Non-EGI byte; send out unmodified     8
         putnic_checkbyte(b);
      romdir.f.len--;                                 // Decrement length
      ret = 1;
   }
   return(ret);
}
```

Figure 5.6 Code to send a request to the I/O controller.

For sending a command:

The **check_formargs()** function implements the information exchange protocol for sending commands to the I/O controller. Sending a command currently means to change the output status of port pins on the I/O controller. To implement text inputs etc. from a Web page will require minor modification to the **check_formargs()** function, but has not been implemented in the current software drivers.

To implement sending a command to the I/O controller in a Web page, the following will be required in the HTML code:

- Add a **pushbutton** to the page.
- Change the **form method** to “SUBMIT”.
- Change the **form action** to the name of the Web page.
- Make sure the name of the pushbutton (or input) starts with a ‘Q’ followed by a number, e.g. “Q2”.

The following line gives an indication of what the Web page request to the server looks like when a pushbutton (Q0) is clicked:

```
http://www.petech.ac.za/picweb/Outputs.egi?Q0.x=10&Q0.y=14
```

Figure 5.7 indicates the protocol implementation in the **check_formargs()** function.

```
/* Check for arguments in HTTP request string */
void check_formargs(void)
{
   char c;

   if (match_byte('Q'))          // detects if a valid command is going to be issued      1
   {
      get_byte(c);               // get byte that describes the output to change          2
      putchar('!');              // send the command character to the I/O controller      3
      putchar(c);                // send the output to change
   }
}
```

Figure 5.7 Code to change the output status of port pins on the I/O controller.

1. Check if a valid output needs to be changed.
2. Read the value of the output that needs to be changed.
3. Send the command character (‘!’) to the I/O controller via the USART link.
4. Send the value of the output that needs to be changed.

5.6.4 I/O CONTROLLER DRIVER

The I/O controller also requires a small portion of code to complete the information exchange protocol. This code must always be included in any software that is written for the I/O controller and basically acts as a template for developers as indicated in Figure 5.8.

```
#include <16F877.H>
#device *=16


// ******************** Required Code ********************
#use DELAY(CLOCK=20000000)                                    // setup crystal frequency
#use RS232 (BAUD=115200, XMIT=PIN_C6, RCV=PIN_C7, ERRORS)     // setup USART module         1
#define STOP   putchar('~')                        // STOP character for protocol

#int_rda                                           // interrupt routine for receiving a command / request   2
void serial_isr(void)
{
   char b;
   disable_interrupts(global);
   b = getch();                                    // read character describing a request or command   4
   if (b == '?')                                   // request
   {                                                                                                5
      b = getch();                                 // read request number
      switch(b)
      {
         case '0': {                               // example of sending a reply to a request
                      if (input(PIN_A0))
                         printf("0");              // use printf() for reply
                      else                                                                          6
                         printf("1");
                      STOP;                        // send STOP character after data has been send
                   }
                   break;
      }
   }
   if (b == '!')                                   // command
   {                                                                                                7
      b = getch();                                 // read command number
      switch(b)
      {
         case '0': {                               // example of receiving a command
                      if (LED0)
                         LED0 = 0;
                      else                                                                          8
                         LED0 = 1;
                   }
                   break;

      }
   }
   enable_interrupts(global);                                                                       9
}
// ******************************************************
```

Figure 5.8 Necessary code to include when developing code for the I/O controller.

```
void main(void)
{

// ******************** Required Code ********************
   set_tris_c(0x80);                          // setup RXD and TXD pins for USART
   enable_interrupts(global);                                                        10
   enable_interrupts(int_rda);                // enable interrupt on USART receive
// ***********************************************************

   while(1)                                   // main program loop
   {                                                                                 11
      …………………..;
   }
}
```

Figure 5.8 continued.

1. Set the crystal frequency to 20Mhz and setup the USART module for 115200 baud rate. Also define the '~' character as STOP.
2. The interrupt service routine will be invoked as soon as the Web server sends data (command or request) via the USART link.
3. All other interrupts must be disabled during the interrupt routine to ensure the correct operation of the information exchange protocol.
4. Read the first incoming character which describes if the message is a command or a request.
5. If it is a request ('?'), then read the request number.
6. Reply to the request number using **printf()** functions ending the transmission with a STOP.
7. If it is a command ('!'), then read the command number.
8. Act on the incoming command by changing the status of port pins etc.
9. Other interrupts can now be enabled again.
10. Setup the TRISC register to allow the correct operation of the RC6(TX) and RC7(RX) pins.
11. This is simply the main loop the developer will use to control a certain process as with normal embedded applications.

Chapter 9 will cover complete examples and will demonstrate the information exchange protocol more clearly.

5.7 CONCLUSION

The main purpose of this chapter was to explain the function of the I/O controller and how to interface external applications to it. The most important feature of the I/O controller, namely the information exchange protocol, was explained in detail with reference to the Web server. All the necessary code for implementing the protocol was also explained.

Chapter 6 will introduce the next dynamic component in the project: the reconfigurable hardware. A brief introduction to programmable logic will be given, after which there will be an in-depth explanation of how the reconfigurable hardware is implemented in the project and how it is interfaced with the rest of the circuit.

CHAPTER 6
RECONFIGURABLE HARDWARE

6.1 OVERVIEW

This chapter will cover the reconfigurable hardware component of the project. A brief introduction to PLDs will be given after which the MAX7000S series of PLDs will be introduced. The complete hardware interface will be described and the way in which it fits in with the I/O controller will be explained.

6.2 INTRODUCTION TO LOGIC DEVICES

A brief introduction to logic devices will follow to provide a background to PLDs.

6.2.1 STANDARD LOGIC FAMILIES

6.2.1.1 Gate functions

The primary building block of logic circuits is the logic gate (Carter, 1997, p.2). This is a device which operates on two or more logic signals to give an output which is defined by a logic operator. The standard logic operators are AND, OR and INVERT.

6.2.1.2 Sequential logic

The output of a gate does not depend on the order in which the signals are applied (Bostock, 1996, p.3). If both inputs of a two-input AND gate are LOW, the output will also be LOW. If input A goes HIGH before B, or if B goes HIGH before A the result will be the same.

In the circuit in Figure 6.1 the order of the inputs does however make a difference to the result.

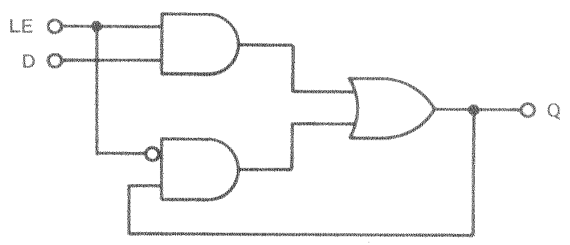


Figure 6.1 D-type latch circuit (Bostock, 1996, p.3).

If input LE is HIGH the output Q will be the same as input D. Suppose that LE is taken LOW. If D was HIGH when LE went LOW, Q will also be HIGH. Conversely, if D was LOW, Q will stay LOW. This circuit is known as a latch. The output of the circuit depends on the sequence in which the signals are applied, hence the term sequential circuit.

Figure 6.2 shows another sequential circuit known as a flip-flop.

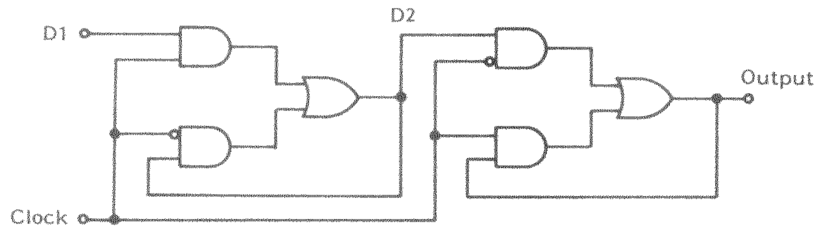


Figure 6.2 D-type flip-flop (Bostock, 1996, p.4).

6.2.1.3 Practical logic circuits

Devices containing one or more gates, latches or flip-flops form the basis of the standard logic families (Carter, 1997, p.6). Circuit designers can use these integrated circuits to build more complex functions by interconnecting these small-scale integrations (SSIs) on a PCB. However, the device manufacturers anticipated these requirements by producing medium-scale integration (MSI) parts which contain many of the standard circuit functions which can be built from gates and flip-flops. Typical examples of MSI functions are one-to-eight line decoder/demultiplexers, four-bit shift registers, and four-bit counters (Figure 6.3).

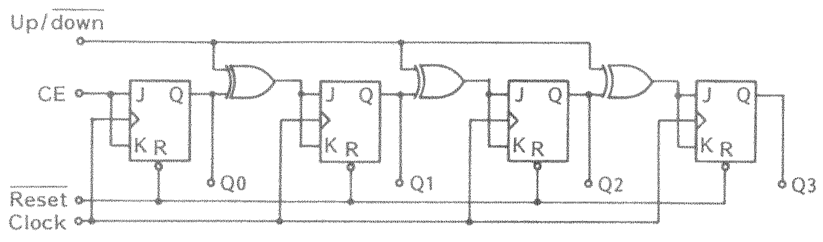


Figure 6.3 Four-bit counter (Bostock, 1996, p.9).

6.2.1.4 Large-scale integration

As processes improved to the point where a thousand or more transistors could be laid on a single chip, large-scale integration (LSI) became feasible. The situation is different to MSI. MSI functions can still be looked on as building blocks with universal application, e.g. a CD player or digital multimeter. LSI circuits are usually a self-contained function, of which the microprocessor is the best example. Apart from microprocessors, most LSI functions are specific to a particular application, e.g. an universal asynchronous receiver/transmitter (UART) will normally only be found in communications equipment and a frequency synthesiser in tuners (Carter, 1997, p.8).

6.2.2 PROGRAMMABLE LOGIC DEVICES

Circuit designers had only two options for building logic circuits: using the standard logic families or using devices called application specific integrated circuits (ASICs) (Bostock, 1996, p.17). With ASIC devices, designers had to design the layout of the LSI circuit and then had to have it developed by a company at high cost. If changes needed to be made to the ASIC, the whole layout had to be resubmitted and developed. This scenario changed very quickly with the evolution of the programmable switch.

6.2.2.1 Programmable switches

There are four types of programmable switches which have been used in programmable logic devices (Carter, 1997, p.20).

The metal fuse was the first type of switch and is traditionally associated with bipolar PLDs. An alloy, such as nichrome or tungsten-titanium, is evaporated onto the surface of the chip and etched into small strips. A current pulse of about 50mA is sufficient to vaporize the metal, which fuses into the overlying silicon dioxide leaving an open circuit at the fuse site.

An alternative fuse in bipolar technology is the avalanche-induced migration (AIM) device (Bostock, 1996, p.20). This is a small transistor with a floating base, so that the emitter-collector path is normally high impedance. If the emitter-base junction is deliberately overstressed, the aluminium from the emitter contact will migrate into the junction causing a short-circuit. The emitter-collector path is now a diode and can be used in its own right as a gating element.

In metal-oxide semiconductor (MOS) technology the transistors are, themselves, very efficient switches which can be turned on and off by applying HIGH or LOW signals to their gates. By adding a second gate, floating between the control gate and the conducting channel, the transistor threshold can be varied by charging or discharging the second gate. In the low threshold condition the transistor acts normally, but in the high threshold state the channel is held off permanently. The floating gate can be charged electrically but needs ultra-violet light to discharge it.

A later development is the antifuse (Carter, 1997, p.21). This is simply a thin layer of silicon oxide/nitride sandwiched between two conducting layers, which may be either silicon or metal. A short voltage pulse of 15V – 20V ruptures the insulating layer and the heat alloys the two layers together. A resistor of less than 1KΩ results, sufficiently low to appear as an ON switch to signals in a complementary metal-oxide semiconductor (CMOS) environment.

6.2.2.2 PAL devices

The most commonly used PLDs are programmable array logic (PAL) devices (Bolton, 1990, p.31). They are based on the idea that any combinational logic function can be represented by a "sum of products" equation. AND functions are sometimes referred to as product terms, and OR functions as sums terms. Sum of products means just the OR combination of a number of AND terms.

Figure 6.4 shows how a full adder can be built using only discrete logic. PAL devices can be incorporated to do exactly the same function as discrete logic as indicated in the example in Figure 6.5 using a PAL4H2 device.

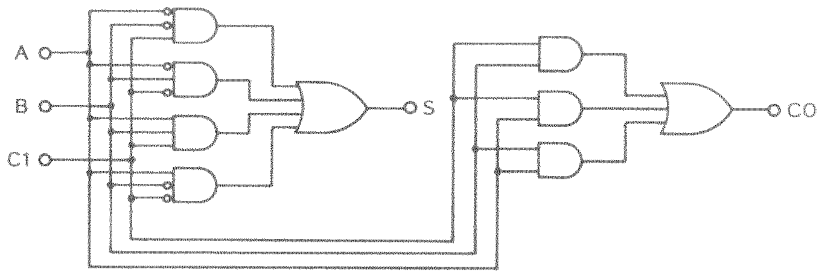


Figure 6.4 Full adder using discrete logic (adapted from Carter, 1990, p.27).

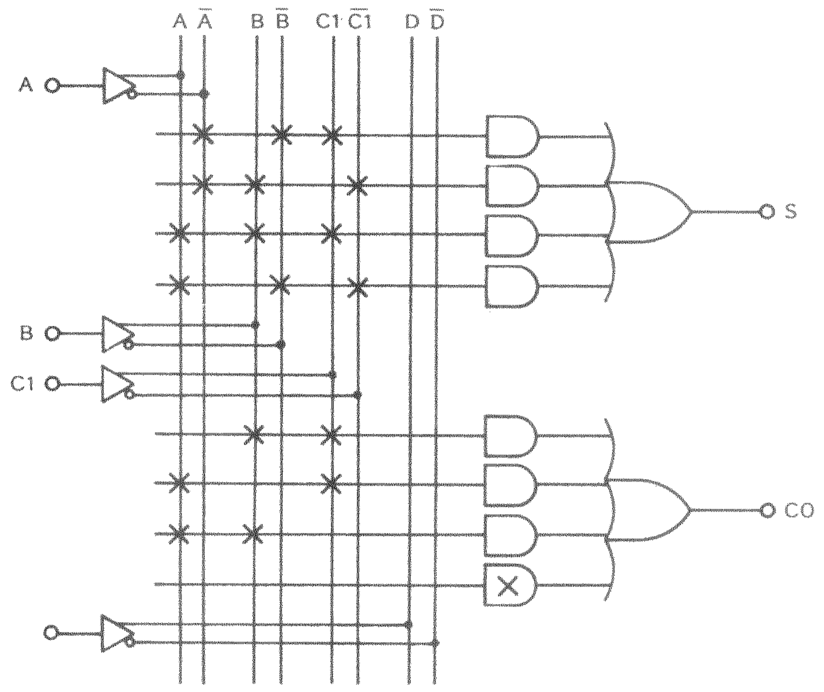


Figure 6.5 Full adder using a PAL4H2 device (adapted from Carter, 1990, p.27).

Each crossing point between a vertical signal line and the input line into each AND gate has a programmable switch which determines whether or not the signal is connected to the AND gate. The diagonal crosses indicate those fuses which are left intact for the application.

6.2.2.3 Generic PLDs, FPLAs and FPLSs

The drawback to some of the ranges of PALs is that designers were restricted to fixed architectures with eight, six, four or no flip-flops (Bostock, 1996, p.24). The introduction of generic macrocells made PLDs more flexible architecturally. Figure 6.6 shows a typical macrocell. The flexibility is achieved by the use of programmable mutliplexers to route the output signal through different paths.

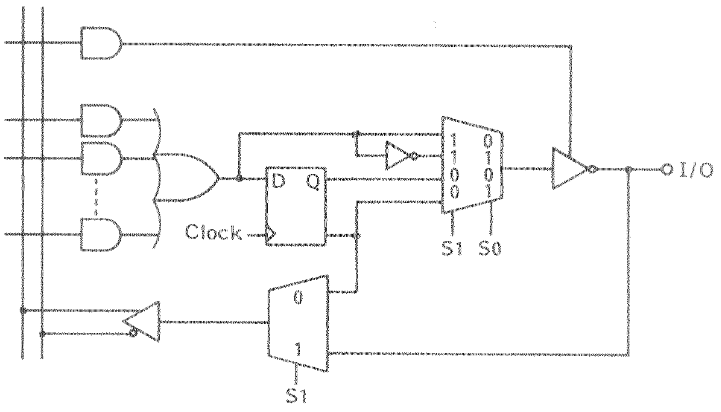


Figure 6.6 A typical macrocell (Bostock, 1996, p.25).

There is another class of PLD which can sometimes manage to overcome product term limitation problems with PALs and GALs; this is the field programmable logic array (FPLA). The difference is that they have 32 product terms which can be accessed by any output (Carter, 1990, p.28). This is achieved by having the OR gates connected by a programmable array to the AND gates, as can be seen in Figure 6.7.

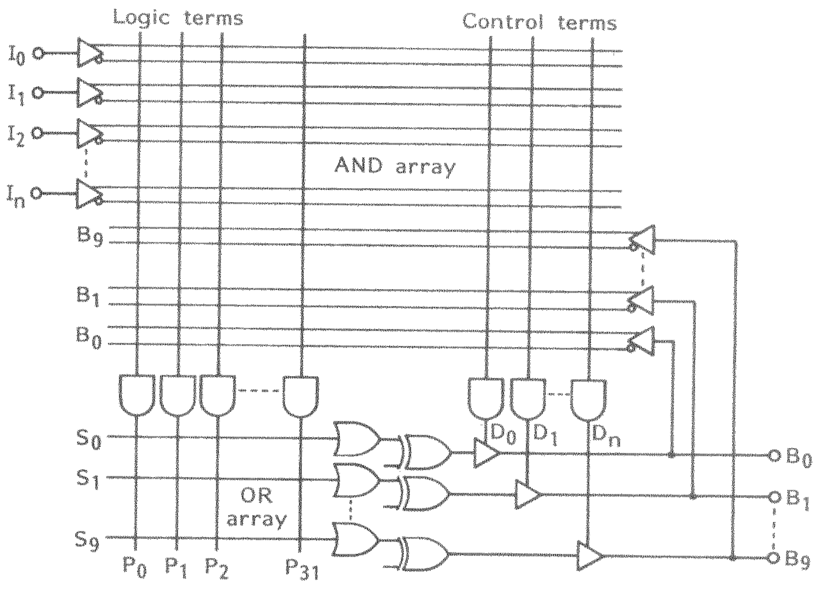


Figure 6.7 Programmable OR-array (Bostock, 1996, p.25).

A field programmable logic sequence (FPLS) device is simply a FPLA with flip-flops added to it as shown in the simplified schematic in Figure 6.8.

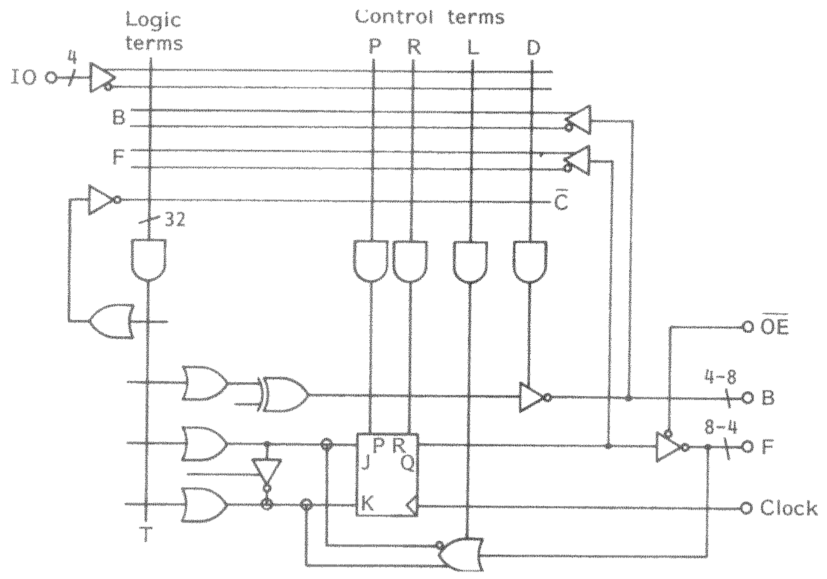


Figure 6.8 Simplified FPLS schematic (Bostock, 1996, p.26).

6.2.2.4 CPLDs

One approach to making PLDs with a higher logic content is to integrate several small PALs in one package (Bolton, 1990, p.41). Different manufacturers may differ in detail in the way they do this but they all have certain features in common. These devices are called complex PLDs (CPLDs) and the basic structure is indicated in Figure 6.9.

The input cells and I/O cells may be taken together as some families have no, or only a few, separate inputs and use I/O cells for this function. In general, an I/O cell connects the logic blocks to the outside world and features a transistor transistor logic (TTL) or CMOS interface. The outputs always have a tri-state capability which can be permanently enabled or disabled, or controlled by internal logic. They can function as dedicated input, dedicated outputs with optional feedback or as bussable outputs. The advantage of being able to choose the function of all I/Os is that the input and output sites are not pre-determined and this gives extra flexibility to the PCB layout (Bostock, 1996, p.30).

The interconnection matrix allows a reduced number of inputs to feed each logic block. It can be considered that each logic block is an individual PAL within the larger PLD, and that each PAL is connected to the others via the interconnection matrix.

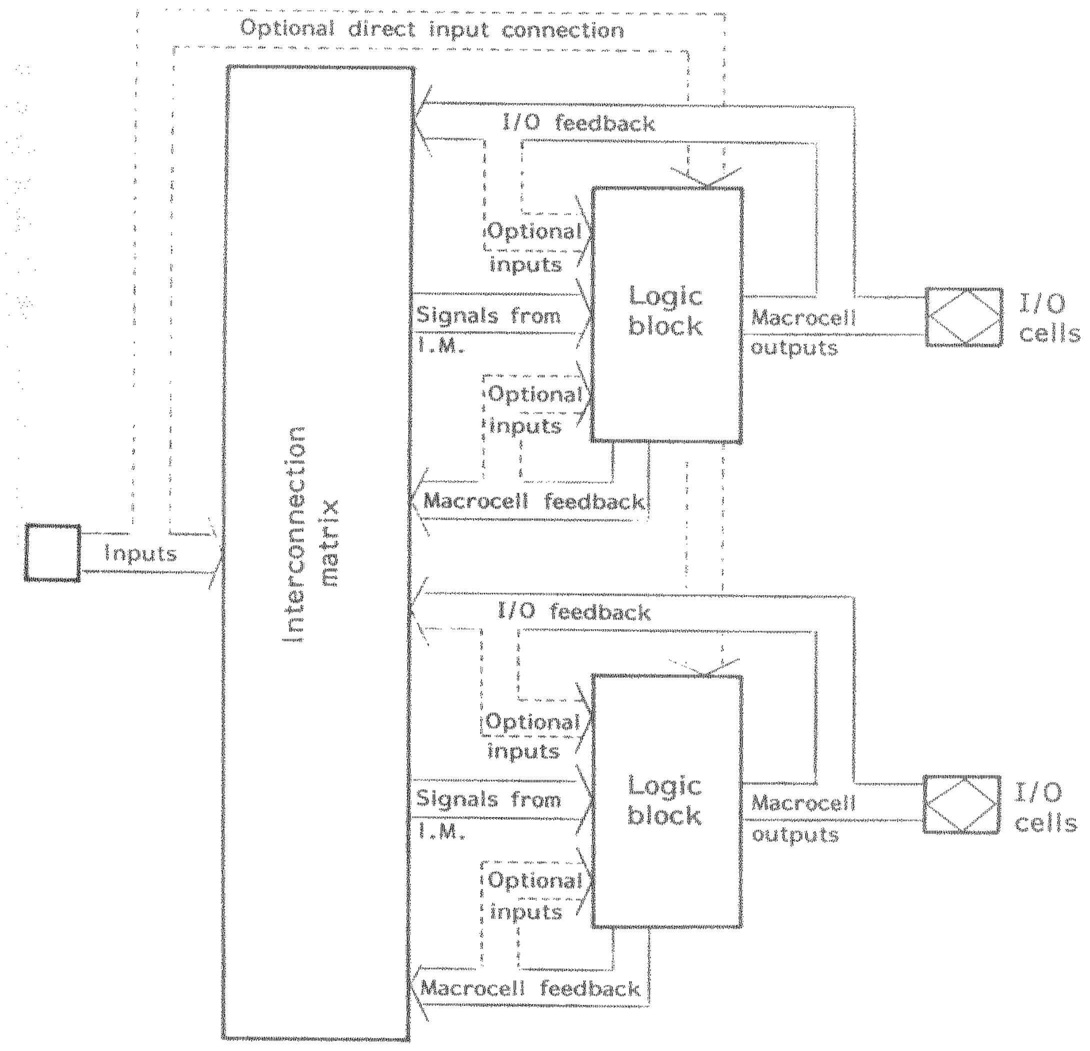


Figure 6.9 CPLD block diagram (Bostock, 1996, p.31).

6.3 MAX7000 DEVICES

6.3.1 ARCHITECTURE

Multiple array matrix (MAX) 7000 devices, introduced by Altera, conform to the general scheme of CPLD architecture, in that they contain logic blocks with a central interconnection matrix linking them together, logically. Figure 6.10 shows the block diagram of a typical MAX7000 device.

The interconnection matrix is called the programmable interconnection array (PIA); the logic block is a logic array block (LAB). All signals, including direct inputs, I/Os and macrocell feedbacks, pass through the PIA.

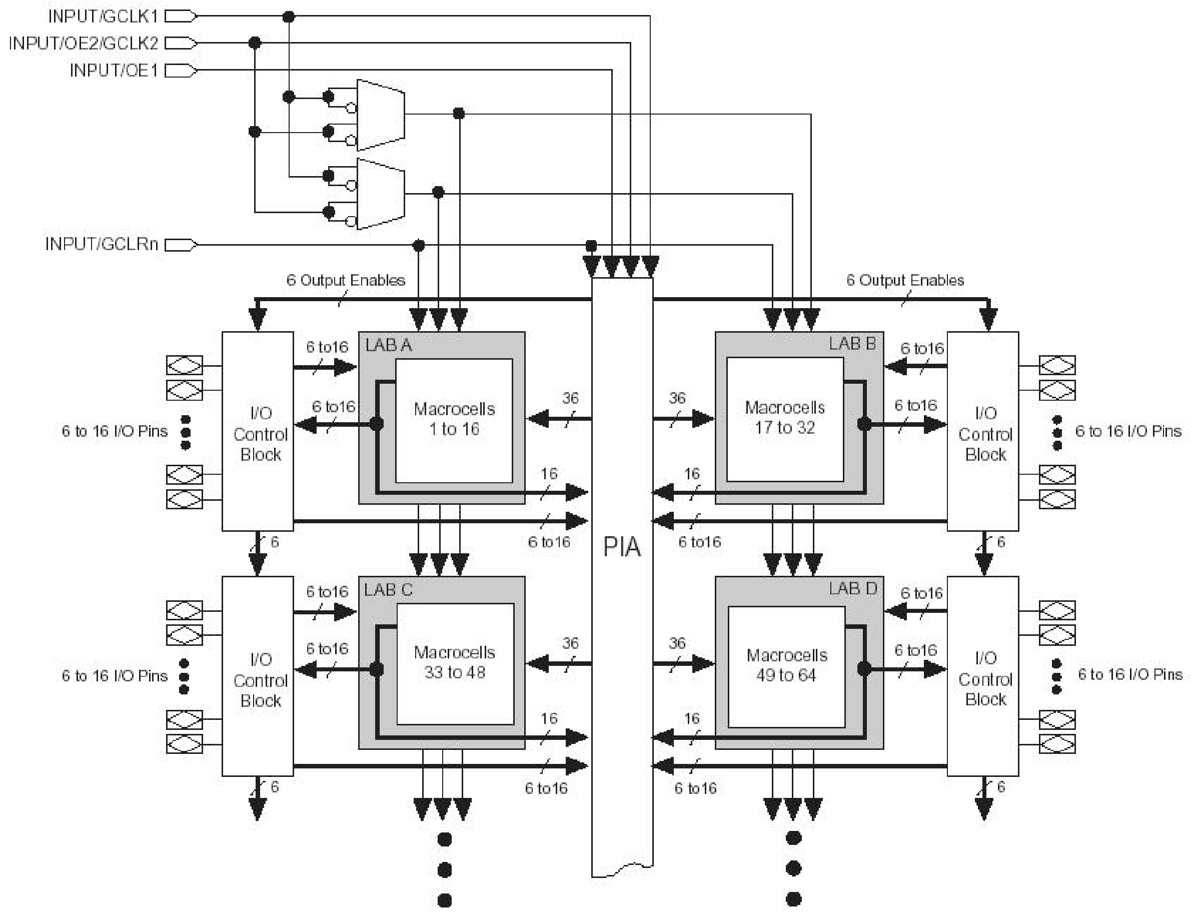


Figure 6.10 MAX7000 device block diagram (MAX7000 Programmable Logic Device Family, p.8).

6.3.1.1 Logic array blocks

The MAX7000 device architecture is based on the linking of high-performance, flexible, logic array modules called logic array blocks (LABs) (MAX7000 Programmable Logic Device Family, p.8). LABs consist of 16 macrocell arrays. Multiple LABs are linked together via the PIA, a global bus that is fed by all dedicated inputs, I/O pins and macrocells.

Each LAB is fed by the following signals:

- 36 signals from the PIA that are used for general logic inputs
- Global controls that are used for secondary register functions
- Direct input paths from I/O pins to the registers that are used for fast setup times

6.3.1.2 Macrocells

The macrocell can be individually configured for either sequential or combinational logic operation. The macrocell consists of 3 functional blocks: the logic array, the product-term select matrix, and the programmable register as indicated in Figure 6.11.

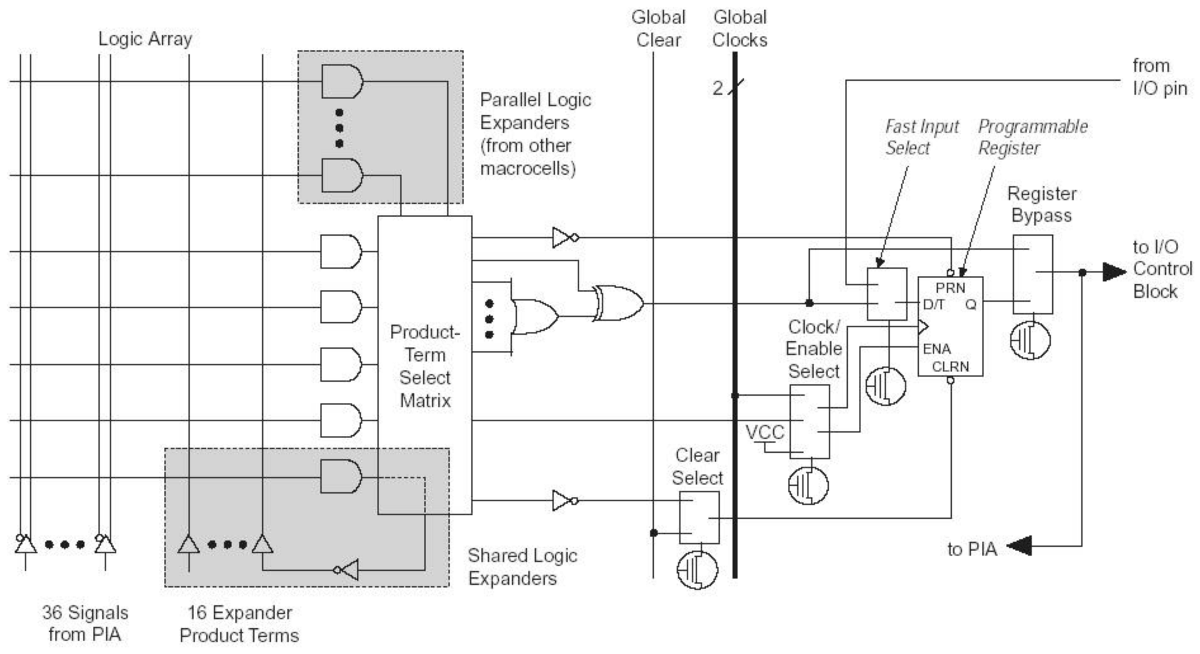


Figure 6.11 MAX7000 macrocell block diagram (MAX7000 Programmable Logic Device Family, p.9).

Combinational logic is implemented in the logic array, which provides five product terms per macrocell. Two kinds of expander product terms (“expanders”) are available to supplement macrocell logic resources (Bostock, 1996, p.79):

- Shareable expanders – inverted product terms that are fed back into the logic array
- Parallel expanders – product terms borrowed from adjacent macrocells

Each macrocell can be individually programmed to implement D, T, JK or SR operation with programmable clock control. The flip-flop can be bypassed for combinational operation.

Each programmable register can be clocked in three different modes:

- A global clock signal – achieves the fastest clock to output performance.
- A global clock signal and enabled by an active HIGH clock enable – provides an enable on each flip-flop while maintaining a fast clock to output performance.
- An array clock implemented with a product term – the flip-flop can be clocked by signals from buried macrocells or I/O pins.

Each register also supports asynchronous preset and clear functions, while the clear function can also be driven by an active LOW dedicated global clear pin (GCLRn).

6.3.1.3 Expander product terms

More complex logic functions require additional product terms than the five available in each macrocell (MAX7000 Programmable Logic Device Family, p.11). Another macrocell can be used to supply the required logic resources, but the MAX7000 architecture also allows both shareable and parallel expander product terms that provide additional product terms directly to any macrocell in the same LAB.

6.3.1.4 Programmable interconnect array

Logic is routed between LABs via the PIA (Bostock, 1996, p.79). The global bus is a programmable path that connects any signal source to any destination on the device. Figure 6.12 shows how PIA signals are routed into the LAB. An EEPROM cell controls one of the inputs to a 2-input AND gate, which selects a PIA signal to drive into the LAB.

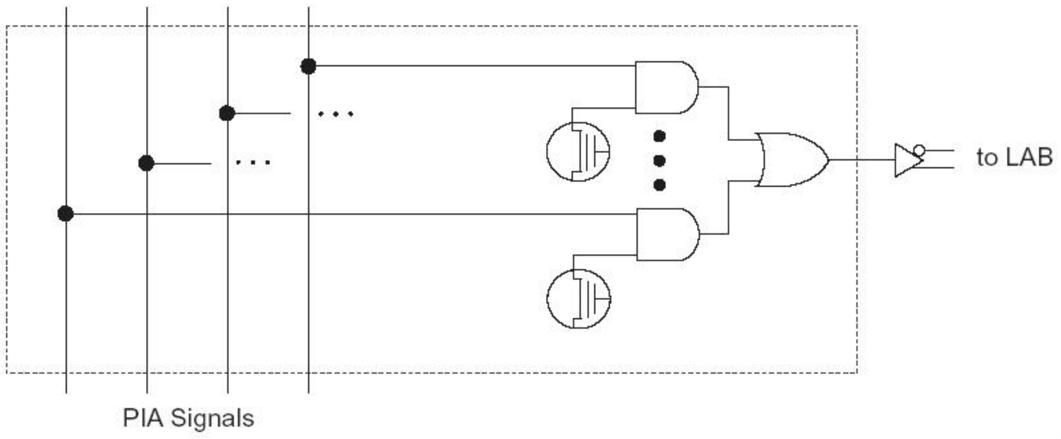


Figure 6.12 PIA routing (MAX7000 Programmable Logic Device Family, p.14).

6.3.1.5 I/O control blocks

The I/O control block allows each I/O pin to be individually configured for input, output, or bi-directional operation (MAX7000 Programmable Logic Device Family, p.14). All I/O pins have a tri-state buffer that is individually controlled by one of the global output enable signals or directly connected to ground or VCC. The I/O control block has six global output enable signals, a subset of the I/O pins, or a subset of the I/O macrocells. Figure 6.13 shows the I/O control block for the MAX7000 device.

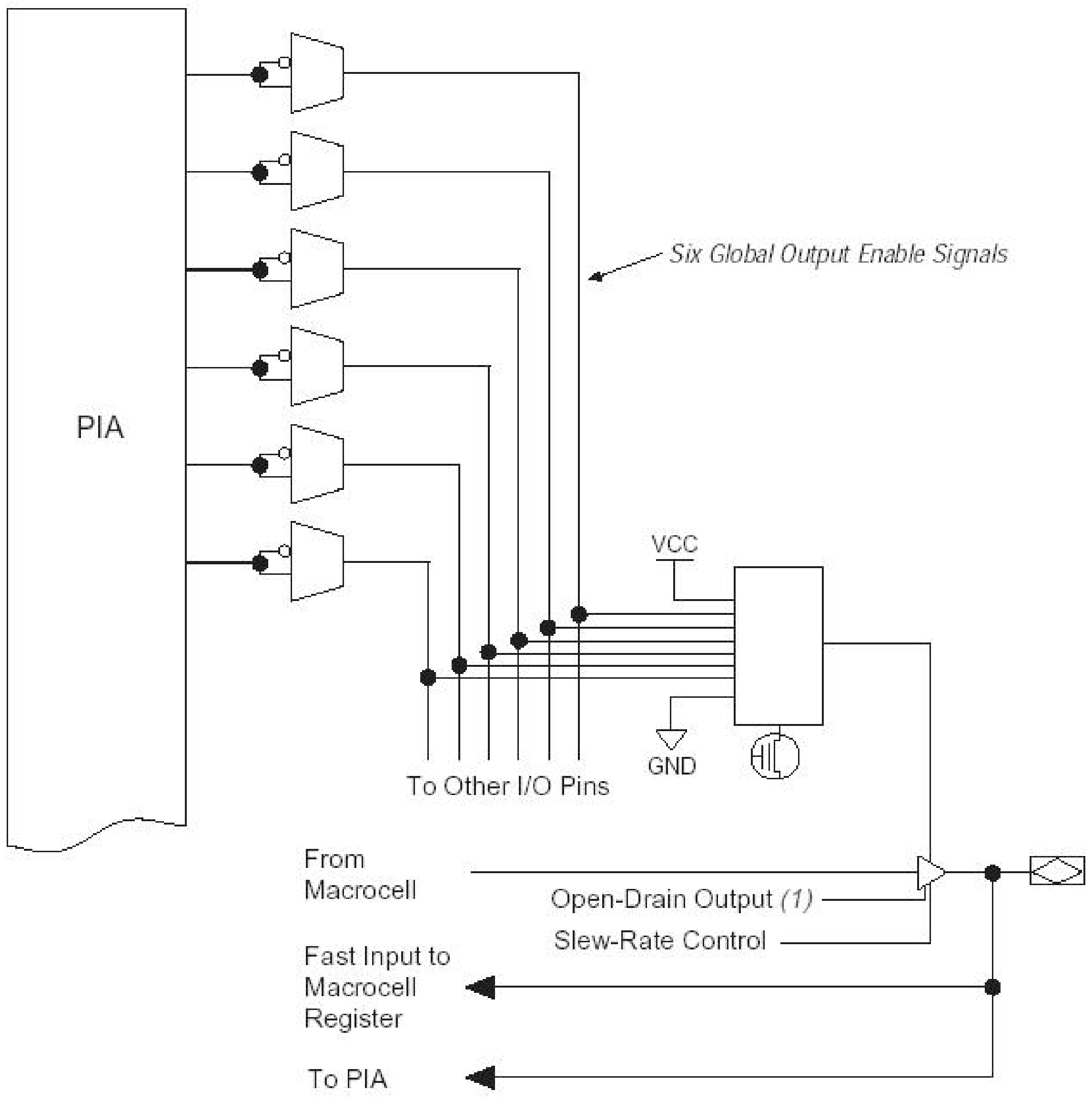


Figure 6.13 MAX7000 I/O control block (MAX7000 Programmable Logic Device Family, p.8).

6.3.2 IN-SYSTEM PROGRAMMABILITY (ISP)

MAX7000S devices are in-system programmable via an industry standard 4-pin JTAG interface (IEEE Std.1149.1-1990) (MAX7000 Programmable Logic Device Family, p.16). ISP allows quick, efficient iterations during design development and debugging cycles. The MAX7000S architecture internally generates the high programming voltage required to program EEPROM cells, allowing in-system programming with only a single 5V power supply. During in-system programming, the I/O pins are tri-stated and pulled up to eliminate board conflicts. The pull-up value is nominally 50KΩ.

ISP simplifies the manufacturing flow by allowing devices to be mounted on a printed circuit board with standard in-circuit test equipment before they are programmed. Programming the devices after they are placed on the board eliminates lead damage on high pin count packages due to device handling and allows a device to be reprogrammed after a system has already shipped to the field (Bostock, 1996, p.80).

6.4 HARDWARE SELECTION CRITERIA

The EPM7128S device from Altera's MAX7000S family was chosen for the project and the reasons for selecting this particular device can be summarised as follows:

6.4.1 REPROGRAMMABILITY

This was the most important criterion for selecting a suitable PLD for the project. As mentioned in section 6.3.2, Altera's MAX7000S family of CPLDs has the ability to be reprogrammed while still being in-circuit through an industry standard JTAG interface. Most of the CPLDs these days have in-system programmable capabilities, but not all of them use the JTAG standard to actually program the devices. This JTAG interface is the feature that makes this project's hardware interface so flexible. It means that any PLD that supports the JTAG interface can be added to the board and can be reprogrammed in-circuit, without changing any embedded software at all.

6.4.2 DEVELOPMENT SOFTWARE AND HARDWARE

Altera's logic development software, MAX+PLUS II, is available from www.altera.com for free. This free package only supports the MAX7000S range of PLDs and only allows graphic entry and VHDL for developing logic circuits. As mentioned in the previous chapter, free development software is a large deciding factor for choosing appropriate development tools for designers.

Probably the biggest deciding factor for choosing one of Altera's devices was the fact that their support was excellent and they donated a development board to PE Technikon which made the development for this project so much easier. The development board allows the designer to test LEDs and switches with an EPM7128S device and also with a FLEX10K device.

6.4.3 FEATURES

The features of the EPM7128S also played a role in the decision of a PLD for the project and here are some of them:

- 2500 usable gates
- 5ns pin-to-pin logic delays with up to 175.4 Mhz counter frequencies
- Open drain output option
- Programmable power-saving mode for a reduction of over 50% in each macrocell
- 100 user I/O pins available
- Programmable security bit for protection of designs

6.5 RECONFIGURABLE HARDWARE INTERFACE

Figure 6.14 indicates the schematic layout of the hardware interface. The two connectors, J10 and J11, provide the interface to the field for connecting peripherals to the board for interfacing with the PLD, the I/O controller, or both. The jumpers for selecting the analog I/O pins and the port D setup for the I/O controller, are also indicated.

Connector J9 is also included to add another PLD device to the JTAG chain which can also be reprogrammed. Another microcontroller which supports JTAG may also be added through this connector to make the system even more flexible.

Each of the pins from the EPM7128S device and the I/O controller are numbered as I/O pins on the J10 and J11 connectors (e.g. IO23 on J10). Table 6.1 gives a summary of these I/O pins and their connection to the I/O controller and the EPM7128S.

Figure 6.14   Reconfigurable hardware interface.

CONN	I/O	PIC		EPM7128S
J10		PORT	PIN	PIN
1	0	RB0	33	4
2	1	RB1	34	5
3	2	RB2	35	6
4	3			8
5	4	RB4	37	9
6	5	RB5	38	10
7	6			11
8	7			12
9	8			15
10	9	RA0/AN0	2	16
11	10	RA1/AN1	3	17
12	11	RA2/AN2	4	18
13	12	RA3/AN3	5	20
14	13	RA4	6	21
15	14	RA5/AN4	7	22
16	15			24
17	16			25
18	17			27
19	18			28
20	19			29
21	20			30
22	21			31
23	22	RC0	15	33
24	23	RC1	16	34
25	24	RC2	17	35
26	25	RC3	18	36
27	26	RC4	23	37
28	27	RC5	24	39
29	28			40
30	29			41
31	30	RD0	19	44
32	31	RD1	20	45
33	32	RD2	21	46
34	33	RD3	22	48
35	34	RD4	27	49
36	35	RD5	28	50
37		VCC		
38		VCC		
39		GND		
40		GND		

CONN	I/O	PIC		EPM7128S
J11		PORT	PIN	PIN
1	36	RD6	29	51
2	37	RD7	30	52
3	38			54
4	39			55
5	40			56
6	41			57
7	42			58
8	43			60
9	44			61
10	45			63
11	46			64
12	47			65
13	48			67
14	49			68
15	50			69
16	51			70
17	52	RE2/AN7	10	73
18	53	RE1/AN6	9	74
19	54	RE0/AN5	8	75
20	55			76
21	56			77
22	57			79
23	58			80
24	59			81
25	IN0			1
26	IN1			2
27	IN2			83
28	IN3			84
29	ADC0	RA0/AN0	2	use JP10
30	ADC1	RA1/AN1	3	use JP11
31	ADC2	RA2/AN2	4	use JP12
32	ADC3	RA3/AN3	5	use JP13
33	ADC4	RA5/AN4	7	use JP14
34	ADC5	RE0/AN5	8	use JP15
35	ADC6	RE1/AN6	9	use JP16
36	ADC7	RE2/AN7	10	use JP17
37		VCC		
38		VCC		
39		GND		
40		GND		

Table 6.1   I/O connections.

From Table 6.1 it can been seen that some of the EPM7128S' pins are connected to the I/O controller's pins. A block diagram as in Figure 6.15 can represent this configuration.

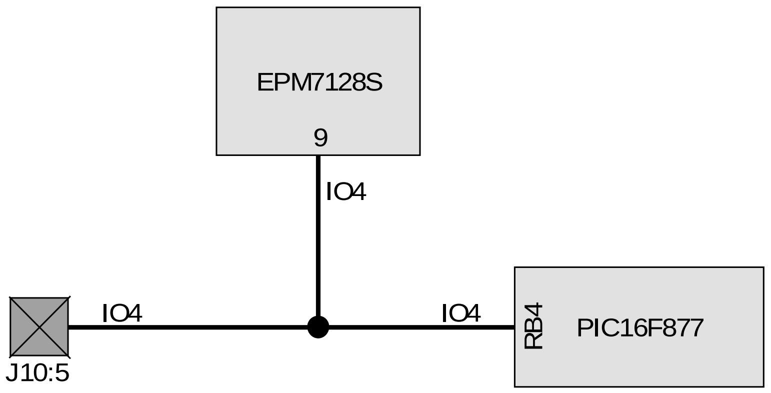


Figure 6.15 I/O controller and EPM7128S pin configuration.

The block diagram indicates that the pins of the two devices are connected directly without any buffers or resistors between them, which might cause problems if not used correctly. Any unused pin of the EPM7128S will be pulled down to ground, which will cause a short if an I/O controller pin on that same net is forced HIGH and will cause damage to either the I/O controller or the EPM7128S. For this reason all unused pins on the EPM7128S which are also connected to the I/O controller are programmed as inputs. This will allow the I/O controller to be able to drive the line either HIGH or LOW without causing any damage. The same applies to unused I/O controller pins; these pins must be programmed as inputs by using the PIC16F877's TRIS registers which will allow the EPM7128S to drive the line HIGH or LOW.

Figure 6.16 shows the graphic representation for assigning the unused pins of the EPM7128S, that are also connected to the I/O controller, as inputs. It will also be used as a template for doing a logic design on the board. The left-hand side is all the input pins and the right-hand side the outputs. The inputs have net names which correspond with the I/O controller pin it is connected to and the output pins have net names that correspond to the I/O connector pin it is connected to, e.g. input RA3 (pin 5 on the PIC16F877) is connected to I/O number 12 (pin 13 on connector J10). Each of the input and output symbols have the word "template@" next to it. This is simply the name of the current project ("template" in this case) followed by the actual pin number of the EPM7128S it is connected to. Using this as a template is easy. The designer can simply move around the inputs and outputs and add other building blocks to circuits. Other inputs and outputs can also be added, and the existing inputs can even be made outputs and vice versa.

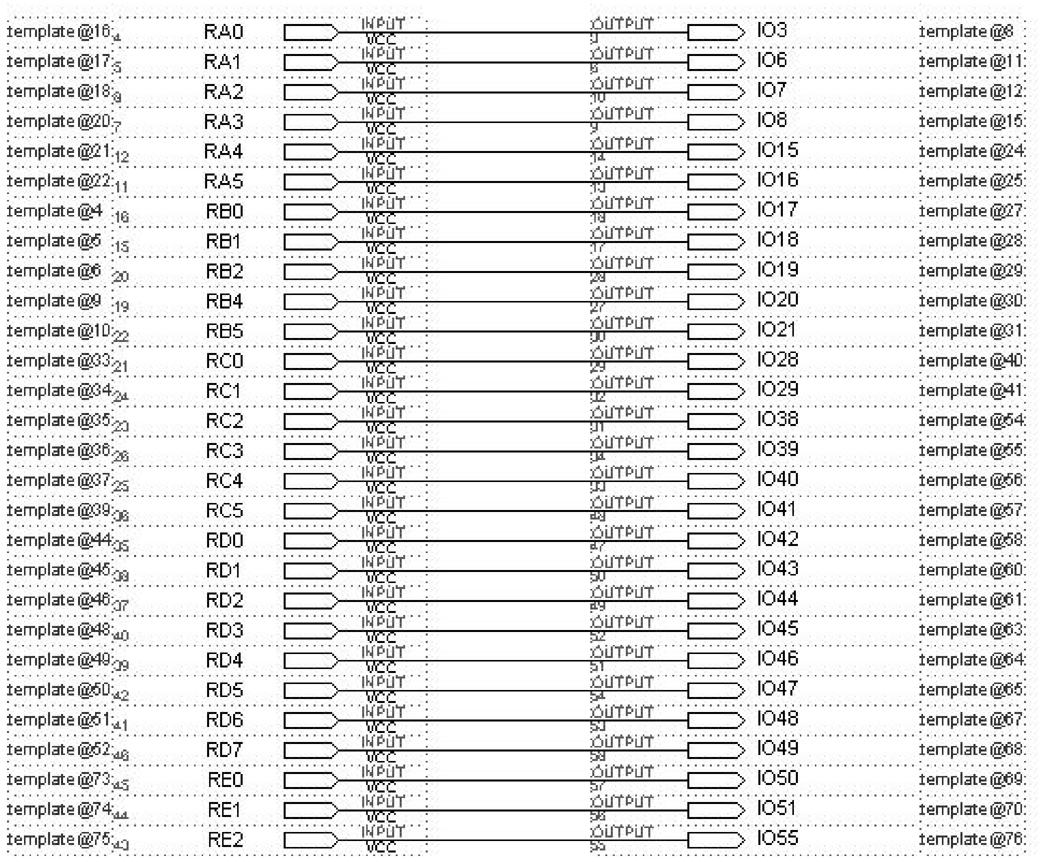

template@16 RA0 INPUT VCC OUTPUT IO3 template@8
template@17 RA1 INPUT VCC OUTPUT IO6 template@11
template@18 RA2 INPUT VCC OUTPUT IO7 template@12
template@20 RA3 INPUT VCC OUTPUT IO8 template@15
template@21 RA4 INPUT VCC OUTPUT IO15 template@24
template@22 RA5 INPUT VCC OUTPUT IO16 template@25
template@4 RB0 INPUT VCC OUTPUT IO17 template@27
template@5 RB1 INPUT VCC OUTPUT IO18 template@28
template@6 RB2 INPUT VCC OUTPUT IO19 template@29
template@9 RB4 INPUT VCC OUTPUT IO20 template@30
template@10 RB5 INPUT VCC OUTPUT IO21 template@31
template@33 RC0 INPUT VCC OUTPUT IO28 template@40
template@34 RC1 INPUT VCC OUTPUT IO29 template@41
template@35 RC2 INPUT VCC OUTPUT IO38 template@54
template@36 RC3 INPUT VCC OUTPUT IO39 template@55
template@37 RC4 INPUT VCC OUTPUT IO40 template@56
template@39 RC5 INPUT VCC OUTPUT IO41 template@57
template@44 RD0 INPUT VCC OUTPUT IO42 template@58
template@45 RD1 INPUT VCC OUTPUT IO43 template@60
template@46 RD2 INPUT VCC OUTPUT IO44 template@61
template@48 RD3 INPUT VCC OUTPUT IO45 template@63
template@49 RD4 INPUT VCC OUTPUT IO46 template@64
template@50 RD5 INPUT VCC OUTPUT IO47 template@65
template@51 RD6 INPUT VCC OUTPUT IO48 template@67
template@52 RD7 INPUT VCC OUTPUT IO49 template@68
template@73 RE0 INPUT VCC OUTPUT IO50 template@69
template@74 RE1 INPUT VCC OUTPUT IO51 template@70
template@75 RE2 INPUT VCC OUTPUT IO55 template@76

Figure 6.16 Graphic entry for assigning unused pins as inputs.

6.6 CONCLUSION

The chapter gave a brief introduction to programmable logic devices to promote the idea of reprogrammable hardware in the project. The MAX7000S device from Altera was introduced as part of the detailed explanation of the actual reconfigurable hardware interface in the project.

The next chapter will explain the most important section of the project in terms of reconfigurability: the actual programming interface. This will also be the last chapter discussing the hardware for the project.

CHAPTER 7
THE EMBEDDED PROGRAMMING INTERFACE

7.1 OVERVIEW

This is the last chapter on the hardware section of the project and will discuss the programming interface in detail. In-circuit programming in PICmicro devices will first be explained after which in-system programming in Altera's MAX7000S devices will be discussed. A detailed description of the hardware for the actual programming interface will follow. The chapter will conclude with a discussion on the necessary embedded software for the programming interface.

7.2 IN-CIRCUIT SERIAL PROGRAMMING (ICSP) WITH THE PIC16F877

The PIC16F877 is used as the I/O controller in the project and needs to be dynamic in the sense that it can be reprogrammed. Microchip's PICmicro devices are programmed using a serial method and will allow them to be reprogrammed while still plugged into the system (In-circuit serial programming for PIC16F8XX FLASH MCUs, p.1). The PIC16F877 may also be programmed using a single +5V supply in low voltage programming mode.

7.2.1 HARDWARE REQUIREMENTS

The application circuit is the most important aspect when determining the requirements for ICSP in PICmicro devices (In-circuit serial programming guide, p.21). Figure 7.1 shows a typical application circuit.

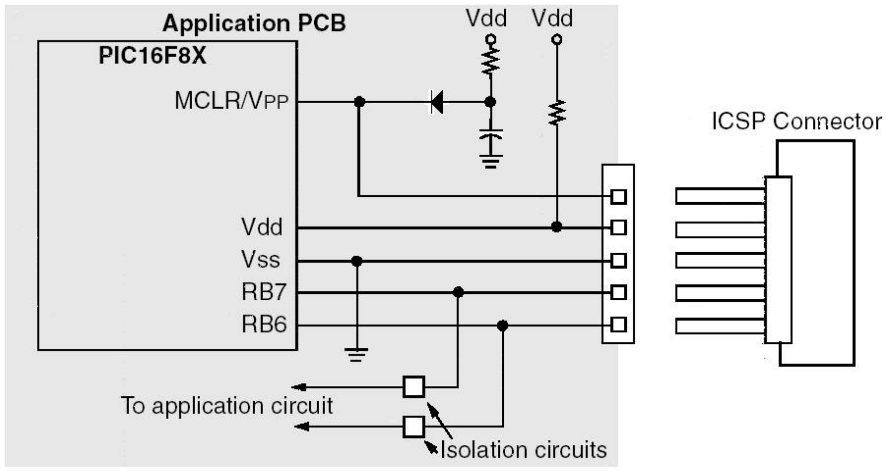


Figure 7.1 Typical application circuit (In-circuit serial programming guide, p.21).

The application must compensate for the following:

- Isolation of the MCLR/Vpp pin from the rest of the circuit
- Isolation of pins RB6 and RB7 from the rest of the circuit
- Capacitance on each of the Vdd, MCLR/Vpp, RB6 and RB7 pins
- Minimum and maximum operating voltage for Vdd
- PICmicro oscillator
- Interface to the programmer

The MCLR/Vpp pin is normally connected to an RC circuit (In-circuit serial programming guide, p.21). In normal programming mode this pin is driven to +13V to allow programming of the device, so the application circuit should be isolated from this voltage. Isolation is provided by the diode.

Pins RB6 and RB7 are used by the PIC16F877 for serial programming. RB6 is the clock line, which is driven by the programming device. RB7 is a bi-directional data line and is driven by die programming device when programmed and by the PIC16F877 when data is read or verified. These pins are not dedicated for programming and can be used for I/O functions when the PICmicro runs normally. To prevent signal interference during programming, however, these pins must be isolated from the application circuit. This is normally done by introducing a series resistor between the application circuit connection and the pin (In-circuit serial programming for PIC16F8XX FLASH MCUs, p.165).

The total capacitance on the programming pins affects the rise rates of these signals as they are driven out of the programming device (In-circuit serial programming guide, p.22). The programming device must be able to drive these lines strong enough to overcome the capacitance. Buffers are normally added between the programming device and the PICmicro if the programmer's driving capability is too weak.

Microchip states that the PIC16F877 should be programmed at a Vdd level or +5V, which is the general supply voltage for embedded circuits. Sometimes, however, PICmicro devices are used in circuits which are powered from a series of 1.5V cells. These cells will not supply the circuit with exactly +5V, but with +4.5V instead. To ensure optimal programming, the PICmicro device must programmed at +5V and verified at +4.5V.

The programming device must drive the MCLR/Vpp pin to the programming voltage before the oscillator toggles four or more times. Failure to do this causes the PICmicro to start up and increment program counter to some value X. As soon as the device then enters programming mode, the programming will start at an offset of X. This scenario is not applicable when using a crystal oscillator for the PICmicro, since it makes use of an oscillator start-up circuit. With RC oscillators, however, the oscillator pins must first be driven LOW before rising the MCLR/Vpp pins to prevent the device from starting up (In-circuit serial programming guide, p.22).

Both the Vdd and programming supply voltages must have a minimum resolution of 0.25V. Table 5.1 in Chapter 5 describes the programming pins for the PIC16F877.

7.2.2 PROGRAM MODE ENTRY

The user memory space extends from 0x0000 to 0x1FFF (8K). In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000 – 0x1FFF) being user program memory and the second half (0x2000 – 0x3FFF) being configuration memory. The PIC16F877's program counter (PC) will increment from 0x0000 to 0x1FFF and wrap to 0x0000; and increment from 0x2000 to 0x3FFF and wrap around to 0x2000.

In the configuration memory space, only 0x2000 – 0x200F are physically implemented, of which only 0x2000 to 0x2007 are available. The first four memory locations (0x2000 – 0x2003) in the configuration memory are the ID locations which can be used to store any ID information for the device. It is recommended that only the lower four bits of the word must be used for storing ID information (In-circuit serial programming for PIC16F8XX FLASH MCUs, p.166). The configuration word is at location 0x2007.

Figure 7.2 gives a summary of the program memory map.

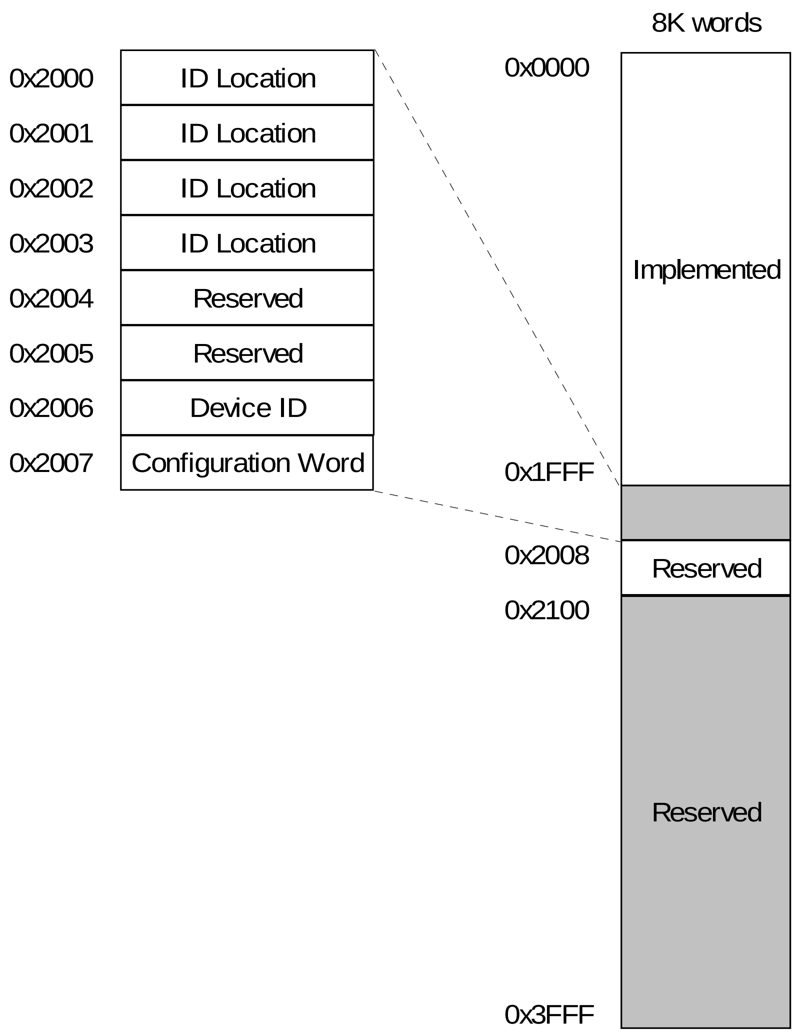


Figure 7.2 Program memory map for the PIC16F877 (Adapted from In-circuit serial programming for PIC16F8XX FLASH MCUs, p.167).

7.2.2.1 Low-voltage ICSP mode

When the low-voltage programming (LVP) bit in the configuration word is set to 1, the low-voltage ICSP entry is enabled. Pin RB3 will be dedicated to programming in the low-voltage ICSP mode. This programming mode is entered by following the next steps:

- Hold pins RB6 and RB7 LOW
- Bring the MCLR/Vpp pin to +5V
- Bring pin RB3 to +5V

This will cause the device to go into programming mode and places all other logic into the reset state (high impedance inputs).

7.2.2.2 Serial program/verify operation

The PIC16F877 makes use of certain commands for the programming/verification of the device. All commands are transmitted least significant bit (LSB) on the rising edge of the clock. The commands are:

- LOAD CONFIGURATION
  After this command the PC will be set to 0x2000. The only way to get back to the user program memory is to reset the device by driving MCLR/Vpp LOW.

- LOAD DATA FOR PROGRAM MEMORY
  After receiving this command, the chip will load a 14-bit data word.

- LOAD DATA FOR DATA MEMORY
  After receiving this command, the chip will load a 14-bit data word. The data memory is, however, only 8-bits wide and thus only the first 8-bits of data after the start bit will be programmed into the data memory.

- READ DATA FROM PROGRAM MEMORY
  After receiving this command, the PICmicro device will transmit data bits out of the program memory currently accessed.

- READ DATA FROM DATA MEMORY
  After receiving this command, the PICmicro device will transmit data bits out of the data memory currently accessed.

- INCREMENT ADDRESS
  The PC will increment when this command is received.

- BEGIN PROGRAMMING
  Programming of the appropriate memory will begin after this command is received and decoded. A load command must be given before every 'begin programming' command.

- BULK ERASE PROGRAM MEMORY

  After this command is performed, the next command will erase the entire program memory.

- BULK ERASE DATA MEMORY

  After this command is performed, the next command will erase the entire data memory.

7.3 IN-SYSTEM PROGRAMMABILITY WITH THE MAX7000S

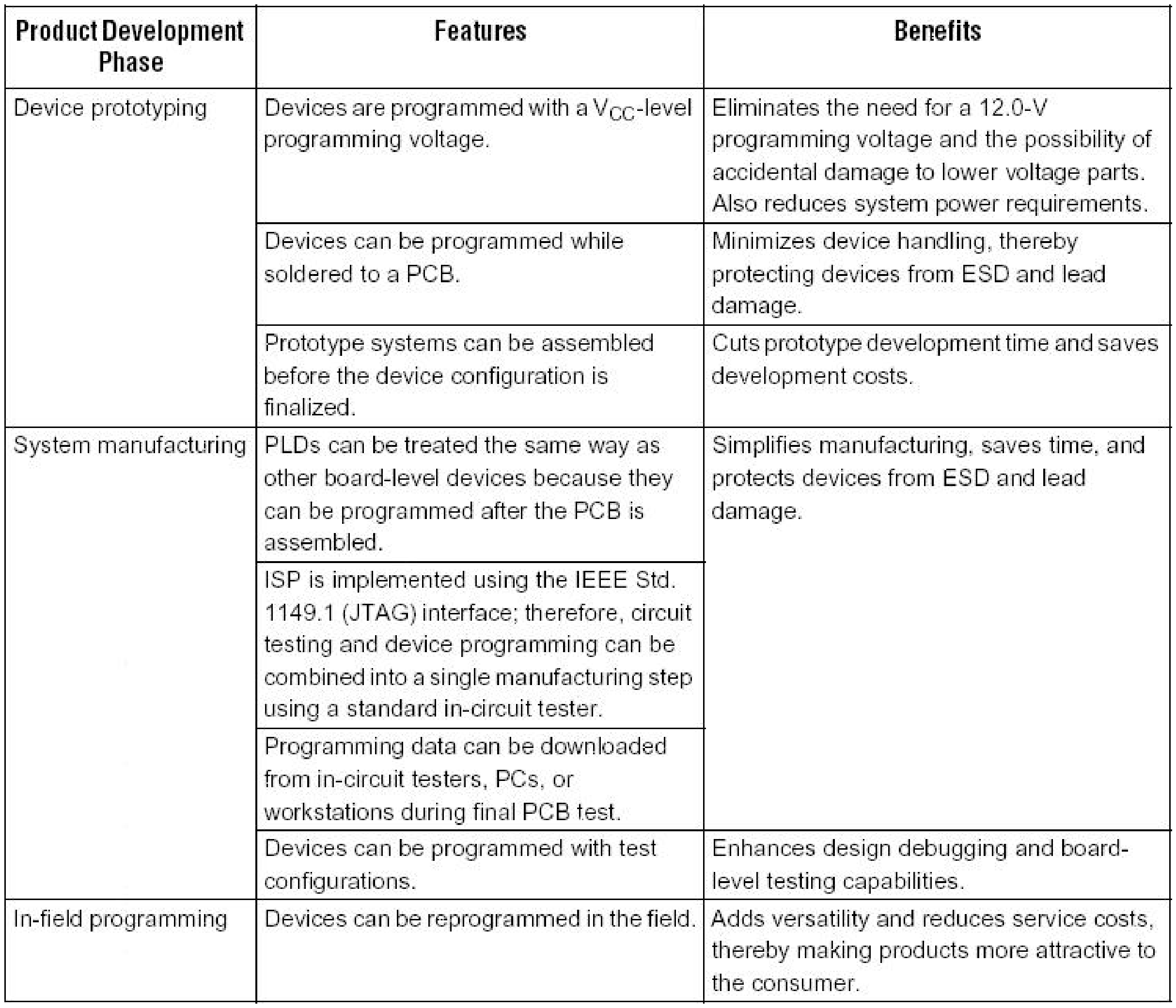
MAX7000S devices are PLDs, based on the Altera Multiple Array Matrix (MAX) architecture that supports the IEEE Std. 1149.1 JTAG interface. MAX devices are also in-system programmable, which adds programming flexibility and provides benefits in many phases of product development, manufacturing and field use. Seeing that the EPM7128S provides the reconfigurable hardware interface to the project, the programming interface also needs to be able to program the MAX device using an embedded processor.

7.3.1 FEATURES AND BENEFITS

ISP reduces cost, shortens development time and provides a wider range of programming options than standard device programming methods (Introduction to ISP, p.1). With ISP, the developer can:

- Program and reprogram devices after they are soldered onto the PCB, minimizing the possibility of lead damage or electrostatic discharge (ESD) exposure.
- Manufacture systems before finalizing device configuration.
- Perform boundary-scan test (BST) procedures and program devices using in-circuit testers.
- Upgrade systems in the field after they have been shipped.

Table 7.1 summarises the features and benefits of using ISP-capable devices.



Product Development Phase	Features	Benefits
Device prototyping	Devices are programmed with a V_{CC}-level programming voltage.	Eliminates the need for a 12.0-V programming voltage and the possibility of accidental damage to lower voltage parts. Also reduces system power requirements.
	Devices can be programmed while soldered to a PCB.	Minimizes device handling, thereby protecting devices from ESD and lead damage.
	Prototype systems can be assembled before the device configuration is finalized.	Cuts prototype development time and saves development costs.
System manufacturing	PLDs can be treated the same way as other board-level devices because they can be programmed after the PCB is assembled.	Simplifies manufacturing, saves time, and protects devices from ESD and lead damage.
	ISP is implemented using the IEEE Std. 1149.1 (JTAG) interface; therefore, circuit testing and device programming can be combined into a single manufacturing step using a standard in-circuit tester.	
	Programming data can be downloaded from in-circuit testers, PCs, or workstations during final PCB test.	
	Devices can be programmed with test configurations.	Enhances design debugging and board-level testing capabilities.
In-field programming	Devices can be reprogrammed in the field.	Adds versatility and reduces service costs, thereby making products more attractive to the consumer.

Table 7.1 Features and benefits of ISP-capable MAX devices (Introduction to ISP, p.3).

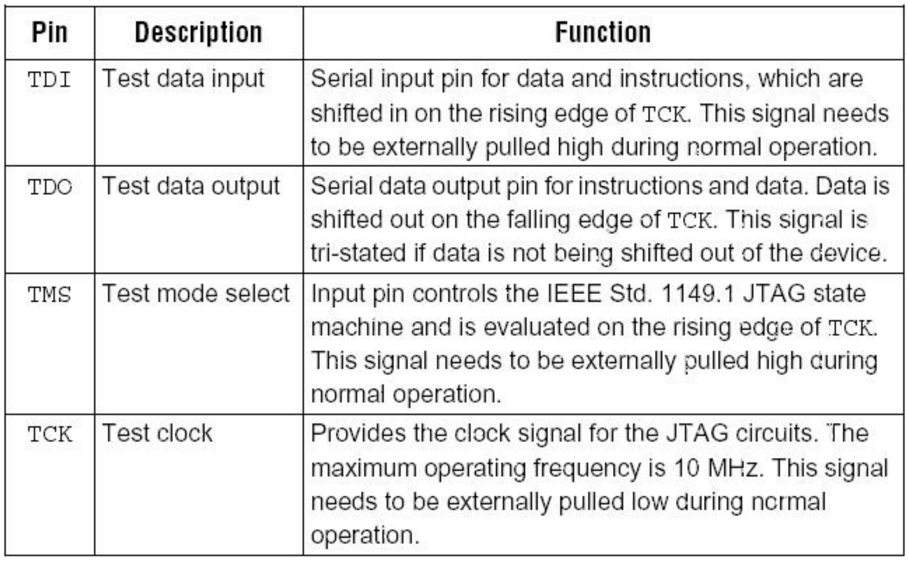
7.3.2 VCC-LEVEL PROGRAMMING

MAX7000S devices support ISP through a Vcc-level programming voltage (In-system programmability guidelines, p.4). The devices generate a +12V programming voltage internally to program, verify and erase the device's EEPROM cells, eliminating the need for the external +12V programming voltage typically required for programming.

MAX7000S devices are guaranteed for 100 erase and programming cycles with 100% programming and functional yields (Introduction to ISP, p.5).

7.3.3 JTAG INTERFACE

The IEEE Std. 1149.1 JTAG interface makes use of four lines to program a MAX device. These lines are described in Table 7.2. Not only MAX devices can be programmed using this interface, but also any other device that supports JTAG. This is what makes this interface so powerful in this project, because now any other device (even microcontrollers) can be added to the board and also be programmed using the same programming interface.



Pin	Description	Function
TDI	Test data input	Serial input pin for data and instructions, which are shifted in on the rising edge of TCK. This signal needs to be externally pulled high during normal operation.
TDO	Test data output	Serial data output pin for instructions and data. Data is shifted out on the falling edge of TCK. This signal is tri-stated if data is not being shifted out of the device.
TMS	Test mode select	Input pin controls the IEEE Std. 1149.1 JTAG state machine and is evaluated on the rising edge of TCK. This signal needs to be externally pulled high during normal operation.
TCK	Test clock	Provides the clock signal for the JTAG circuits. The maximum operating frequency is 10 MHz. This signal needs to be externally pulled low during normal operation.

Table 7.2 JTAG pins (Introduction to ISP, p.8).

During erasure, programming, and verification, all device I/O pins are tri-stated to eliminate interference from other devices on the PCB. Devices are programmed by applying the appropriate signals on the TMS and TCK inputs and shifting data into and out of the devices on the TDI and TDO pins respectively. After programming, the IEEE Std. 1149.1 JTAG Test Access Port (TAP) controller state machine must be advanced to the RESET state, which is maintained by external pull-up resistors on the TCK, TMS and TDI pins. During normal operation, the pull-up resistors prevent the device from entering other modes. Figure 7.3 in the next section shows a typical JTAG connection to a device.

7.3.4 PROGRAMMING THE MAX7000S DEVICE

The JTAG interface can be used to program a single device or a chain of devices, depending on the layout of the PCB.

7.3.4.1 Single-device programming

This is the simplest form for programming a JTAG device and is indicated in Figure 7.3.

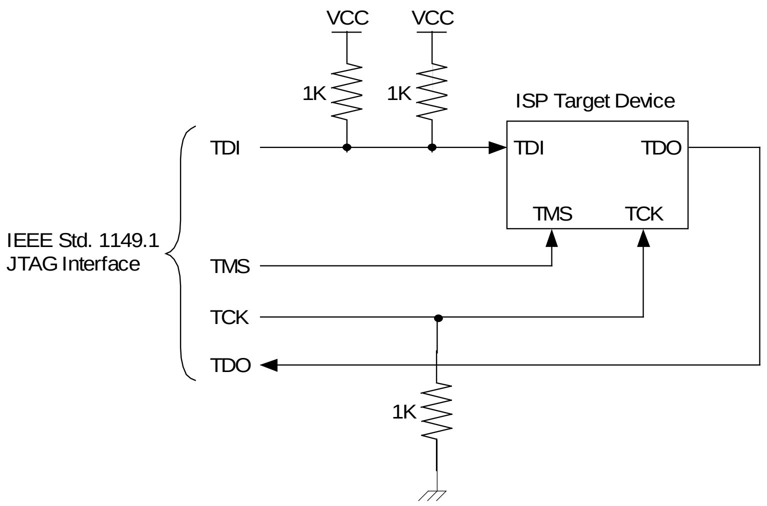


Figure 7.3 Single device programming (Adapted from In-system programmability in MAX devices, p.9).

7.3.4.2 JTAG-chain device programming

When programming a chain of devices, the JTAG interface is connected to several devices. The number of devices in the JTAG chain is limited only by the drive capability of the JTAG interface. Buffers are often implemented to improve this. Figure 7.4 shows the typical JTAG connection for chain device programming.

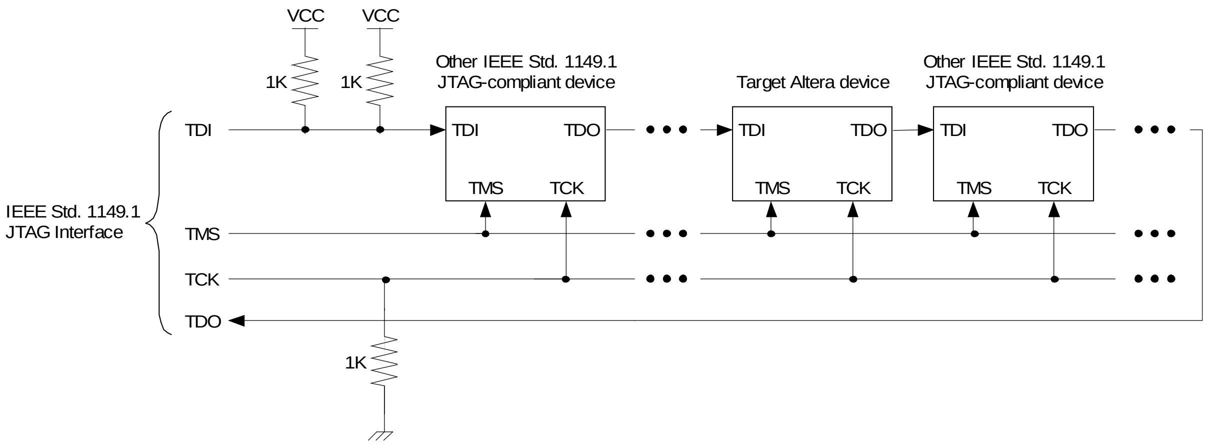


Figure 7.4 JTAG-chain device programming (Adapted from In-system programmability in MAX devices, p.10).

Chain device programming can be further divided into:

- Sequential programming

  It is the process of programming multiple devices in a chain one device at a time (In-system programmability guidelines, p.7). After the first device in the chain is finished being programmed, the next device is programmed. The sequence continues until specified devices in the JTAG chain are programmed. After a device is programmed, it uses a JTAG BYPASS instruction to pass data to subsequent devices in the chain.

- Concurrent programming

  Concurrent programming is used to program devices from the same family in parallel. The result is a considerably faster programming time than sequential programming (In-system programmability guidelines, p.8). The only drawback of this scheme is that the devices must all be of the same family (i.e. microcontrollers and CPLDs cannot be in the same chain).

7.4 THE PROGRAMMING INTERFACE

The whole programming process for the I/O controller and the reconfigurable hardware is controlled by an 80C552 microcontroller, a derivative of the very popular 80C51. This microcontroller has various enhancements from its predecessor, but will not be elaborated on seeing that it wasn't a factor when a suitable programming processor was chosen. The main requirements for a processor was:

- Must have at least 64K of program memory (could be internal or accessed externally)
- Must have at least 21 free bi-directional I/O pins (for status, commands, ICSP programming for the PIC16F877, the JTAG interface and I^2C communication).
- Must be an 8051 derivative, seeing that the Jam player for programming the MAX7000S device is written for an 8051 core microcontroller.

The main reason for settling for the 80C552 is simply for its local availability and cost. There are, however, other 8051 derivatives available that can run at even higher speeds than the 80C552.

A simplified block diagram indicating the programming interface with the 80C552 is shown in Figure 7.5.

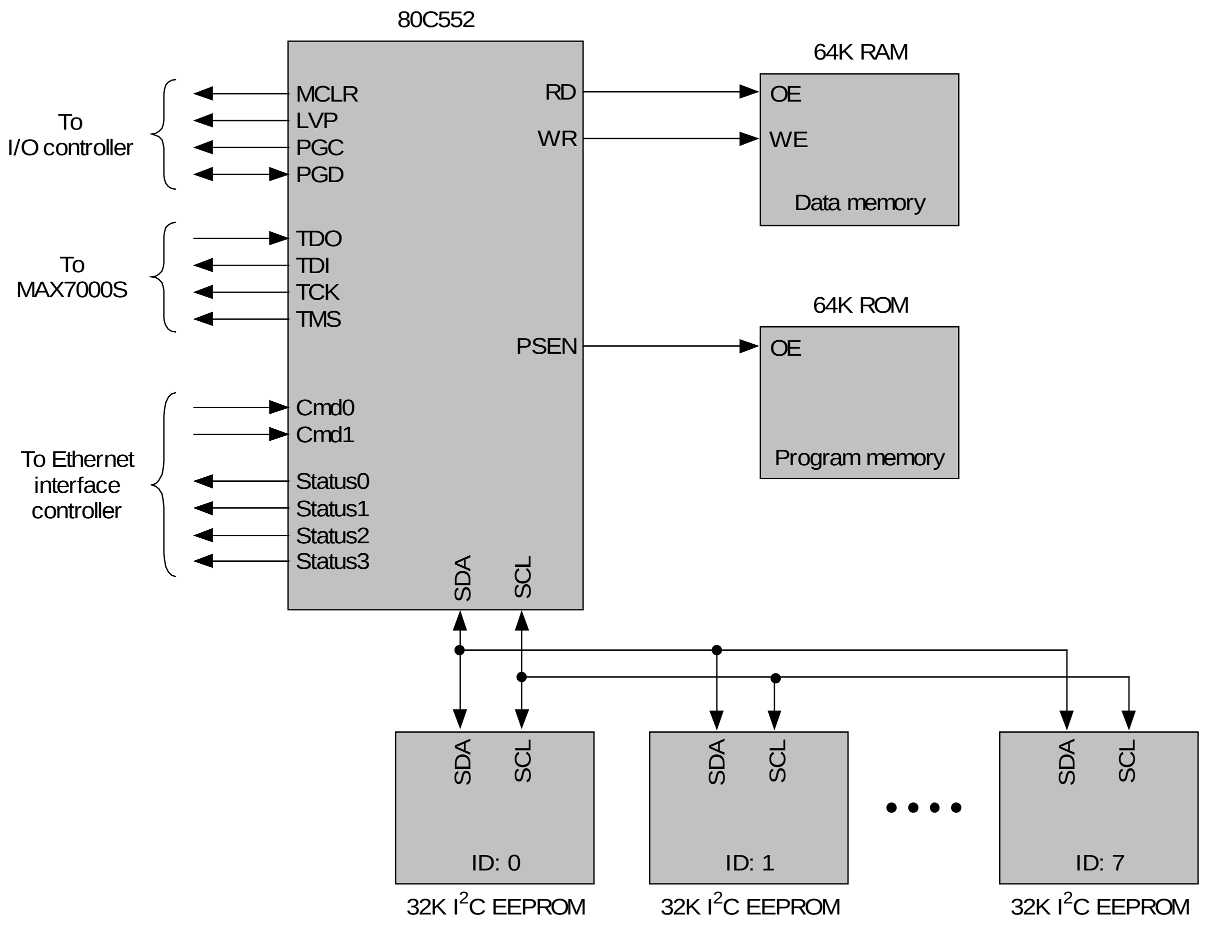


Figure 7.5 Simplified block diagram of programming interface.

The 80C552 retrieves and executes instructions from ROM or program memory. Part of executing instructions involves controlling I/O pins that provide access to ROM, RAM, I/O ports and addresses. When the 80C552 retrieves an instruction to access external data, or RAM, the processor automatically toggles the RD pin such that the information in the RAM is retrieved and stored in the appropriate internal registers. These actions are performed automatically by the processor. The ROM contains the embedded software for programming the I/O controller and the MAX7000S device and makes use of the external RAM, or data memory, to store temporary data for calculations.

The interface also makes use of eight EEPROM memory devices, which are all connected to the 80C552 via a common I^{2}C bus. The I^{2}C bus is implemented by using two bi-directional lines (SDA, SCL) on the 80C552. The source files for the I/O controller and the MAX7000S are transferred via the Internet and each gets stored in one of these devices. At the moment only two of the EEPROMs are used; one for the source file of the I/O controller and one for the source file of the MAX7000S. The rest of the EEPROMs are used purely for future developments.

The 80C552 also have four status lines (Status0 – Status3), which are directly connected to four pins of the Ethernet interface controller. It is used to indicate to the Ethernet interface controller how far the source code is loaded into memory and how far the programming process is. The status lines will also indicate if any problems occurred during loading or programming the device. Table 7.3 summarises the truth table for the status lines.

STATUS3	STATUS2	STATUS1	STATUS0	Description	Value
0	0	0	0	Nothing	0
0	0	0	1	Loading 4%	1
0	0	1	0	Loading 20%	2
0	0	1	1	Loading 36%	3
0	1	0	0	Loading 52%	4
0	1	0	1	Loading 68%	5
0	1	1	0	Loading 84%	6
0	1	1	1	Loading 100%	7
1	0	0	0	Programming 4%	8
1	0	0	1	Programming 20%	9
1	0	1	0	Programming 36%	10
1	0	1	1	Programming 52%	11
1	1	0	0	Programming 68%	12
1	1	0	1	Programming 84%	13
1	1	1	0	Programming 100%	14
1	1	1	1	Error Loading/Programming	15

Table 7.3 Truth table for status lines.

The two command lines (Cmd0 and Cmd1) are also directly connected to two pins of the Ethernet interface controller. The Ethernet interface uses these lines to instruct the programming processor (80C552) to start with the programming process of the I/O controller or the MAX7000S. Table 7.4 summarises the function of the two lines.

CMD1	CMD0	Description	Value
0	0	Nothing	0
0	1	Program RECONFIGURABLE HARDWARE	1
1	0	Program I/O CONTROLLER CPU	2

Table 7.4 Truth table for command lines.

Both the interfaces for programming the I/O controller and the MAX7000S use four lines each. The MAX7000S programming interface makes use of one input line (TDO) and three output lines (TDI, TMD, TCK). The I/O controller interface makes use of three output lines (PGC, LVP, MLCR) and one bi-directional line (PGD). Both these sets of programming lines first go through buffers to provide them with enough driving current to program the devices. This automatically causes a complication in the bi-directional lines since the buffers operate only in one direction. The bi-directional lines for the I^{2}C bus are also first taken through the buffer. To support the bi-directional lines, a new scheme must be used as shown in Figure 7.6.

Instead of having only one bi-directional pin on the 80C552, the line is split into a _IN and an _OUT pin, e.g. the PGD pin is split into a PGD_IN and a PGD_OUT pin. The original bi-directional line still operates as usual, but it has two pins connected to it; a pin that will drive the line and a pin that will be driven by the line itself.

The JTAG FROM CPU connector (J7) is connected to the MAX7000S programming interface from the 80C552 after the buffers. The JTAG IN connector (J8) is connected straight to the programming pins of the MAX7000S device. Both these connectors were added purely for development reasons and there could have been just a straight connection between the two connectors for the interface to operate correctly. At the moment a ribbon cable is used to provide this straight connection.

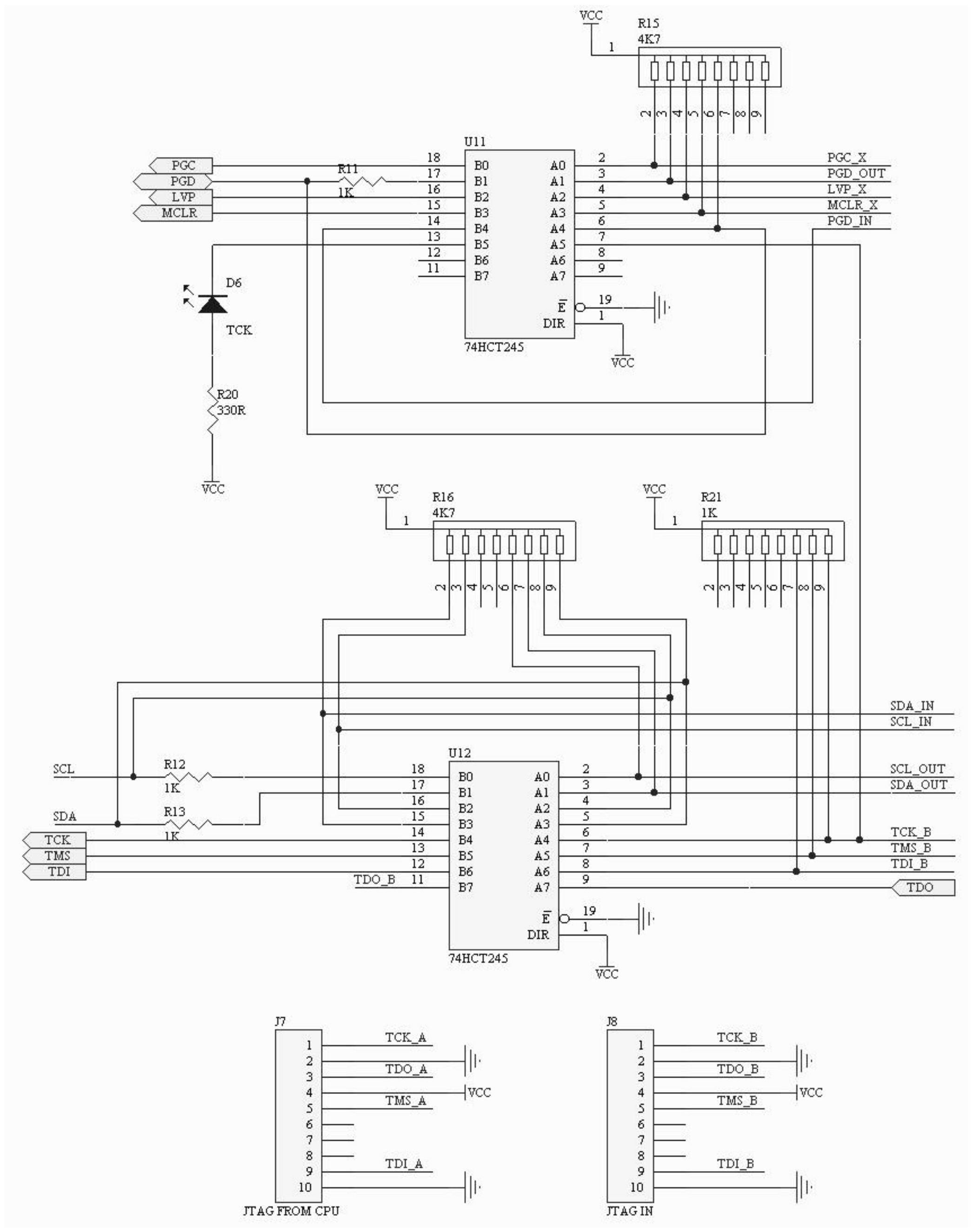


Figure 7.6 Bi-directional ports using buffers.

7.5 SOURCE FILE STORAGE

As mentioned above, the source files for the I/O controller and the MAX7000S are stored in the I^{2}C EEPROMs. To program the specific device, the 80C552 first fetches all the information from the EEPROM and stores it in a large buffer (in the external RAM) from where it then programs the device. This section will explain the storage structure for the two devices' source files.

7.5.1 I/O CONTROLLER STORAGE

The storage of the source file for the I/O controller is complicated by the fact that the PIC16F877 uses 14-bit instruction words. The I^{2}C EEPROM memory only has a data width of 8-bits, which means that each instruction word will have to be divided into a HI byte and a LO byte. The four main programming areas in the PIC16F877 is the Program memory, the ID locations, the Configuration word and the Data memory. Not all four these areas will always need to be programmed and most of the time only the Program memory will be utilised. To cater for this, a scheme for determining the amount of bytes used by a particular programming area has been devised and will also help to separate these areas in the EEPROM itself. Figure 7.7 shows a typical structure for a PIC16F8777 source file stored in the I^{2}C EEPROM.

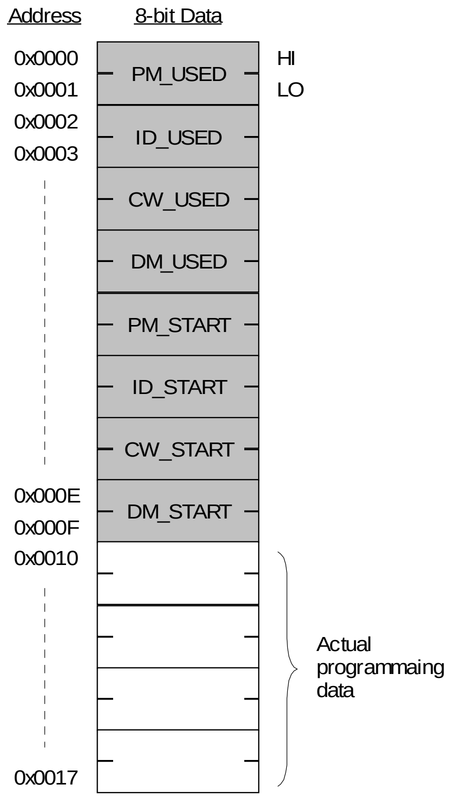
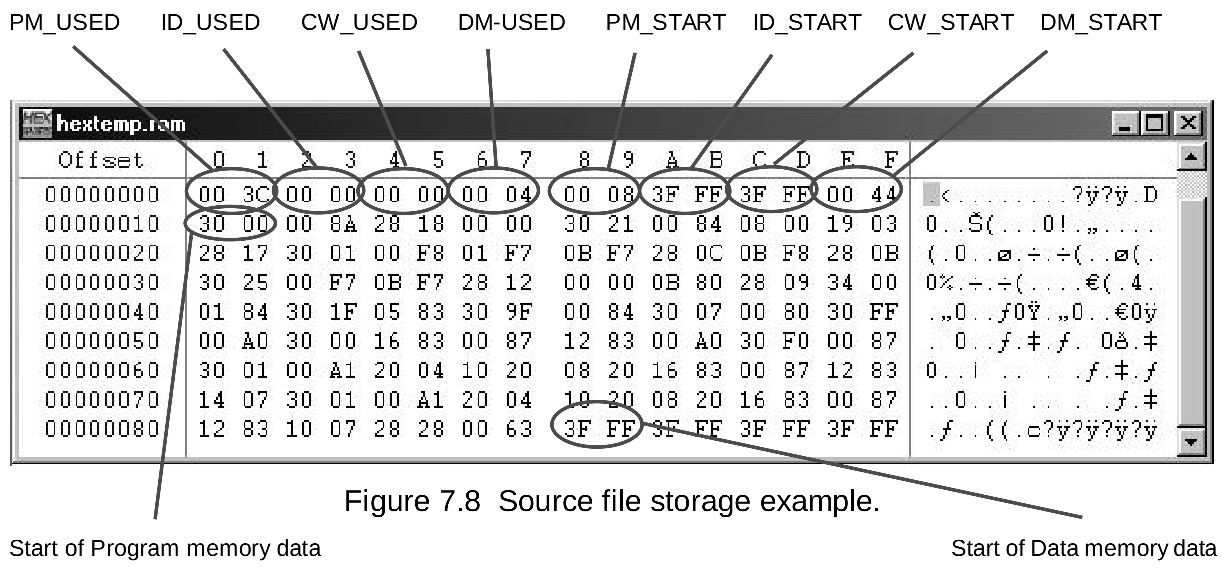


Figure 7.7 Structure of a PIC16F877 source file stored in EEPROM.

From the figure it can be seen that in each case the HI byte is stored first followed by the LO byte. The PM_USED parameter indicates how many words have been used for the Program memory area. The same applies for the ID_USED (ID locations), CW_USED (Configuration word) and DM_USED (Data memory) parameters. PM_START simply acts as a “pointer” to indicate at exactly what EEPROM address the data for the Program memory starts. Once again, the same applies to ID_START, CW_START and DM_START parameters. The addresses stored in the parameters must be doubled to find the exact EEPROM address, seeing that the PIC16F877 works with 14-bit words.

Figure 7.8 is an example of a source file after it has been stored in EEPROM. A normal hex editor was used to read the contents of the EEPROM and display it.



```
 Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000   00 3C 00 00 00 00 00 04  00 08 3F FF 3F FF 00 44   .<........?ÿ?ÿ.D
00000010   30 00 00 8A 28 18 00 00  30 21 00 84 08 00 19 03   0..Š(...0!.„....
00000020   28 17 30 01 00 F8 01 F7  0B F7 28 0C 0B F8 28 0B   (.0..ø.÷.÷(..ø(.
00000030   30 25 00 F7 0B F7 28 12  00 00 0B 80 28 09 34 00   0%.÷.÷(....€(.4.
00000040   01 84 30 1F 05 83 30 9F  00 84 30 07 00 80 30 FF   .„0..ƒ0Ÿ.„0..€0ÿ
00000050   00 A0 30 00 16 83 00 87  12 83 00 A0 30 F0 00 87   . 0..ƒ.‡.ƒ. 0ð.‡
00000060   30 01 00 A1 20 04 10 20  08 20 16 83 00 87 12 83   0..¡ .. . .ƒ.‡.ƒ
00000070   14 07 30 01 00 A1 20 04  10 20 08 20 16 83 00 87   ..0..¡ .. . .ƒ.‡
00000080   12 83 10 07 28 28 00 63  3F FF 3F FF 3F FF 3F FF   .ƒ..((.c?ÿ?ÿ?ÿ?ÿ
```

Figure 7.8 Source file storage example.

The example shows that 60 words are used by the Program memory and 4 words by the Data memory. There are no words used for the ID locations and Configuration word. The PM_START parameter reads the value 0x0008, but must be doubled to indicate the start address of the Program memory data (0x0010). The same applies to the DM_START parameter to indicate the start of the Data memory (0x0088). Both the ID_START and CW_START parameters read as 0x3FFF. This is simply a dummy start position, seeing that both these areas are not utilised in this example.

7.5.2 MAX7000S STORAGE

The storage structure for the MAX7000S is much less complicated since the Jam byte-code player (explained in the next section) uses 8-bit words for programming the device. The only extra information required is the number of bytes that needs to be programmed into the MAX device. Figure 7.9 shows a typical structure for a MAX7000S source file stored in the I^2C EEPROM.

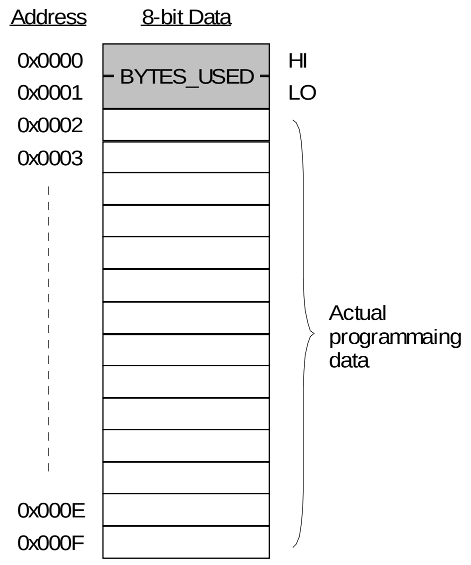


Figure 7.9 Structure of a MAX7000S source file stored in EEPROM.

Seeing that this particular EEPROM can store data in up to 32K different address allocations, it is once again necessary to split the BYTES_USED parameter into a HI byte and a LO byte. The rest of the programming data, however, is used as single bytes and does not need separate HI and LO bytes.

7.6 PROGRAMMING INTERFACE SOFTWARE

The embedded software for the programming interface can be divided into four different parts:

- ICSP functions for programming the PIC16F877
- The Jam byte-code player for programming the MAX7000S
- Code for driving the I^2C bus
- The main program loop which will do all the controlling of the different parts

7.6.1 ICSP FUNCTIONS

As mentioned in section 7.2.3.2, the internal programming interface of PIC16F877 uses certain commands to program itself. Figure 7.10 shows the typical flow chart for programming the program memory and Figure 7.11 shows the flow chart for programming the configuration word and the ID locations.

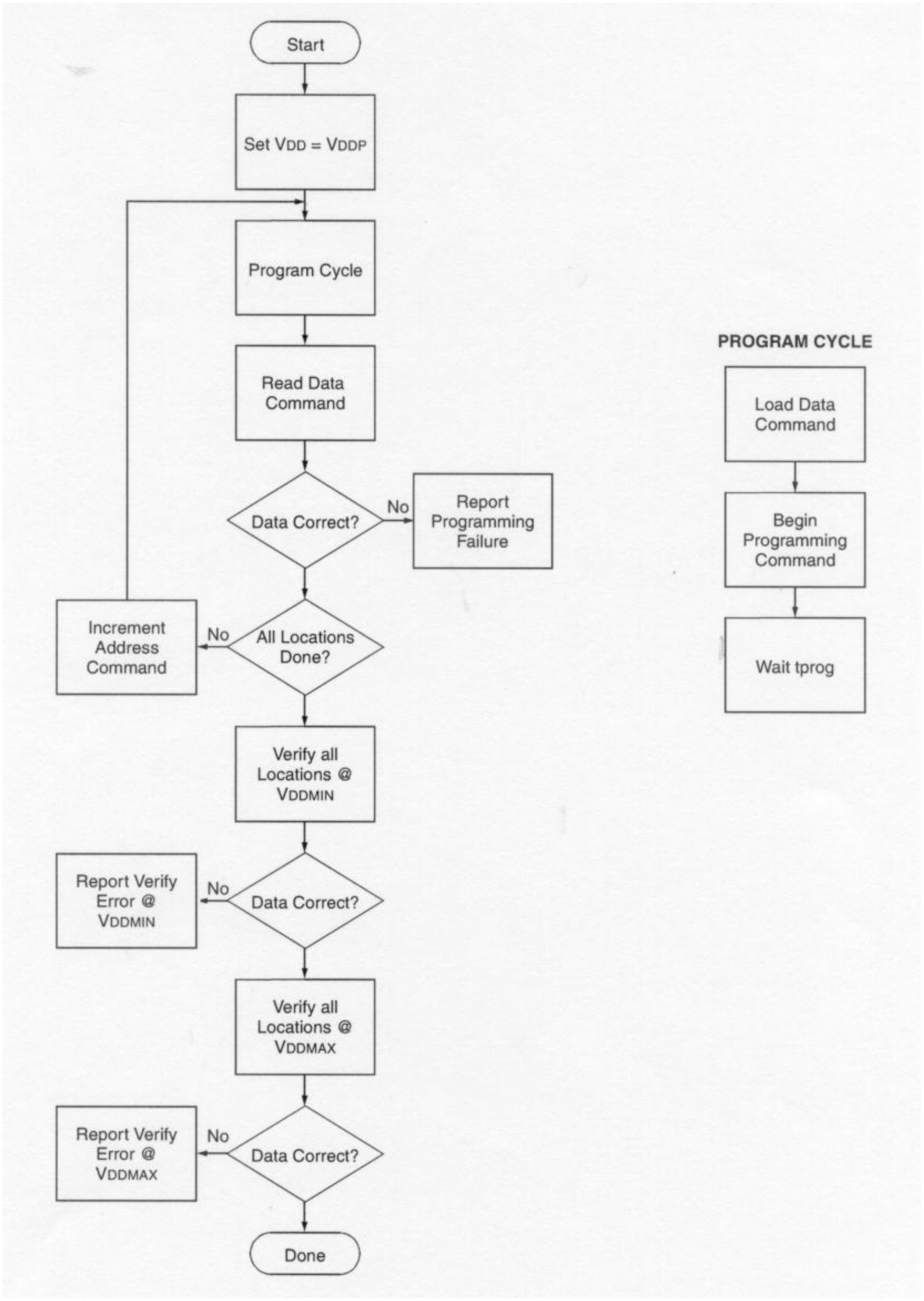

Start
Set VDD = VDDP
Program Cycle
Read Data Command
Data Correct?
No
Report Programming Failure
Increment Address Command
No
All Locations Done?
Verify all Locations @ VDDMIN
Report Verify Error @ VDDMIN
No
Data Correct?
Verify all Locations @ VDDMAX
Report Verify Error @ VDDMAX
No
Data Correct?
Done
PROGRAM CYCLE
Load Data Command
Begin Programming Command
Wait tprog

Figure 7.10 Flow chart for programming the program memory (In-circuit serial programming for PIC16F8XX FLASH MCUs, p.170).

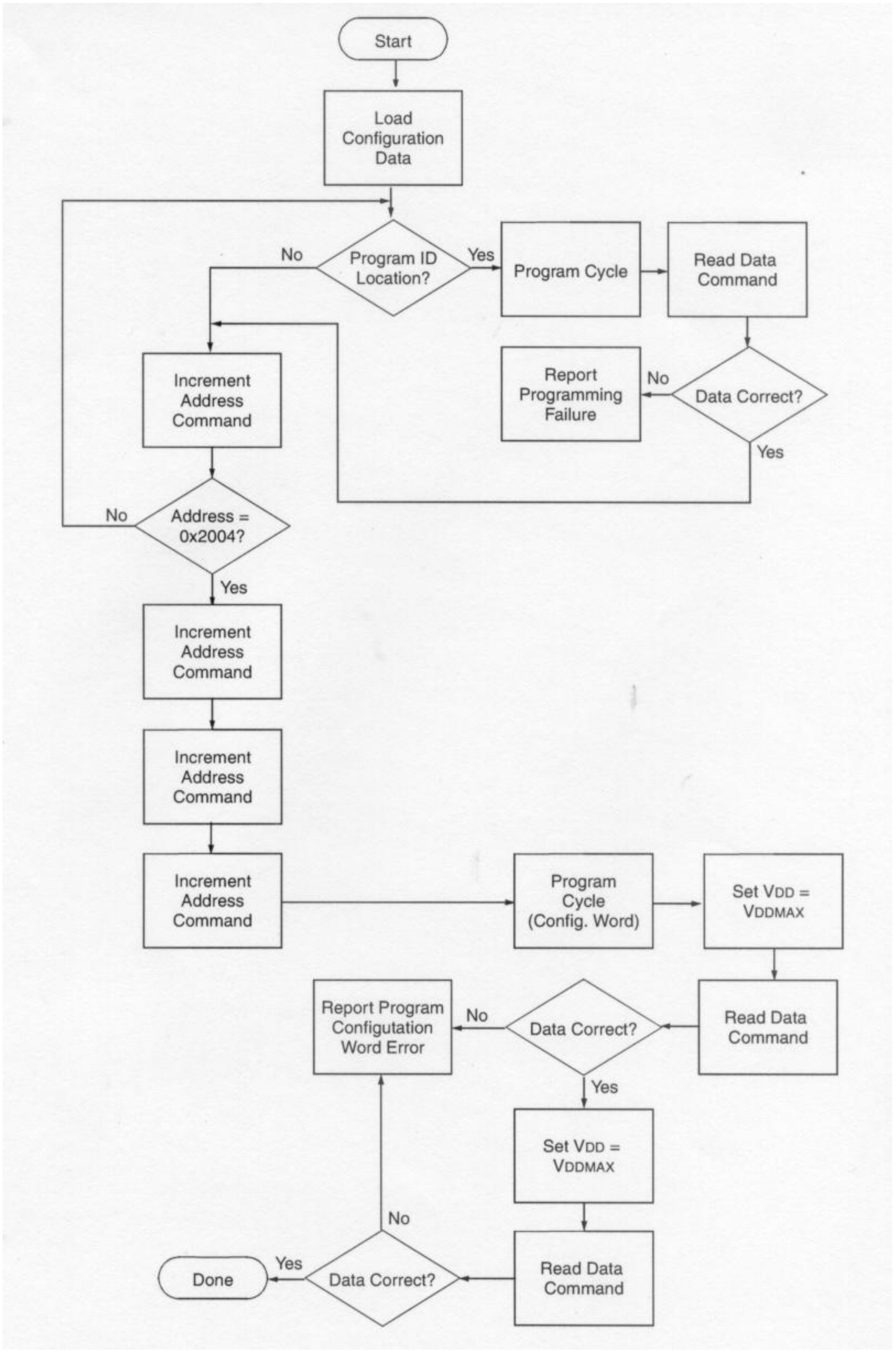


Figure 7.11 Flow chart for programming the ID locations and configuration word (In-circuit serial programming for PIC16F8XX FLASH MCUs, p.171).

To implement these flow charts and apply the internal commands, the programming interface from the 80C552 must drive the PIC16F877 with the correct signals.

The 80C552 makes use of the following I/O definitions throughout the program to access its own port pins for driving the PIC16F877's programming interface. The data line is bi-directional, therefore it is necessary to use separate pins (as explained in section 7.4).

```
#define MCLR        P1_5        // connected to MCLR (PIC)
#define CLK         P1_2        // connected to RB6 (PIC)
#define DATA_OUT    P1_3        // connected to RB7 (PIC)
#define DATA_IN     P1_6        // connected to RB7 (PIC)
#define LVP         P1_4        // connected to RB3 (PIC)
```

Figure 7.12 I/O definitions for programming the PIC16F877.

Pin RB6 is used as the clock input by the PIC16F877 and RB7 is used for entering command bits and data input/output during serial programming. The 80C552 makes use of the **sendbit()** and **recvbit()** functions to send a bit (a HIGH or a LOW) and receive a bit from the PIC16F877.

```
void sendbit(bit b)
{
        if (b==1) DATA_OUT = 1; else DATA_OUT = 0;
        CLK = 1;
        delay_us(1);                    // tset1
        CLK = 0;                        // clocked into PIC on this edge (falling)
        delay_us(1);                    // thld1
        DATA_OUT = 0;                   // idle with data line low
}


bit recvbit(void)
{
        bit b;
        CLK = 1;
        delay_us(1);                    // tdly3
        CLK = 0;                        // data is ready just before this
        b = DATA_IN;
        delay_us(1);                    // thld1
        return b;
}
```

Figure 7.13 Bit I/O functions.

Each bit is latched on the falling edge of the clock (RB6 or CLK). After sending a bit on the RB7 line, it is very important to make the DATA_OUT pin HIGH to allow the PIC16F877 to drive this line if the 80C552 needs data from it. The **delay_us()** function is used to implement the minimum setup and hold times according to the PIC16F877's AC/DC specifications.

The programming processor makes use of the **sendcmd()** function to send a 6-bit command to the PIC16F877. At the same time it also uses the **senddata()** and **recvdata()** functions to send and receive a whole 14-bit word from the PIC16F877.

```
void sendcmd(byte b)
{
        byte XDATA_AREA i;

        delay_us(2);                        // thld0
        for(i = 6; i > 0; i--)
        {
                sendbit(b & 1);             // send LSB first
                b = b >> 1;
        }
        delay_us(2);                        // tdly2
}

void senddata(word w)                       // sends 14-bit word from bottom of w
{
        byte XDATA_AREA i;

        delay_us(2);                        // thld0
        sendbit(0);                         // one garbage bit
        for(i = 14; i > 0; i--)
        {
                sendbit(w & 1);             // 14 data bits LSB first
                w = w >> 1;
        }
        sendbit(0);                         // one garbage bit
        delay_us(2);                        // tdly2
}

word recvdata()                             // receives 14-bit word, LSB first
{
        byte XDATA_AREA i;
        bit b;
        word XDATA_AREA w = 0;

        DATA_OUT = 1;                       // enables DATA_IN as input
        delay_us(2);                        // thld0
        recvbit();                          // one garbage bit
        for(i = 0; i < 14; i ++)
        {
                b = recvbit();
                w = w | ((word)b << i);     // 14 data bits
        }
        recvbit();                          // one garbage bit
        delay_us(2);                        // tdly2
        DATA_OUT = 0;                       // idle with DATA line low
        return w;
}
```

Figure 7.14 Data send and receive functions.

These functions make use of the above mentioned bit I/O functions to fulfil its function. Commands that have data associated with them are specified to have a minimum delay of 1us between the command and the data. Once again it is important to keep the DATA_OUT pin at an idle state to allow the PIC16F877 to drive it.

The **progcycle()** function is used to establish the program cycle as was shown in the flow chart of Figure 7.10. The function simply sends a 6-byte command followed by a 14-bit argument, issues the BEGIN PROGRAMMING command and waits for 100 microseconds.

```
void progcycle(byte cmd, word arg)                    // send a command and argument
{
        sendcmd(cmd);
        senddata(arg);
        sendcmd(BEGINERASEPROG);
        delay_us(100);
}
```

Figure 7.15 The program cycle function.

The **programall()** function is the main ICSP function which will program the different areas of the PIC16F877. It makes use of all the abovementioned functions to physically access the programming interface for the PIC16F877. The function uses 6 parameters:

- **Mode**

  Specifies if the function must be used to verify (= 0) the contents of the specified area of the PIC16F877 or whether be used to program (= 1) that area.
- **Mask**

  Since the program uses two memory locations for each data word (i.e. 16 bits), the function uses this mask to specify which data bits must be ignored (e.g. a mask of 0x3FFF, or 0011 1111 1111 1111 in binary, will ignore the 2 most significant bits).
- **Writecommand**

  Specifies the 6-bit command for the particular PICmicro device (PIC16F877 in this case) for writing to the specified area.
- **Readcommand**

  Specifies the 6-bit command for the particular PICmicro device (PIC16F877 in this case) for reading from the specified area.

- **Base**

Indicates where the start of the data in EEPROM is for the specified area that needs to be programmed.

- **Used**

Indicates to the function how many words are used of the specified area for programming.

```
bit programall(int mode, word mask, byte writecommand, byte readcommand, word base, word used)
{
        word XDATA_AREA i, w, temp;
        unsigned char XDATA_AREA lo, hi;
        float XDATA_AREA previous, t;

        if (base == PBASE)                                      //used for calculating % programmed
                previous = 0;
        else if (base == IBASE)
                previous = PUSED;
        else if (base == CBASE)
                previous = PUSED+IUSED;
        else if (base == DBASE)
                previous = PUSED+IUSED+CUSED;

        printf("%04X bytes to program\nProgress: ", used);      //shows % finished

        for(i = base * 2; i < (base + used) * 2; i=i+2)         //start reading from base up to number
        {                                                       //of words used
                if (((i - (base * 2)) % 16) == 0)
                        printf("%04X\b\b\b\b", i - (base * 2));
                hi = jbi_program[i];                            //read HI byte
                lo = jbi_program[i+1];                          //read LO byte
                temp = (hi << 8) | lo;
                printf("%04X    ", i/2);
                if (mode == PROGRAM)                            //program word if specified as program
                        progcycle(writecommand, (temp & mask));         //use mask to ignore unused bits
                printf("%04X\n", temp & mask);
                sendcmd(readcommand);                                   //read word from PIC16F877
                w = (recvdata() & mask);                                //verify word
                if (w != (temp & mask))                                 //display error if not verified
                {
                        printf("Failed at %04X: Expecting %04X, found %04X.\n", i/2, temp & mask, w);
                        return 0;
                }
                sendcmd(INCREMENTADDRESS);                              //increment PC to next location
                previous++;
                t = (previous/(PUSED+IUSED+CUSED+DUSED) * 100) + 100;           //loading percentage
                write_status(t);
        }
        return 1;
}
```

Figure 7.16   The main programming function.

This function can be used to program the Program memory, Data memory, ID locations and Configuration word of the PIC16F877, without the need of a separate function for each area.

7.6.2 JAM BYTE-CODE PLAYER

The Jam language has two parts: the Jam File and the Jam Player (Embedded programming using the 8051 & Jam byte-code, p.4). A Jam File, which contains all the information to program ISP-capable devices, is generated from the MAX+PLUS II development software. This file is then transferred via the Internet to the board and stored in one of the I^2C EEPROMs (as shown in Figure 7.17). Like with the PIC16F877 programming, this file must first be read into RAM before any programming can be done. The Jam Player runs on the 80C552 processor (stored in ROM), interprets the information in the Jam File and generates the binary data stream for device programming. Because updates may only be needed by and are confined to the Jam File, the Jam Player requires no changes and is used to program any vendor's device.

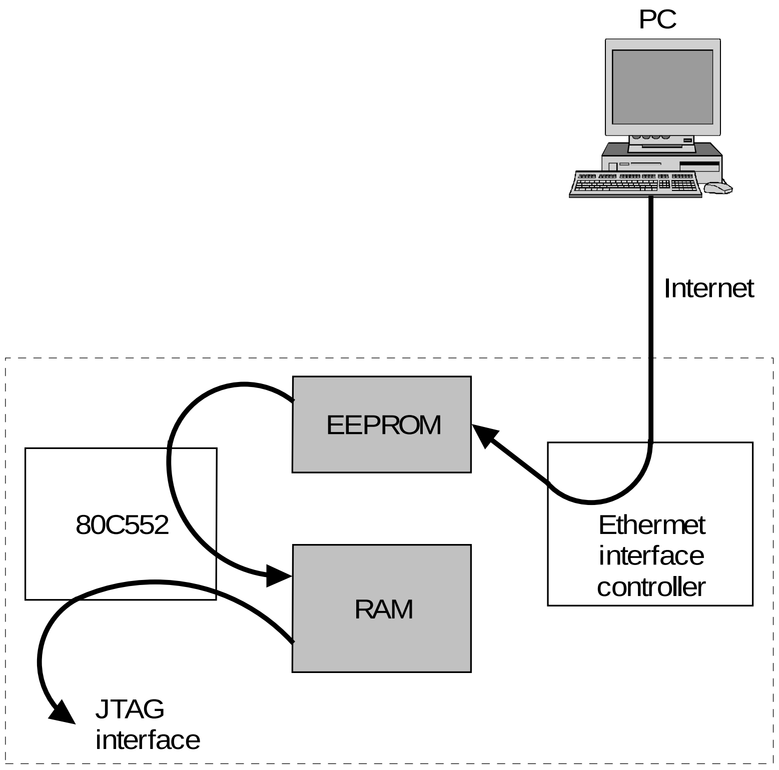


Figure 7.17 Transferring and loading of the Jam file.

Figure 7.18 shows a block diagram of how in-system programming is achieved with the Jam language.

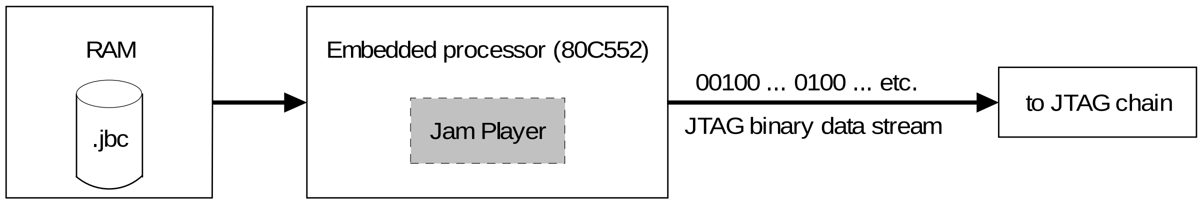


Figure 7.18 Block diagram of ISP using a Jam file and Jam Player (Adapted from Embedded programming using the 8051 & Jam byte-code, p.4).

7.6.2.1 The Jam file

Jam Files are compact files containing programming data and algorithm information needed to program any device through the IEEE Std. 1149.1 JTAG port. Altera products support two separate implementations of the Jam File: the Jam Byte-code File (.jbc) and the ASCII Jam File (.jam). The JBC file is a binary file, while the Jam file is text only. In this project only the JBC file is implemented because it provides much smaller file sizes and faster programming times. The block diagram in Figure 7.19 shows how a JBC file is generated for in-system programming using the MAX+PLUS II software.

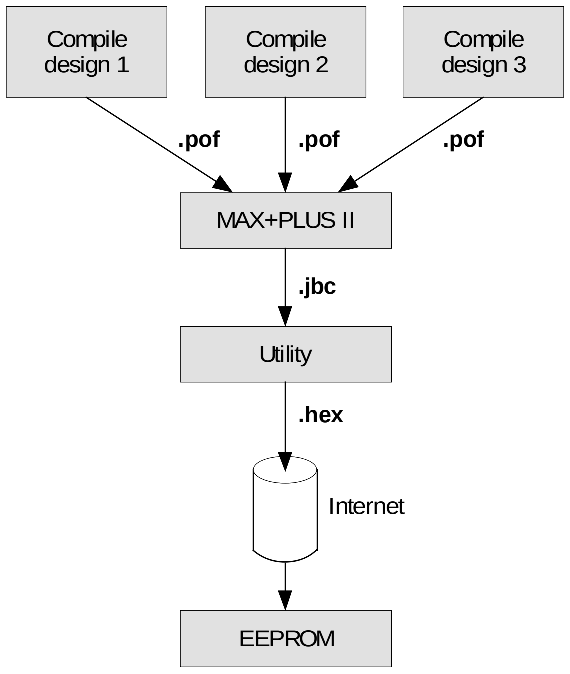


Figure 7.19 Generating a JBC file using the MAX+PLUS II software.

7.6.2.2 The Jam Player

The Jam Player is a C program that parses the JBC file (which is read from EEPROM into RAM), interprets each Jam instruction, and reads and writes data to and from the JTAG chain. Figure 7.20 illustrates the Jam Player source code structure.

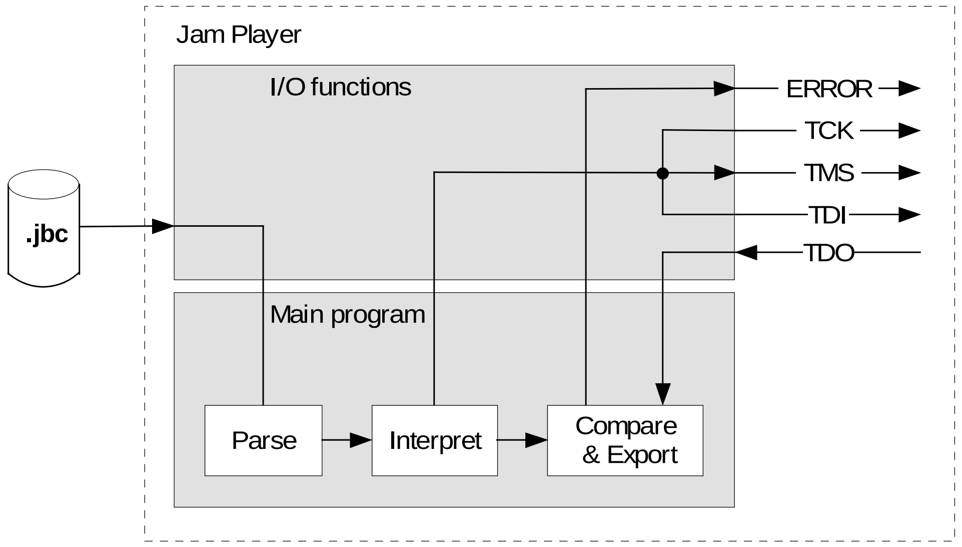


Figure 7.20 Jam Player source code structure (Adapted from Embedded programming using the 8051 & Jam byte-code, p.9).

The Jam Player resides permanently in system memory (ROM), where it interprets the commands given in the JBC file (now in RAM) and generates a binary data stream for device programming. The structure confines all upgrades to the JBC file and allows the Jam Player to adapt to any system architecture.

The Jam Byte-code Player is written in the C programming language for the 8051 core processor and can be downloaded from www.jamisp.com. The source code can be compiled for any 8051 variant as long as the supporting compiler can compile C code. To implement the Jam Byte-code Player on the 80C552 in this project, some changes had to be made to the source code:

- The JBC file can either be stored in ROM or RAM. In this project the JBC file will constantly change, therefore must be stored in RAM. To specify that it will be stored in RAM, the following line must be added to the beginning of the source code:

  ```
  #define JBC_FILE_IN_RAM = 1
  ```

The JBC file will actually be stored in one of the I^2C EEPROMs, be read into RAM and from there will be read again to program the device. The **jbi_load_jbc_file()** function will be used to read the JBC file from EEPROM into RAM as shown in Figure 7.21.

```
JBI_RETURN_TYPE jbi_load_jbc_file(void)
{
        JBI_RETURN_TYPE XDATA_AREA crc_result = JBIC_IO_ERROR;
        unsigned int XDATA_AREA expected_crc;
        unsigned int XDATA_AREA actual_crc;
        unsigned int XDATA_AREA note_offset = 0;
        unsigned char XDATA_AREA lo, hi;
        word XDATA_AREA i, used;
        float XDATA_AREA temp=0, t;

        EA = 0;                                     //disable interrupts
        hi = IIread_eeprom(DEVICE2, 0x0000);        //reads HI byte of how many bytes are used
        lo = IIread_eeprom(DEVICE2, 0x0001);        //read LO byte
        used = (hi << 8) | lo;
        printf("Number of bytes to read: %04X \n", used);
        printf("Loading from ROM...\n");

        for (i = 0; i < used; i++)                  //start reading data from EEPROM
        {
        jbi_program[i] = IIread_eeprom(DEVICE2, i + 2);
                temp++;
                if ((i % 100) == 0)                 //update loading percentage every 100 bytes
                {
                        printf("%04X\b\b\b\b", i);
                        t = (temp/used) * 100;      //loading percentage
                        write_status(t);            //change status lines to indicate loading %
                }
        }
        printf("\nLoaded %u bytes\n", i);
        printf("Checking CRC\n");
        crc_result = jbi_check_crc(&expected_crc, &actual_crc);         //do CRC check on file
        if (crc_result == JBIC_SUCCESS)
        {
                printf("CRC matched: expected %04X, actual %04X\n",   expected_crc, actual_crc);
        }
        else
        {
                printf("CRC mismatch: expected %04X, actual %04X\n", expected_crc, actual_crc);
        }
        if (crc_result == JBIC_SUCCESS)                                 //Dump out NOTE fields
        {
                while (jbi_get_note(&note_offset, jbi_note_key, jbi_note_value, 256) == 0)
                {
                        printf("NOTE \"%s\" = \"%s\"\n", jbi_note_key, jbi_note_value);
                }
        }

        return (crc_result);
}
```

Figure 7.21 Loading the JBC file into RAM.

- The **jbi_jtag_io()** function is the interface to the IEEE Std. 1149.1 JTAG signals TDI, TMS, TCK and TDO. The signals are mapped to the 80C552's hardware ports as follows:

Signal	Hardware Port
TDI	P1.0
TMS	P1.1
TCK	P1.7
TDO	P3.5

Table 7.5 JTAG interface port pins.

The function receives three parameters indicating the required state for the TMS, TDI and TDO signals, and returns one parameter indicating the state of the TDI signal.

```
BIT jbi_jtag_io(BIT tms, BIT tdi, BIT read_tdo)
{
        BIT tdo = 0;

        P1_1 = tms;             /* set TMS signal */
        P1_0 = tdi;             /* set TDI signal */

        if (read_tdo)
        {
                tdo = P3_5;     /* read TDO signal */
        }

        P1_7 = 1;               /* strobe TCK signal */
        P1_7 = 0;

        return (tdo);
}
```

Figure 7.22 Function for interfacing to the JTAG signals.

- Pulses of varying widths are used to program the internal EEPROM cells of the MAX devices. The Jam Player uses the **jbi_delay()** routine to implement these pulse widths. This routine must be customized based on the speed of the processor and the time it takes the processor to execute a single loop. The 80C552 in this project, however, is running on a 12Mhz crystal which is slow already, so the delay in the routine is just set to the minimum (= 0).

```
void jbi_delay(unsigned long microseconds)
{
        unsigned long count = microseconds >> 3;

        while (count != 0) count--;
}
```

Figure 7.23 Delay function for Jam Player.

7.6.2.3 Jam Player operation

The Jam Player provides an interface for manipulating the IEEE Std. 1149.1 JTAG TAP state machine. The TAP controller is a 16-state state machine that is clocked on the rising edge of TCK, and uses the TMS pin to control JTAG operation in a device. Figure 7.24 shows the flow of the TAP controller state machine.

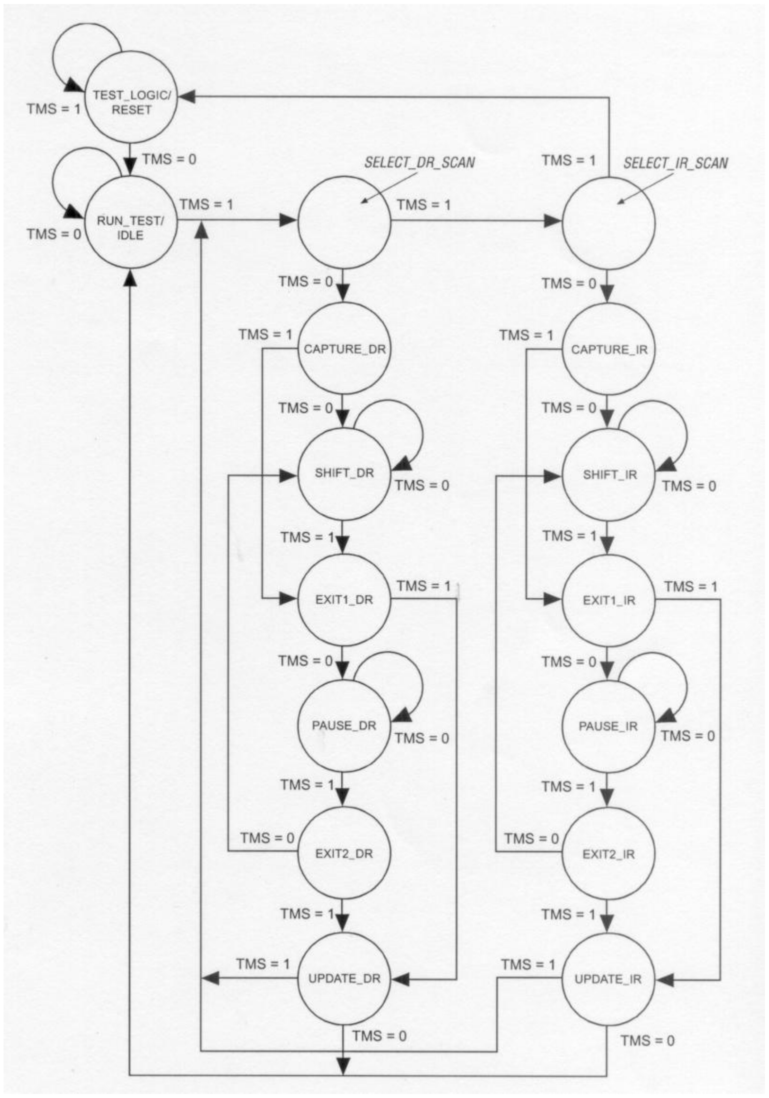


Figure 7.24 JTAG TAP controller state machine (Embedded programming using the 8051 & Jam byte-code, p.20).

While the Jam Player provides a driver that manipulates the TAP controller, the JBC File provides the high-level intelligence needed to program a given device. All Jam instructions that send JTAG data to the device involves moving the TAP controller through either the data register leg of the state machine or the instruction register leg.

The high-level Jam instructions are the DRSCAN instruction for scanning the JTAG data register, the IRSCAN instruction for scanning the instruction register, and the WAIT command that causes the state machine to sit idle for a specified period of time. Each leg of the TAP controller is scanned repeatedly, according to instructions in the JBC file, until all of the target devices are programmed.

7.6.3 I^{2}C FUNCTIONS FOR EEPROMs

Inter-Integrated Circuit (I^{2}C) is a two wire bus for eight bit data transfer applications (24LC256 Datasheet, p.1). The two wires (serial clock and serial data) carry information between the devices connected to the bus. The serial data wire is bi-directional but data may flow in only one direction at a given time. Each device on the bus can operate as either a transmitter or receiver, but not simultaneously. Some devices may be only transmitters while others only receivers. An example of a receive only device is a display, while a memory would both receive and transmit information. The device that generates the clock pulses is called the master and the device that receives is called the slave.

The 24LC256 is a 32K x 8-bit serial EEPROM and is used to store the source files in the this project. It uses two pins, serial clock (SCK) and serial data (SDA), to transmit and receive data from the 80C552. Three chip select pins (A0, A1, A2) are also used to allow more than one 24LC256 to be used on the one I^{2}C bus. The pins are hardwired to either +5V or GND to set the address for each EEPROM. Seeing that there are three chip select pins, there can only be a maximum of eight EEPROMs on the I^{2}C bus.

The 80C552 makes use of the following pin definitions to access the I^{2}C bus.

```
#define SCL_IN          P4_0
#define SCL_OUT         P4_1
#define SDA_OUT         P4_3
#define SDA_IN          P4_2
```

Figure 7.25 Pin definitions for accessing the I^{2}C bus.

The SDA and SCL line each have an input and an output pin on the 80C552 (as explained in section 7.4 in this chapter.

When the I^{2}C bus is not in use, the SDA and SCL lines must be HIGH. This means that the output pins SDA_OUT and SCL_OUT must also be HIGH, otherwise these pins will pull the SDA and SCL lines LOW. Data can only be transmitted when the bus is not busy. Any data transfer must be preceded by a START condition on the bus and ended with a STOP condition. The START condition is created when the SDA line is pulled LOW before the SCL line, and a STOP condition when the SCL line is pulled HIGH before the SDA line. The 80C552 uses the **IIstart()** and **IIstop()** functions to create the START and STOP conditions.

```
bit IIstart(void)
{
        SDA_OUT = 1;                                        // release SDA and SCL lines
        SCL_OUT = 1;
        if ((SCL_IN != 1) || (SDA_IN != 1))                 // check if bus is free
                return(0);
        delay_us(1);
        SDA_OUT = 0;                                        // pull SDA LOW before SCL
        delay_us(1);
        SCL_OUT = 0;
        delay_us(1);
        return(1);
}

void IIstop(void)
{
        SCL_OUT = 0;
        SDA_OUT = 0;
        delay_us(1);                                        // pull SCL HIGH before SDA
        SCL_OUT = 1;                                        // release SCL line
        delay_us(1);
        SDA_OUT = 1;                                        // release SDA line
        delay_us(1);
}
```

Figure 7.26 Functions for creating START and STOP conditions on the I^{2}C bus.

The **IIclk()**, **IIout()**, and **IIin()** functions are used to write and read 8 bits of data to the I^{2}C bus. Data is transmitted and received with the most significant byte (MSB) first. Each receiving device, when addressed, is obliged to generate an acknowledgement signal after the reception of each byte of data. The master must generate an extra clock pulse which is associated with this acknowledge bit. The slave must pull the SDA line LOW during the acknowledgement pulse to acknowledge the data. The **IIread_ack()** function is used for this purpose in the program. It will return a 1 if the data was acknowledged, otherwise it will return a 0.

```
void IIclk(void)                                    // generates clock pulse
{
        SCL_OUT = 1;
        delay_us(1);
        SCL_OUT = 0;
}

void IIout(byte DATA)                               // transmits a byte onto the I2C bus
{
        byte i;

        for (i = 128; i >= 1; i=i/2)                // MSB first
        {
                if (i & DATA)
                        SDA_OUT = 1;
                else
                        SDA_OUT = 0;
                delay_us(1);
                IIclk();                            // generate clock pulse
        }
        SDA_OUT = 1;                                // release SDA line
        delay_us(1);
}


byte IIin(void)
{
        byte i, DATA=0;

        SDA_OUT = 1;                                // to enable SDA_IN as input
        for(i=128; i >= 1; i=i/2)                   // MSB first
        {
                SCL_OUT = 1;
                delay_us(1);
                if (SDA_IN)                         // read bit on lagging edge of clock line
                        DATA = DATA + i;
                SCL_OUT = 0;
                delay_us(1);
        }
        return(DATA);
}
```

Figure 2.27   Functions for accessing the I^2C bus interface.

```
bit IIread_ack(void)
{
            SDA_OUT = 1;
            SCL_OUT = 1;
            delay_us(1);
            if (SDA_IN)                                    // if SDA is HIGH, then byte IS NOT acknowledged
            {
                        SCL_OUT = 0;
                        return(0);
            }
            else                                           // if SDA is LOW, then byte IS acknowledged
            {
                        SCL_OUT = 0;
                        return(1);
            }
}
```

Figure 7.28 Function to test if a sent data byte was acknowledged.

The 80C552 will not need to write data to the EEPROMs, therefore no EEPROM write functions were added to the program. The main purpose of the I^{2}C bus for the 80C552 is to read data from the serial EEPROMs and is accomplished by using the **IIstart_read()** and **IIread_eeprom()** functions. Reading data from a specific memory location in the serial EEPROM involves the following steps from the 80C552:

- Generate a START condition.
- Send a control word with R/W bit set as 0.

  The control word consists of a 4-bit control code which is always 1010 for the 24LC256. The next three bits are the chip select bits (A2, A1, A0). These bits allow the selection of individual 24LC256 devices on the same I^{2}C bus. The last bit (R/W) simply defines the operation to be performed. When a one read operation is selected, and when it is a zero write operation is selected.
- Check acknowledgement bit.
- Send the HI byte of the required memory location to be read.
- Check acknowledgement bit.
- Send the LO byte of the required memory location to be read.
- Check acknowledgement bit.
- Generate a START condition.
- Send a control word with R/W bit set as 1.
- Check acknowledgement bit.
- Read the incoming data (MSB first).
- The master will not acknowledge the transfer, but does generate a STOP condition.

Figure 7.29 summarises this read operation.

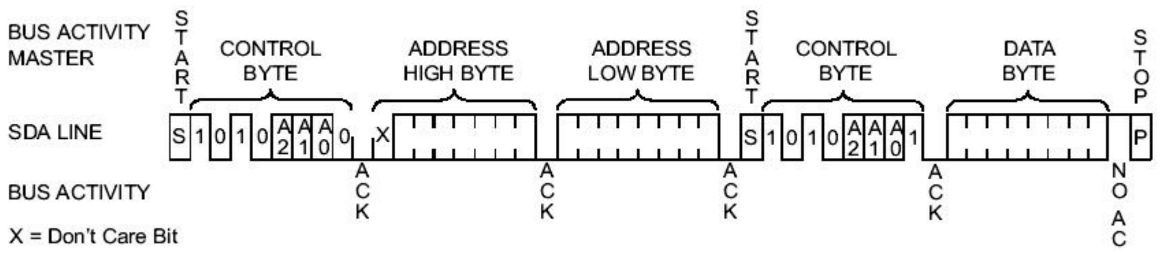


Figure 7.29 Reading data from the 24LC256 (24LC256 Datasheet, p.10).

```
#define DEVICE0            0xA0            // device Ids according to each device's hardwiring
#define DEVICE1            0xA2            // of pins A2 – A0.  First 4-bits of ID is 1010 which is
#define DEVICE2            0xA4            // default for the 24LC256.
#define DEVICE3            0xA6
#define DEVICE4            0xA8
#define DEVICE5            0xAA
#define DEVICE6            0xAC
#define DEVICE7            0xAE
#define READ               1
#define WRITE              0

void IIstart_read(byte CS, word ADDRESS)           // starts the reading procedure from EEPROM
{
        IIstart();                                 // generate START condition
        IIout(EEPROM | CS | WRITE);                // send control word - write
        IIread_ack();                              // check ack
        IIout((byte)(ADDRESS >> 8));               // send HIGH byte of ADDRESS
        IIread_ack();                              // check ack
        IIout((byte)(ADDRESS & 0x00ff));           // send LO byte of ADDRESS
        IIread_ack();                              // check ack
        IIstart();                                 // generate START condition
        IIout(EEPROM | CS | READ);                 // send control word - read
        IIread_ack();                              // check ack
}


byte IIread_eeprom(byte CS, word ADDRESS)          // reads the data from EEPROM
{
        byte DATA = 0;

        IIstart_read(CS, ADDRESS);
        DATA = IIin();                             // read data
        IIstop();                                  // generate STOP condition

        return(DATA);
}
```

Figure 7.30 Functions for reading data from a specific memory location in the 24LC256.

7.6.4 MAIN PROGRAMMING LOOP

With all the functions for ICSP with the PIC16F877, ISP for the MAX7000S and the I^2C communications defined, the rest of the program is quite simple. All the 80C552 has to do now is to wait in a loop until a command is received from the Ethernet interface controller. The **read_cmd()** function monitors the CMD0 and CMD1 lines from the Ethernet interface to determine the required instruction. It almost acts like the popular **getch()** C function; the program waits in a loop until a key is pressed (a command is received).

```
byte read_cmd(void)
{
        byte XDATA_AREA val=0;

        CMD0 = 1;                                   // enables CMD0 and CMD1 as inputs
        CMD1 = 1;

        while((CMD0 == 0) & (CMD1 ==0));            // wait until CMD0 or CMD1 changes to HIGH

        if ((CMD0 == 1) & (CMD1 ==0))               // program I/O controller (val = 2)
                val=1;
        else
                if ((CMD0 == 0) & (CMD1 ==1))       // program hardware (val = 1)
                        val=2;
                else
                        if ((CMD0 == 1) & (CMD1 ==1)) // future command
                                val=3;
                        else
                                val=0;              // unknown command

        while((CMD0 != 0) & (CMD1 != 0));           // wait until Ethernet interface controller takes both
        return(val);                                // lines LOW again.  Ethernet interface controller
}                                                   // only pulses the lines for the command
```

Figure 7.31 Reading an instruction from the Ethernet interface controller.

Right throughout the program the **printf()** statement is used to write information to the 80C552's RS-232 module. This is purely done for debugging purposes and can be omitted.

Figure 7.32 shows the main programming loop. It makes use of a **switch()** statement to determine the required operations for the command received. The **write_status()** function manipulates the Status0 – Status3 lines to indicate the percentage completed (as described in section 7.4).

```
while (1)
{
        printf("\nEmbedded Ethernet Controller v2.00\n");                    //display message menu
        printf("1.  Load, Program & Run Reconfigureable Hardware\n");
        printf("2.  Load, Program & Run Controller CPU\n");
        printf("3.  Load, Program & Run Server CPU\n\n-> ");
        b = read_cmd();                                                       //wait for command
        printf("\n");
        switch(b)
        {
                case 0:   printf ("Invalid choice!\n");      break;           //invalid choice

                case 1:   {
                                write_status(1);                              //force loading 4%
                                printf ("********** Hardware **********\n");
                                crc_result = jbi_load_jbc_file();             //load file from EEPROM
                                init_list = jbi_init_list_program;            //set ISP parameters
                           }    break;
                case 2:   {
                                write_status(1);                              //force loading 4%
                                printf ("********** Controller CPU **********\n");
                                printf("Erasing PIC ...\n");
                                EraseAll();                                   //erase contents of PIC
                                printf("Loading ...\n");
                                LoadICSPFromRom();                            //load file from EEPROM
                                printf("Programming Program Memory ...\n");
                                vppreset();                                   //enter programming mode
                                programall(PROGRAM,   PMASK,   LOADPROGRAM,   READPROGRAM,
PBASE, PUSED);                                                                //program program memory
                                printf("Programming ID Locations ...\n");
                                sendcmd(LOADCONFIG);
                                senddata(DEFAULTCONFIG);
                                programall(PROGRAM,   IMASK,   LOADPROGRAM,   READPROGRAM,
IBASE, IUSED);                                                                //program ID locations
                                printf("Programming Configuration Byte ...\n");
                                for (i = 0; i < CBASE - IBASE - IUSED; i ++)
                                sendcmd(INCREMENTADDRESS);
                                programall(PROGRAM,   CMASK,   LOADPROGRAM,   READPROGRAM,
CBASE, CUSED);                                                                //program config word
                                vppreset();
                                printf("Programming EEPROM Memory ...\n");
                                programall(PROGRAM,   DMASK,   LOADDATA,   READDATA,   DBASE,
DUSED);                                                                       //program data memory
                                Stop();
                                Run();                                        //exit program mode & run
                                write_status(0);
                          }     break;
                case 3: {
                                                                              //future command
                          }     break;
                default: printf ("Invalid choice!\n");       break;           //invalid choice
        }  // end switch()

        if (b == 1)                                                           //MAX7000S selected
        {
                // Jam Player execution code
        }
}  //end while()
```

Figure 7.32   The main programming loop.

7.7 CONCLUSION

This chapter took an in depth look at in-system programmability with the PIC16F877 and the MAX7000S devices and Jam Player were explained in detail. The storage scheme for the source files was demonstrated and an example was also included. The software code for the different programming components was explained with extracts from the completed embedded code.

This chapter was the last hardware chapter and the next chapter will give a brief overview of the operation of the software involved for developing software for the devices on the board and for transferring the files. Accessing the board using a standard Web browser (e.g. Internet Explorer 5.0) will also be demonstrated.

CHAPTER 8
SOFTWARE ENVIRONMENT

8.1 OVERVIEW

This chapter will take a look at the software side of the project. First the operation and description of the software utility (EmEther) for transferring and programming the source files will be explained. It will be followed by a brief description of how to access the remote system. The chapter will then end with a short summary of the development software and reference material used in this project.

8.2 EMBEDDED ETHERNET (EmEther) UTILITY

The user must use the embedded Ethernet (EmEther) utility to transfer new source files to the remote board and issue the commands to the programming processor to start programming the required section. The utility will also provide a progress bar to indicate the percentage of completion.

8.2.1 OPERATION

Figure 8.1 shows a screen capture of the EmEther utility.

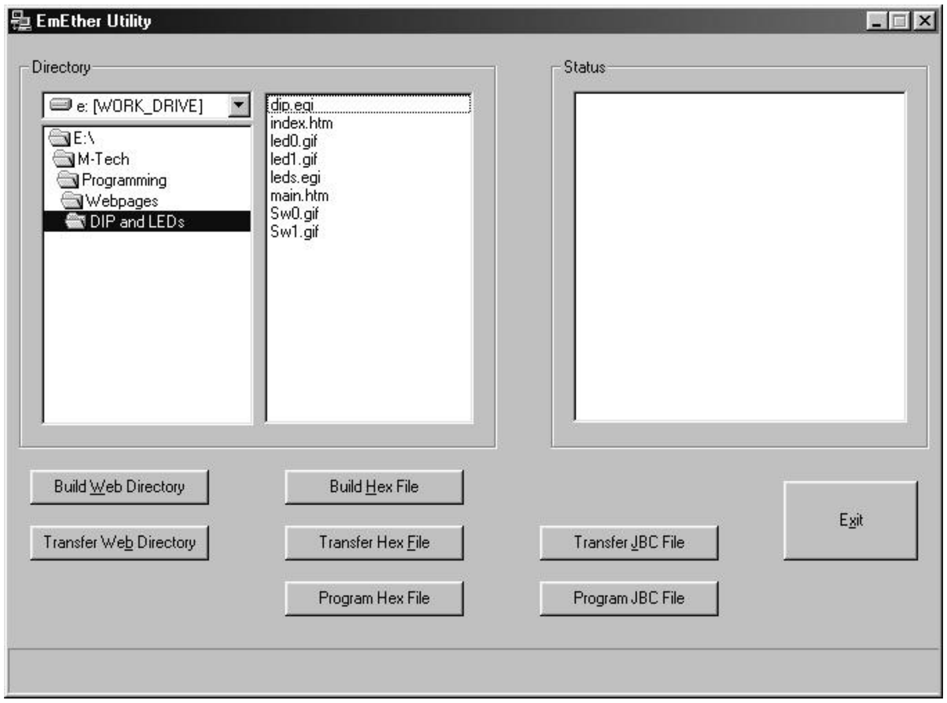
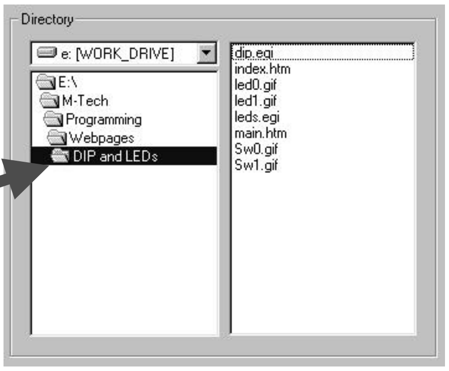


Figure 8.1 The EmEther utility.

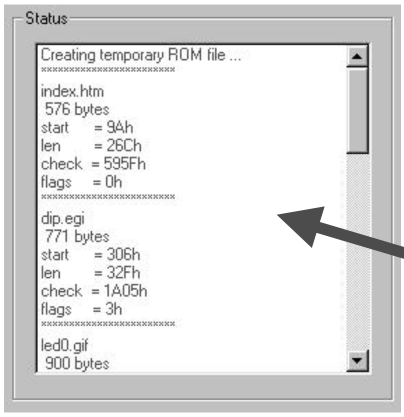
Building and transferring HTML documents:

1. Choose the required Web page directory to be transferred.



2. Click the **Build Web Directory** button.

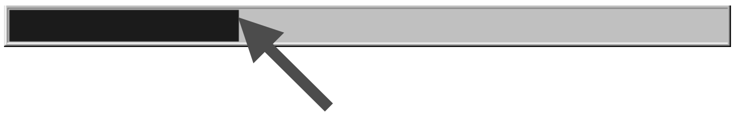
   Before a Web page directory can be transferred to the remote system, it must first be built into a special file format that allows it to be stored in the remote system's EEPROM. A detailed description of the built file will be displayed in the **Status** frame. It gives a breakdown of each file's name, size, start address in EEPROM, length, checksum and EGI flags.



```
Creating temporary ROM file ...
xxxxxxxxxxxxxxxxxxxxxxxxx
index.htm
 576 bytes
start    = 9Ah
len      = 26Ch
check  = 595Fh
flags    = 0h
xxxxxxxxxxxxxxxxxxxxxxxxx
dip.egi
 771 bytes
start    = 306h
len      = 32Fh
check  = 1A05h
flags    = 3h
xxxxxxxxxxxxxxxxxxxxxxxxx
led0.gif
 900 bytes
```

3. Click the **Transfer Web Directory** button.

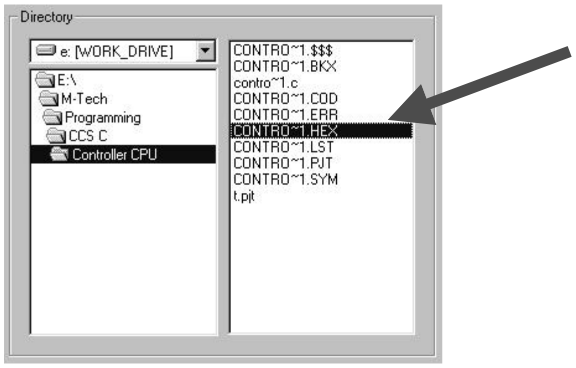
   The progress bar at the bottom of the window will indicate the percentage of file transfer.



   The transferred Web directory takes immediate effect on the remote system once it is transferred and does not need to be issued a command to program it.

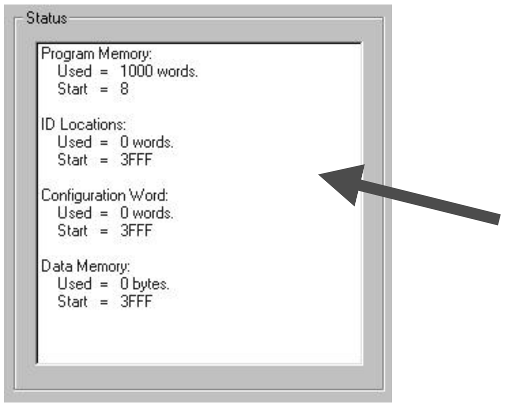
Building, transferring and programming PICmicro source files:

1. Choose the required PICmicro source file (.hex) to be transferred.



2. Click the **Build Hex File** button.

   Before a PICmicro source file can be transferred to the remote system, it must first be built into a special file format that allows it to be stored in the remote system's EEPROM. A detailed description of the built file will be displayed in the **Status** frame. It gives a breakdown of the size of each of the memory areas in the PIC16F877 and their start addresses inside the EEPROM.

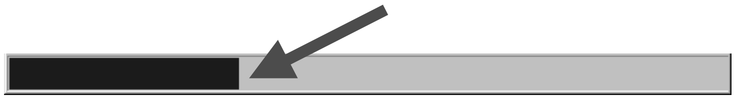


Status

```
Program Memory:
   Used =  1000 words.
   Start  =  8

ID Locations:
   Used =  0 words.
   Start  =  3FFF

Configuration Word:
   Used =  0 words.
   Start  =  3FFF

Data Memory:
   Used =  0 bytes.
   Start  =  3FFF
```

3. Click the **Transfer Hex File** button.

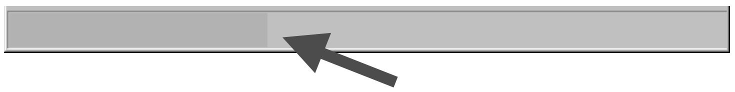
   The progress bar at the bottom of the window will indicate the percentage of file transfer.



4. Click the **Program Hex File** button.

   Before the transferred PICmicro source file can take effect on the remote system, the programming interface on the remote system must first get the instruction to load the source file from EEPROM and then program the PIC16F877. The first progress bar will indicate how much of the source file is loaded by the programming processor from the EEPROM.

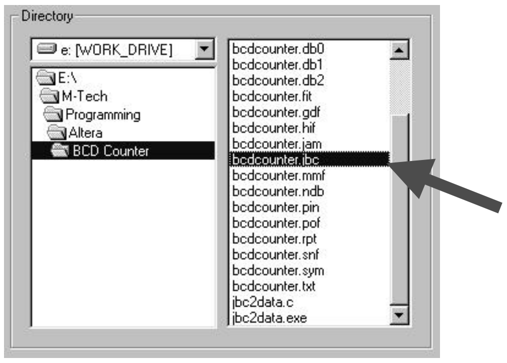
   Shortly after the programming processor has read the whole source file from EEPROM, the next progress bar will appear, indicating the percentage of the actual programming of the PIC16F877.



   The PIC16F877 will now be updated with the new source file and will take immediate effect.

<u>Transferring and programming MAX7000S source files:</u>

1. Choose the required MAX7000S source file (.jbc) to be transferred. If other JTAG devices were added to the circuit, those files can also be transferred using the same method.



Directory

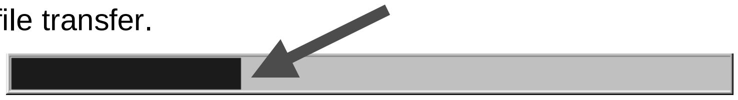
e: [WORK_DRIVE]

E:\
M-Tech
Programming
Altera
BCD Counter

bcdcounter.db0
bcdcounter.db1
bcdcounter.db2
bcdcounter.fit
bcdcounter.gdf
bcdcounter.hif
bcdcounter.jam
bcdcounter.jbc
bcdcounter.mmf
bcdcounter.ndb
bcdcounter.pin
bcdcounter.pof
bcdcounter.rpt
bcdcounter.snf
bcdcounter.sym
bcdcounter.txt
jbc2data.c
jbc2data.exe

2. Click the **Transfer JBC File** button.

   The source file for the MAX7000S is already in a suitable file format for storing it in one of the EEPROMs on the remote system, and therefore it is not necessary to build the file first. The progress bar at the bottom of the window will indicate the percentage of file transfer.



3. Click the **Program JBC File** button.

   Before the transferred MAX7000S source file can take effect on the remote system, the programming interface on the remote system must first get instruction to load the source file from EEPROM and then program the MAX7000S. The progress bar will indicate how much of the source file is loaded by the programming processor from the EEPROM.

   Unfortunately the Jam Byte-Code Player does not provide a feedback of the progress for programming a JTAG device. An EPM7128S device has a programming time of roughly 5 minutes. The next chapter will demonstrate three different test applications for the system and the programming times for each application will be indicated. The user can see if the programming processor is busy with programming a device by simply trying to access the Web page. If the system is still busy, a "Busy programming!" page will appear.

8.2.2 SOURCE CODE DESCRIPTION

The EmEther utility was developed using Visual Basic 6.0. The source code can be divided into four different sections: building source files, transferring source files, instructing the remote programming processor, and progress indication.

8.2.2.1 Building source files

Only source files for the PIC16F877 and the HTML documents for the Web server needs to be compiled into a special format. This format will allow the file to be stored in one of the I^2C EEPROMs of the remote system. The source file for the MAX7000S device is already in the correct format and can be stored as is.

The **cmdBuildWebDir_Click()** and **cmdBuildHexFile_Click()** functions are used to build the source files. It simply creates an image of what the EEPROM contents should be on the local PC's drive ("hextemp.rom" and "webtemp.rom"). The file structure for both these sections has been explained in Chapter 4 and 7, therefore a detailed description of the source code is not necessary.

8.2.2.2 Transferring source files

Transferring the source files means transferring the "hextemp.rom" and "webtemp.rom" files via the Internet to the remote system's EEPROMs. The .jbc file for the MAX7000S is transferred just as it is.

Send_Socket() is the most important function for transferring source files via the Internet.

```
Private Sub Send_Socket(data As String)
   Winsock1.Connect                           ‘ create TCP socket connection on port 1000
   Do Until Winsock1.State = sckConnected     ‘ wait until connected
      DoEvents: DoEvents: DoEvents: DoEvents  ‘ dummy
      If Winsock1.State = sckError Then       ‘ if error, display message
         MsgBox "Problem connecting!"
         Exit Sub
      End If
   Loop

   Winsock1.SendData data                     ‘ send data via socket connection

   Do Until Winsock1.State = sckClosing       ‘ wait until connection is closed by remote system
      DoEvents: DoEvents: DoEvents: DoEvents  ‘ dummy
      If Winsock1.State = sckError Then       ‘ if error, display message
         MsgBox "Problem connecting!"
         Exit Sub
      End If
   Loop
   Winsock1.Close                             ‘ close TCP socket
End Sub
```

Figure 8.2 Sending data via the Internet using a TCP socket.

The function makes use of the Winsock ActiveX control to establish a TCP socket connection with the remote system. A socket connection is basically a data pipe that allows reliable bi-directional data transfer between two nodes (Wilder, 1999, p.169). The Winsock control uses the following two lines to setup the remote destination's IP address and port number:

```
Winsock1.RemoteHost = "picweb.petech.ac.za"
Winsock1.RemotePort = 1000
```

The EmEther utility transfers source files using the **cmdTransWebDir_Click()**, **cmdTransHexFile_Click()** and **cmdTransJBCFile_Click()** functions. Each of these functions operates exactly the same and makes use of the **packet_data** array to break up the file to be transferred into data packets of 64 bytes each (data can only be written to a I^{2}C EEPROM in 64-byte pages). The **packet_listcount** variable indicate the number of packets that need to be transferred.

```
Const PACKET_SIZE As Byte = 64                 ‘ size of the data packets in bytes

Open "webtemp.rom" For Binary As #1            ‘ determine the size of the source file in bytes
leng = LOF(1)
For i = 0 To leng – 1                          ‘ read the contents of the source file into a buffer
   buff(i) = Input(1, #1)
Next i
Close #1

temp_str = ""                                  ‘ temporary string
next_stop = PACKET_SIZE                        ‘ next break point in the source file for a complete packet
packet_listcount = 0                           ‘ number of packets that need to be transferred = 0

For i = 0 To leng – 1                          ‘ start reading from the buffer
   If i < next_stop Then                       ‘ if the current byte position is next than the next break
      temp_str = temp_str + buff(i)            ‘ add current byte to temporary string
   Else
      packet_data(packet_listcount) = temp_str ‘ load current packet with the temporary string
      packet_listcount = packet_listcount + 1  ‘ increase number of packets to be transferred
      next_stop = next_stop + PACKET_SIZE      ‘ increase next break position with 64 bytes
      temp_str = ""                            ‘ clear temporary string
      temp_str = temp_str + buff(i)            ‘ add current byte to temporary string
   End If
Next i
If leng - 1 < next_stop Then                   ‘ if last packet has less than 64 bytes
   packet_data(packet_listcount) = temp_str
End If
```

Figure 8.3 Breaking up the source file into data packets for transferring.

These functions transfer each packet using a 4-byte header. Each packet is sent with a start address so that the remote Web server knows where to store the data in the EEPROM. The packet length and source file type is also added for the Web server to identify in which EEPROM the data should be stored and how much data will be stored. The final packet is shown in Figure 8.4.

Packet Length	Source File Type	EEPROM Start Address - Hi byte	EEPROM Start Address - Lo byte	Data

Figure 8.4 The data transfer packet.

```
   Const WEB_FILE_TYPE As Byte = 0                    ‘ constants to indicate file types
   Const ICSP_FILE_TYPE As Byte = 1
   Const JTAG_FILE_TYPE As Byte = 2

   high_address = 0                                   ‘ start at EEPROM address 0x0000
   low_address = 0

   For j = 0 To packet_listcount                      ‘ start transferring packets one by one
      packet_len = Chr(Len(packet_data(j)) + 4)       ‘ packet length = header length + data length

      Send_Socket (packet_len + Chr(WEB_FILE_TYPE) + Chr(high_address) + Chr(low_address) +
packet_data(j))                                       ‘ send packet using the TCP socket on port 1000

      low_address = low_address + PACKET_SIZE         ‘ increase the next EEPROM start address with
      If low_address > 255 Then                       ‘ 64 bytes
         low_address = 0
         high_address = high_address + 1
      End If
   Next j
```
Figure 8.5 Transferring the source file using data packets.

The main purpose of these functions is to transfer the built source files as they are and store them in the remote system’s EEPROMs exactly in the same order, starting at EEPROM address 0x0000.

8.2.2.3 Instructing the remote programming processor

For the update source files of the PIC16F877 and the MAX7000S to take effect, the programming processor first needs to be instructed to program the devices with the contents of the corresponding EEPROM. Only the transferred HTML documents for the Web server takes immediately effect and does not need to be programmed.

The EmEther utility makes use of the **cmdProgHexFile_Click()** and **cmdProgJBCFile_Click()** functions to issue the begin programming command to the remote programming processor. The command is also transferred via the Internet using a packet, although this packet only has two bytes. The first byte simply indicates to the Web server that this is not a file transfer, but a command packet and always has the value of 255 (or 0xFF). The second byte indicates which device must be programmed. Both these functions again operate exactly the same.

```
Const PROGRAM_COMMAND As Byte = &HFF            ‘ command constants
Const PROGRAM_SERVER As Byte = &H3
Const PROGRAM_CONTROLLER As Byte = &H2
Const PROGRAM_HARDWARE As Byte = &H1

Private Sub cmdProgHexFile_Click()
   If (PROGRAMMING_STATUS = 0) Then              ‘ if utility IS NOT busy programming another device

      Send_Socket (Chr(PROGRAM_COMMAND) + Chr(PROGRAM_CONTROLLER))        ‘ send command

      PROGRAMMING = True                         ‘ set busy programming flag
   Else
      MsgBox "error", vbExclamation, "Program Error"      ‘ display message if programming another device
   End If
End Sub
```

Figure 8.6 Indicating the remote programming processor to begin programming.

8.2.2.4 Progress indication

A Timer ActiveX control calls up a function every second to request the programming progress (only while programming) from the remote Web server. Once again it issues a command in the format of a packet with the first byte as 0xFF. The last byte will indicate to the Web server that the utility needs an indication of the programming progress.

```
Const PROGRAM_CHECK As Byte = &H0               ‘ request programming progress command

Private Sub Timer1_Timer()
   If PROGRAMMING = True Then                   ‘ only if busy programming a device
      Send_Socket (Chr(PROGRAM_COMMAND) + Chr(PROGRAM_CHECK))              ‘ send command
   End If
End Sub
```

Figure 8.7 Requesting the programming progress from the Web server.

The remote Web server also uses a 2-byte packet method to feed data back to the EmEther utility and the general structure looks as follows:

Programming progress (= 1) OR File transfer progress(= 0)	Data (1 byte)

Figure 8.8 Data feedback packet from the remote Web server.

The first byte indicates to the EmEther utility that it is a programming progress feedback packet or a file transfer progress feedback packet. The data byte either indicates the value of the Status lines (from the programming processor for programming progress indication) or it indicates the number of bytes of the transferred file that was programmed into the EEPROM.

Winsock1_DataArrival() is a Winsock control method that is called as soon as data arrives on the TCP socket connection and is used to capture feedback from the remote Web server. The Progress bar ActiveX control is used to indicate the programming progress.

```
Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
   Dim temp As String                                          ‘ temporary string

   Winsock1.GetData temp                                       ‘ read data from TCP socket

   Select Case (Asc(Mid(temp, 1, 1)))                          ‘ program feedback or file feedback
      Case 0:    TotalBytesSend = TotalBytesSend + (Asc(Mid(temp, 2, 1)) - 3)
                 progProgress.Position = (CInt((TotalBytesSend / TotalBytesToSend) * 100))
                 frmMain.Refresh                               ‘ update progress bar

      Case 1:    PROGRAMMING_STATUS = Asc(Mid(temp, 2, 1))     ‘ programming feedback
                 If (PROGRAMMING_STATUS = 0) Then
                    PROGRAMMING = False
                    progProgress.Position = 0
                    progProgress.ForeColor = vbBlue
                    txtStatus.Text = ""
                    txtStatus.Visible = False
                 Else
                    If PROGRAMMING_STATUS > 7 Then
                       progProgress.ForeColor = vbCyan
                    Else
                       progProgress.ForeColor = vbGreen
                    End If
                    Select Case (PROGRAMMING_STATUS)
                       Case 1: progProgress.Position = 4       ‘ File loading 4%
                       Case 2: progProgress.Position = 20      ‘ File loading 20%
                       Case 3: progProgress.Position = 36      ‘ File loading 36%
                       Case 4: progProgress.Position = 52      ‘ File loading 52%
                       Case 5: progProgress.Position = 68      ‘ File loading 68%
                       Case 6: progProgress.Position = 84      ‘ File loading 84%
                       Case 7: progProgress.Position = 100     ‘ File loading 100%
                       Case 8: progProgress.Position = 4       ‘ Programming device 4%
                       Case 9: progProgress.Position = 20      ‘ Programming device 20%
                       Case 10: progProgress.Position = 36     ‘ Programming device 36%
                       Case 11: progProgress.Position = 52     ‘ Programming device 52%
                       Case 12: progProgress.Position = 68     ‘ Programming device 68%
                       Case 13: progProgress.Position = 84     ‘ Programming device 84%
                       Case 14: progProgress.Position = 100    ‘ Programming device 100%
                    End Select
                 End If
   End Select
End Sub
```

Figure 8.9 Receiving data from the remote Web server using the TCP socket connection.

8.3 ACCESSING THE REMOTE SYSTEM

The remote system can be accessed using a standard Web browser, e.g. Internet Explorer 5.0. A permanent demonstration unit has been installed at PE Technikon and can be seen at Internet URL: picweb.petech.ac.za. Figure 8.10 shows a screen capture of one sample application for the remote system.

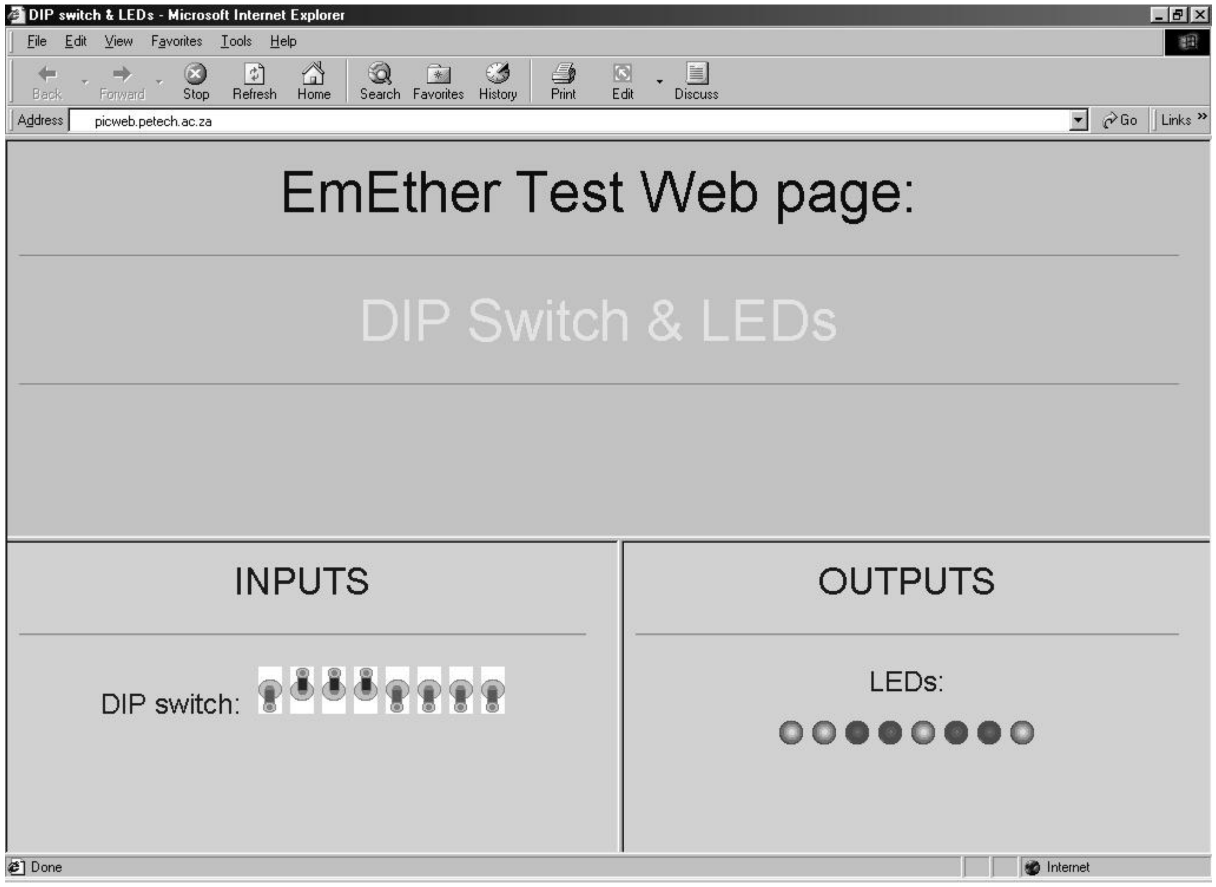
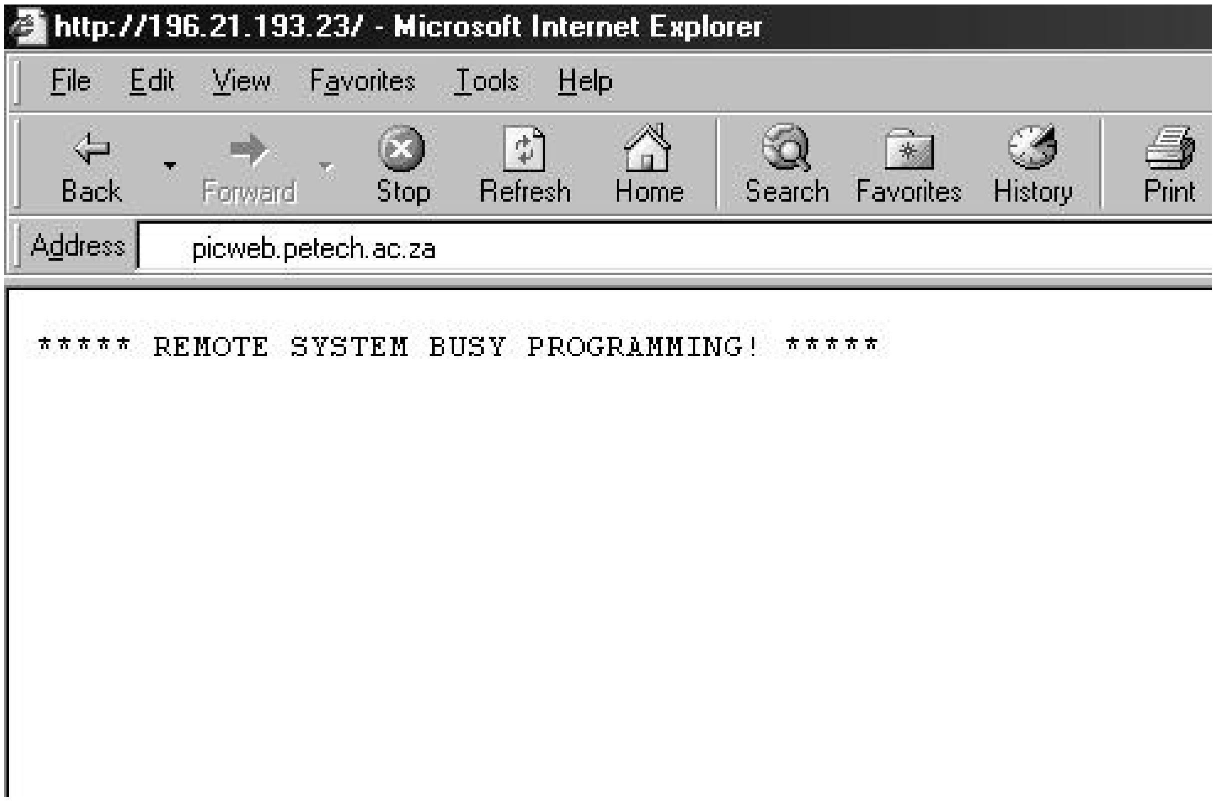

DIP switch & LEDs - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address picweb.petech.ac.za
Go Links
EmEther Test Web page:
DIP Switch & LEDs
INPUTS
DIP switch:
OUTPUTS
LEDs:
Done
Internet

Figure 8.10 An example of an embedded Web page.

Trying to access the Web page while the programming processor is busy with its programming duties, will result in a Web page containing the words: "REMOTE SYSTEM BUSY PROGRAMMING!" (Figure 8.11). This feature could be used to determine if the programming processor is finished programming the MAX7000S device, seeing that no progress information is available from the Jam Byte-code Player (as explained in section 8.2.1).


http://196.21.193.23/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Print
Address
picweb.petech.ac.za

```
***** REMOTE SYSTEM BUSY PROGRAMMING! *****
```

Figure 8.11 Error message when trying to access the remote system while it is in programming mode.

8.4 DEVELOPMENT SOFTWARE

The user will have to use some kind of software package for developing embedded code for the I/O controller (PIC16F877), developing logic designs for the MAX7000S device, and for creating new embedded web pages. Development packages that were used for testing purposes in this project is shown in Table 8.1.

Device	**Development software**	**Vendor**	**Website**	**Manual/Reference material**
PIC16F877	MPLAB	Microchip	www.microchip.com	MPLAB User's Guide
MAX7000S	MAX+PLUS II	Altera	www.altera.com	MAX+PLUS II: Getting Started.
Web pages	Frontpage	Microsoft	www.microsoft.com	Using HTML, JAVA and CGI

Table 8.1 Development packages and reference material used in this project.

Full working examples of applications for the remote system will be explained in the next chapter, and will give a clear indication of the relevant development software used for each section.

8.5 CONCLUSION

The main purpose of this chapter was to explain the working and operation of the EmEther utility for transferring and programming new source files.

The next chapter will demonstrate 3 sample applications used with the remote system, including the source code for the different sections. Programming times, access times and special comments will be provided with each example. The chapter will end with possible future developments for the remote system and the conclusion for the whole project.

CHAPTER 9
SYSTEM TESTING, FUTURE DEVELOPMENTS AND CONCLUSION

9.1 OVERVIEW

This final chapter will indicate the test results of the whole system by testing it with three different applications. Typical programming times will be given for each scenario. Possible future development areas will also be highlighted followed by the final conclusion of the project.

9.2 SYSTEM TESTING

To demonstrate the operation of the project, it was tested using three different applications. The software for the applications was developed and programmed from two different remote locations within PE Technikon's Intranet. Due to security risks, a firewall will block any attempts to program the board (using the EmEther utility) from a remote location outside the PE Technikon. Remote users can, however, still monitor and control the current programmed application through a standard Web browser (e.g. Internet Explorer 5) by establishing a connection to the board itself at http://picweb.petech.ac.za. All the information about this project, the code for the three different applications, the EmEther utility, and relevant datasheets can also be downloaded from the following URL: http://www.petech.ac.za/emether.

The three applications that were chosen are:

- A simulation of eight inputs and outputs using an 8-way DIP switch and LEDs.
- A temperature logger with a real-time clock.
- A Stepper motor, PWM, and push button simulation using an APPLIC-71 unit.

9.2.1 DIP SWITCH & LEDs

This simple application demonstrates how a user can monitor the status of an 8-way DIP switch and control the output of eight LEDs. Figure 9.1 shows the required external hardware and the connection to the remote Internet I/O controller board.

Figure 9.1 Required hardware for the DIP switch & LEDs application.

The LEDs are all connected as common-anode, which means that a logic LOW is necessary on the input for a LED to light up. Each switch in the DIP switch array is pulled up on the one side with a 10K resistor and on the other side it is connected to ground. This means that when a switch is open the output will be HIGH, and when the switch is closed the output will be LOW. Using Table 6.1 (Chapter 6, p.79), the connections to the I/O controller are as follows:

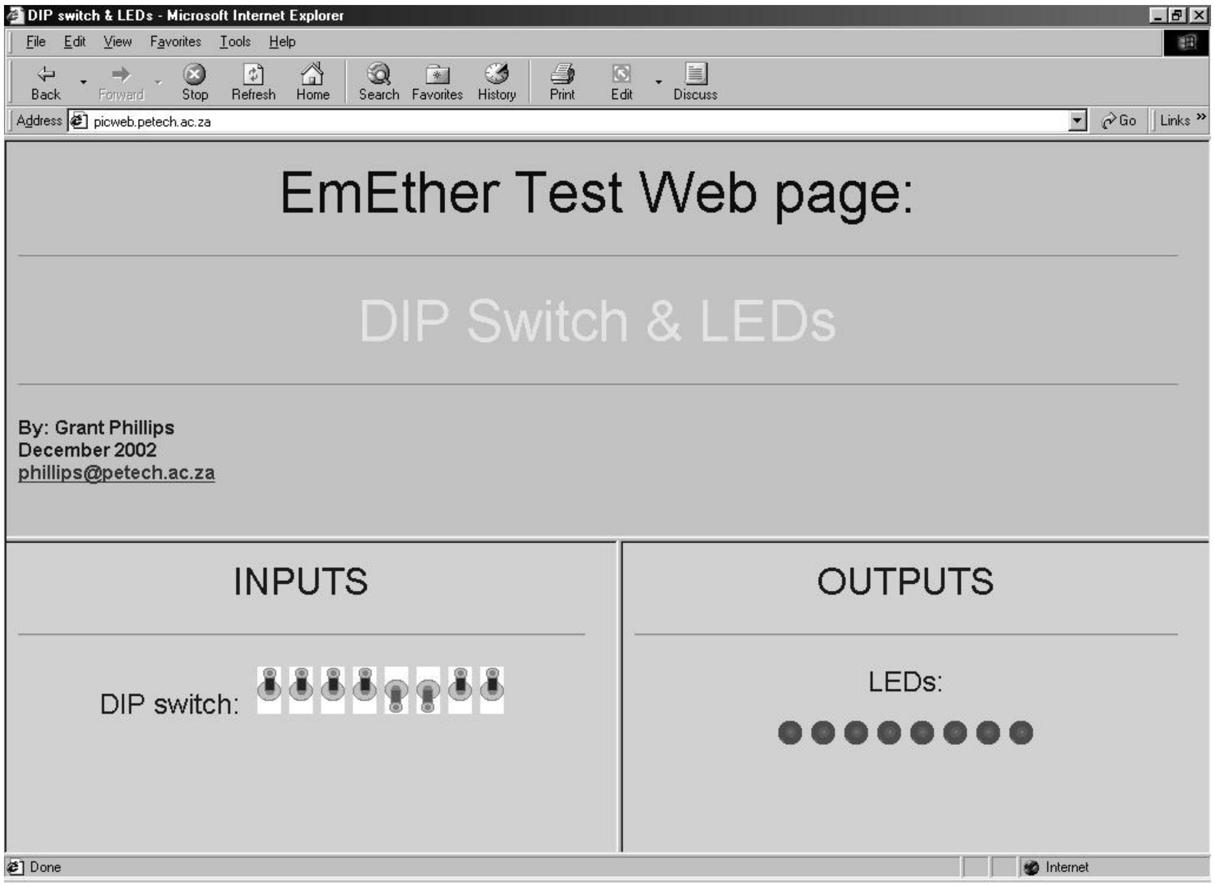
Description	**Connector J10 pin**	**I/O controller pin**
LED0	36	RD5
LED1	35	RD4
LED2	34	RD3
LED3	33	RD2
LED4	32	RD1
LED5	31	RD0
LED6	3	RB2
LED7	5	RB4
SWITCH0	10	RA0
SWITCH1	11	RA1
SWITCH2	12	RA2
SWITCH3	13	RA3
SWITCH4	14	RA4
SWITCH5	15	RA5
SWITCH6	1	RB0
SWITCH7	2	RB1

Table 9.1 I/O connections for the DIP switch & LEDs application.

Jumpers JP10 to JP14 must be in the 1-2 position, because port A on the I/O controller is configured as digital inputs and not as analog inputs.

9.2.1.1 Web page

Figure 9.2 shows the Web page for the DIP switch & LEDs application.


DIP switch & LEDs - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address picweb.petech.ac.za
Go Links

EmEther Test Web page:

DIP Switch & LEDs

By: Grant Phillips
December 2002
phillips@petech.ac.za

INPUTS

DIP switch:

OUTPUTS

LEDs:

Figure 9.2 Web page for the DIP switch & LEDs application.

The user can remotely monitor the status of the DIP switch and the LEDs. LEDs on the remote board can also be toggled ON or OFF by clicking on the required LED. The directory structure for the Web page is indicated in Figure 9.3.

```
⊞
├ dip.egi
├ index.htm
├ led0.gif
├ led1.gif
├ leds.egi
├ main.htm
├ sw0.gif
└ sw1.gif
```

Figure 9.3 Directory structure for the DIP switch & LEDs application.

The led_.gif and sw_.gif files are image files for indicating if a LED or switch is ON or OFF. The index.htm file basically divides the Web page into three different frames: the main title frame (main.htm), the inputs frame (dip.egi) and the outputs frame (leds.egi). A list of all the EGI characters for requesting the status of variables from the remote board is indicated in Table 9.2 (refer to section 4.4 for more detail about EGI characters).

Character	Function
0	Returns status of SWITCH0
1	Returns status of SWITCH1
2	Returns status of SWITCH2
3	Returns status of SWITCH3
4	Returns status of SWITCH4
5	Returns status of SWITCH5
6	Returns status of SWITCH6
7	Returns status of SWITCH7
a	Returns status of LED0
b	Returns status of LED1
c	Returns status of LED2
d	Returns status of LED3
e	Returns status of LED4
f	Returns status of LED5
g	Returns status of LED6
h	Returns status of LED7

Table 9.2 EGI request characters for the DIP switch & LEDs application.

The EGI output characters for controlling the LEDs remotely are shown in Table 9.3. Each of the LED images are push button controls with identifier names starting with a "Q" followed by one of the EGI output characters.

Figure 9.4 to 9.6 shows the HTML code for the main.htm, leds.egi and dip.egi files.

Character	Function
0	Toggles the output for LED0
1	Toggles the output for LED1
2	Toggles the output for LED2
3	Toggles the output for LED3
4	Toggles the output for LED4
5	Toggles the output for LED5
6	Toggles the output for LED6
7	Toggles the output for LED7

Table 9.3 EGI output characters for the DIP switch & LEDs application.

```
<head>
<title>DIP switch & LEDs</title>
</head>
<body bgcolor="#99CCFF">
<p align="center"><font face="Arial" size="7" color="#000080">EmEther Test Web
page:</font></p>
<hr>
<p align="center"><font face="Arial" size="7" color="#FFFF00">DIP Switch &amp;
LEDs</font></p>
<hr>
<p align="left"><b><font color="#0000FF">
<font face="Arial">By: Grant Phillips</font></font></b><br><b>
<font face="Arial" size="3"><font color="#0000FF">December 2002</font>
<br><a href="mailto:phillips@petech.ac.za">phillips@petech.ac.za</a></font></b>
</p>
```

Figure 9.4 HTML code for main.htm

```
<head>
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">OUTPUTS</font></p>
<hr>
<p align="center"><font color="#0000FF" face="Arial" size="5">LEDs:</font></p>
<form method="SUBMIT" action="leds.egi">
<p align="center"><input border="0" src="led@h.gif" name="Q7" width="20" height="20"
type="image">&nbsp;
<input border="0" src="led@g.gif" name="Q6" width="20" height="20" type="image">&nbsp;
<input border="0" src="led@f.gif" name="Q5" width="20" height="20" type="image">&nbsp;
<input border="0" src="led@e.gif" name="Q4" width="20" height="20" type="image">&nbsp;
<input border="0" src="led@d.gif" name="Q3" width="20" height="20" type="image">&nbsp;
<input border="0" src="led@c.gif" name="Q2" width="20" height="20" type="image">&nbsp;
<input border="0" src="led@a.gif" name="Q0" width="20" height="20" type="image"></p>
</form>
<p>&nbsp;</p>
```

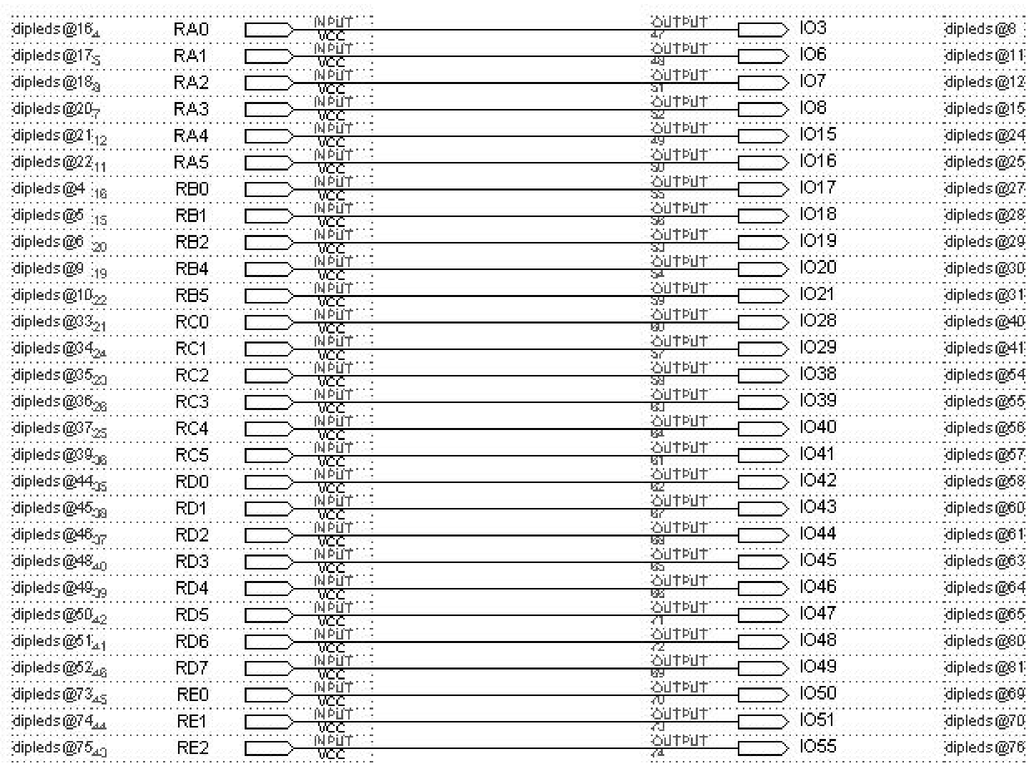
Figure 9.5 HTML code for leds.egi

```
<head><meta http-equiv="refresh" content="3">
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">INPUTS</font></p>
<hr>
<p align="center"><font size="5"><font color="#0000FF"><font face="Arial">DIP
switch:&nbsp; <img border="0" src="sw@7.gif" width="20" height="40"> <img border="0"
src="sw@6.gif" width="20" height="40">
<img border="0" src="sw@5.gif" width="20" height="40"> <img border="0" src="sw@4.gif"
width="20" height="40">
<img border="0" src="sw@3.gif" width="20" height="40"> <img border="0" src="sw@2.gif"
width="20" height="40">
<img border="0" src="sw@1.gif" width="20" height="40"> <img border="0" src="sw@0.gif"
width="20" height="40">
</font></font></font></p>
<p>&nbsp;</p>
```

Figure 9.6 HTML code for dip.egi

9.2.1.2 Code for MAX7000S

The reconfigurable hardware is not needed in this application since the I/O controller will drive and monitor the LEDs and DIP switch directly. All the pins on the MAX7000S that are connected to the I/O controller will therefore be programmed as inputs to prevent contention (described in section 6.5). Figure 9.7 shows the graphic representation for this configuration on the MAX7000S.



dipleds@16 4 RA0 INPUT VCC OUTPUT 47 IO3 dipleds@8
dipleds@17 5 RA1 INPUT VCC OUTPUT 48 IO6 dipleds@11
dipleds@18 3 RA2 INPUT VCC OUTPUT 51 IO7 dipleds@12
dipleds@20 7 RA3 INPUT VCC OUTPUT 52 IO8 dipleds@15
dipleds@21 12 RA4 INPUT VCC OUTPUT 49 IO15 dipleds@24
dipleds@22 11 RA5 INPUT VCC OUTPUT 50 IO16 dipleds@25
dipleds@4 16 RB0 INPUT VCC OUTPUT 55 IO17 dipleds@27
dipleds@5 15 RB1 INPUT VCC OUTPUT 56 IO18 dipleds@28
dipleds@6 20 RB2 INPUT VCC OUTPUT 53 IO19 dipleds@29
dipleds@9 19 RB4 INPUT VCC OUTPUT 54 IO20 dipleds@30
dipleds@10 22 RB5 INPUT VCC OUTPUT 59 IO21 dipleds@31
dipleds@33 21 RC0 INPUT VCC OUTPUT 60 IO28 dipleds@40
dipleds@34 24 RC1 INPUT VCC OUTPUT 57 IO29 dipleds@41
dipleds@35 23 RC2 INPUT VCC OUTPUT 58 IO38 dipleds@54
dipleds@36 28 RC3 INPUT VCC OUTPUT 63 IO39 dipleds@55
dipleds@37 25 RC4 INPUT VCC OUTPUT 64 IO40 dipleds@56
dipleds@39 36 RC5 INPUT VCC OUTPUT 61 IO41 dipleds@57
dipleds@44 35 RD0 INPUT VCC OUTPUT 62 IO42 dipleds@58
dipleds@45 38 RD1 INPUT VCC OUTPUT 67 IO43 dipleds@60
dipleds@46 37 RD2 INPUT VCC OUTPUT 68 IO44 dipleds@61
dipleds@48 40 RD3 INPUT VCC OUTPUT 65 IO45 dipleds@63
dipleds@49 39 RD4 INPUT VCC OUTPUT 66 IO46 dipleds@64
dipleds@50 42 RD5 INPUT VCC OUTPUT 71 IO47 dipleds@65
dipleds@51 41 RD6 INPUT VCC OUTPUT 72 IO48 dipleds@80
dipleds@52 46 RD7 INPUT VCC OUTPUT 69 IO49 dipleds@81
dipleds@73 45 RE0 INPUT VCC OUTPUT 70 IO50 dipleds@69
dipleds@74 44 RE1 INPUT VCC OUTPUT 73 IO51 dipleds@70
dipleds@75 43 RE2 INPUT VCC OUTPUT 74 IO55 dipleds@76

Figure 9.7 MAX7000S graphic entry for the DIP switch & LEDs application.

9.2.1.3 Code for I/O controller

Figure 9.8 shows the source code for the I/O controller. The PIC16F877 simply drives its output pin LOW to turn a LED ON and HIGH to turn it OFF. The interrupt routine is the most important section of the program and handles all the variable requests.

```
#include <16F877.H>
#device *=16

// ********************** I/O Ports **********************
#byte   PORTA            = 0x05
#byte   PORTB            = 0x06
#byte   PORTC            = 0x07
#byte   PORTD            = 0x08
#byte   PORTE            = 0x09

short LED0 = 0;                              // Status of LEDs
short LED1 = 0;
short LED2 = 0;
short LED3 = 0;
short LED4 = 0;
short LED5 = 0;
short LED6 = 0;
short LED7 = 0;

void out_leds(void)                          // Updates LEDs
{
   output_bit(PIN_D5, ~LED0);                // LEDs are connected common anode
   output_bit(PIN_D4, ~LED1);
   output_bit(PIN_D3, ~LED2);
   output_bit(PIN_D2, ~LED3);
   output_bit(PIN_D1, ~LED4);
   output_bit(PIN_D0, ~LED5);
   output_bit(PIN_B2, ~LED6);
   output_bit(PIN_B4, ~LED7);
}

// ******************** Required Code ********************
#use DELAY(CLOCK=20000000)                                   // setup crystal frequency
#use RS232 (BAUD=115200, XMIT=PIN_C6, RCV=PIN_C7, ERRORS)  // setup RS-232 module
#define STOP   putchar('~')                  // STOP character for protocol

#int_rda                                     // interrupt routine for receiving a command / request
void serial_isr(void)
{
   char b;

   disable_interrupts(global);
   b = getch();                              // read character describing a request or command
   if (b == '?')                             // request
   {
      b = getch();                           // read request number
      switch(b)
      {
```

Figure 9.8 I/O controller source code for the DIP switch & LEDs application.

```
case '0': {                               // SWITCH0
             if (input(PIN_A0))           // if input is HIGH, the switch is OFF (pulled up by R2)
                printf("0");              // use printf() for reply
             else
                printf("1");
             STOP;                        // send STOP character after data has been send
          }
          break;
case '1': {                               // SWITCH1
             if (input(PIN_A1))
                printf("0");
             else
                printf("1");
             STOP;
          }
          break;
case '2': {                               // SWITCH2
             if (input(PIN_A2))
                printf("0");
             else
                printf("1");
             STOP;
          }
          break;
case '3': {                               // SWITCH3
             if (input(PIN_A3))
                printf("0");
             else
                printf("1");
             STOP;
          }
          break;
case '4': {                               // SWITCH4
             if (input(PIN_A4))
                printf("0");
             else
                printf("1");
             STOP;
          }
          break;
case '5': {                               // SWITCH5
             if (input(PIN_A5))
                printf("0");
             else
                printf("1");
             STOP;
          }
          break;
case '6': {                               // SWITCH6
             if (input(PIN_B0))
                printf("0");
             else
                printf("1");
             STOP;
          }
          break;
```

Figure 9.8 continued.

```
    case '7': {                                   // SWITCH7
                  if (input(PIN_B1))
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
    case 'a': {                                   // LED0
                  if (~LED0)                      // if variable is HIGH, the LED is ON
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
    case 'b': {                                   // LED1
                  if (~LED1)
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
    case 'c': {                                   // LED2
                  if (~LED2)
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
    case 'd': {                                   // LED3
                  if (~LED3)
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
    case 'e': {                                   // LED4
                  if (~LED4)
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
    case 'f':  {                                  // LED5
                  if (~LED5)
                     printf("0");
                  else
                     printf("1");
                  STOP;
               }
               break;
```
Figure 9.8 continued.

```
        case 'g': {                                  // LED6
                    if (~LED6)
                       printf("0");
                    else
                       printf("1");
                    STOP;
                 }
                 break;
        case 'h': {                                  // LED7
                    if (~LED7)
                       printf("0");
                    else
                       printf("1");
                    STOP;
                 }
                 break;
        deault:  {                                   // request not recognized, then reply with an ‘X’
                    printf("X");
                    STOP;
                 }
                 break;
    }
}

if (b == '!')                                        // command
{
    b = getch();                                     // read command number
    switch(b)
        {
        case '0': {                                  // toggle LED0
                     if (LED0)                       // if variable is HIGH, turn the LED OFF
                       LED0 = 0;
                    else
                       LED0 = 1;
                 }
                 break;
        case '1': {                                  // toggle LED1
                    if (LED1)
                       LED1 = 0;
                    else
                       LED1 = 1;
                 }
                 break;
        case '2': {                                  // toggle LED2
                    if (LED2)
                       LED2 = 0;
                    else
                       LED2 = 1;
                 }
                 break;
        case '3': {                                  // toggle LED3
                    if (LED3)
                       LED3 = 0;
                    else
                       LED3 = 1;
                 }
                 break;
```

Figure 9.8 continued.

```
         case '4': {                                     // toggle LED4
                     if (LED4)
                        LED4 = 0;
                     else
                        LED4 = 1;
                  }
                  break;
         case '5': {                                     // toggle LED5
                     if (LED5)
                        LED5 = 0;
                     else
                        LED5 = 1;
                  }
                  break;
         case '6': {                                     // toggle LED6
                     if (LED6)
                        LED6 = 0;
                     else
                        LED6 = 1;
                  }
                  break;
         case '7': {                                     // toggle LED7
                     if (LED7)
                        LED7 = 0;
                     else
                        LED7 = 1;
                  }
                  break;
      }
   }

   enable_interrupts(global);
}
// ********************************************************


void main(void)                                          // main program loop
{
// ******************** Required Code  ***************
   set_tris_c(0x80);                                     // setup RXD and TXD pins for RS-232
   enable_interrupts(global);
   enable_interrupts(int_rda);                           // enable interrupt on RS-232 receive
// ********************************************************
   set_tris_a(0xff);                                     // set port direction registers
   set_tris_b(0xeb);
   set_tris_d(0x00);
   set_tris_e(0xff);
   portd = 255;
   portb = 255;
   porta = 0;
   porte = 0;

   while(1)
   {
      out_leds();                                        // continuously update the output of the LEDs
   }
}
```

Figure 9.8 continued.

9.2.1.4 Results

Table 9.4 indicates typical times that were measured to transfer the different source code files to the remote Internet I/O controller board and to program it in real-time.

File	Size	Transfer	Program
dipleds.jbc	20933 bytes	8.16 sec	11 min 50.51 sec
dipleds.hex	3650 bytes	0.78 sec	1 min 3.18 sec
Web page	5727 bytes	———	2.41 sec

Table 9.4 Typical loading and programming times for the DIP switch & LEDs application.

9.2.2 TEMPERATURE LOGGER

This application demonstrates how a user can monitor the temperature at a remote location. The system logs the temperature readings every hour and the user can then view the temperature readings of the last twelve hours. Figure 9.9 shows the required external hardware and the connection to the remote Internet I/O controller board.

A DS1305 is used as the real time clock (RTC). It is a SPI device and therefore only needs four lines to access the internal registers of the RTC. The user will be able to set the time using controls on the web page.

A LM35 temperature sensor is used to measure the temperature. It provides an analog output voltage of 10mV/°C and is connected to the ADC0 pin, which in turn is connected to pin RA0 of the I/O controller. Jumper JP10 should be in the 1-3 position because this pin is used as an analog input and not as a digital input.

The two 7-segment LED displays provide feedback of the measured temperature at the remote location itself. They are connected as common-cathode, which means that a logic HIGH is necessary on the input for a LED segment to light up. The displays are driven from the MAX7000S and are not directly connected to the I/O controller.

Figure 9.9   Required hardware for the temperature logger application.

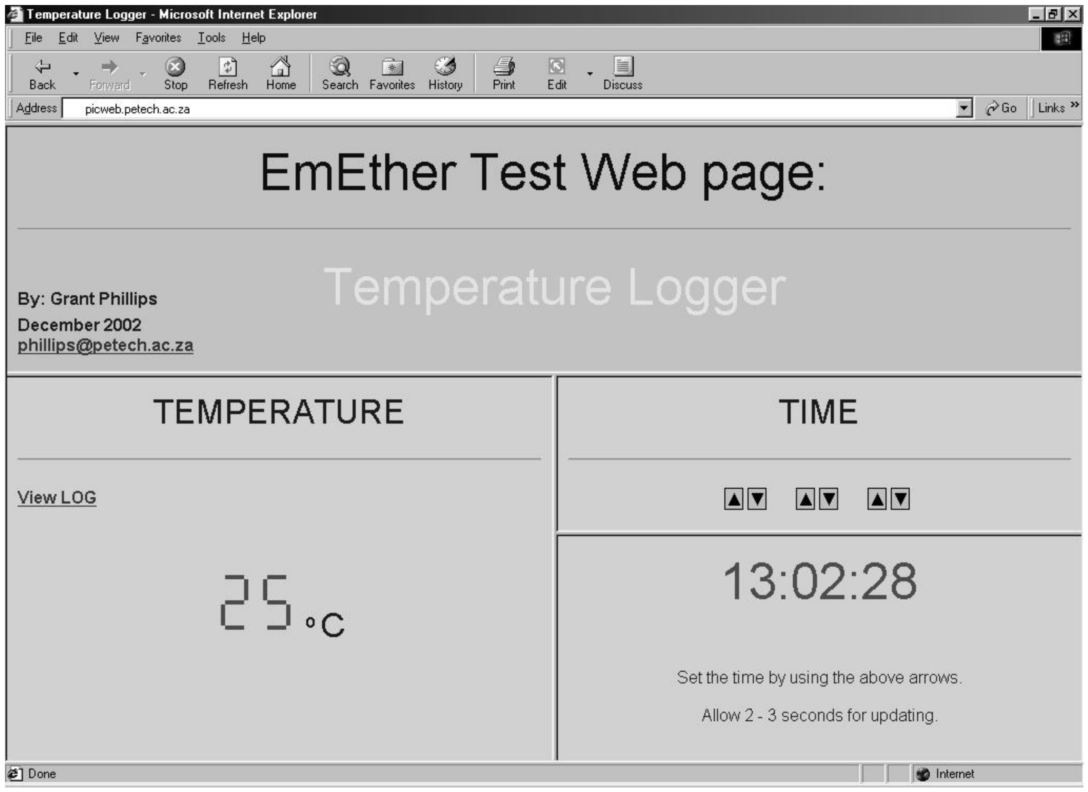
The connections to the I/O controller and the MAX7000S are as follows:

Description	Connector J11 pin	I/O controller pin	MAX7000S pin
Segment A (TENS)	5		56
Segment B (TENS)	6		57
Segment C (TENS)	13		67
Segment D (TENS)	12		65
Segment E (TENS)	11		64
Segment F (TENS)	4		55
Segment G (TENS)	3		54
Segment DOT (TENS)	14		68
Segment A (UNITS)	9		61
Segment B (UNITS)	10		63
Segment C (UNITS)	20		76
Segment D (UNITS)	16		70
Segment E (UNITS)	15		69
Segment F (UNITS)	8		60
Segment G (UNITS)	7		58
Segment DOT (UNITS)	21		77
CE (DS1305)	22		79
SDO (DS1305)	19	RE0	
SDI (DS1305)	18	RE1	
CLK (DS1305)	17	RE2	

Table 9.5 I/O connections for the temperature logger application.

9.2.2.1 Web page

Figure 9.10 shows the Web page for the temperature logger application.


Temperature Logger - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address picweb.petech.ac.za
Go Links
Done
Internet

EmEther Test Web page:

Temperature Logger

By: Grant Phillips
December 2002
phillips@petech.ac.za

TEMPERATURE

View LOG

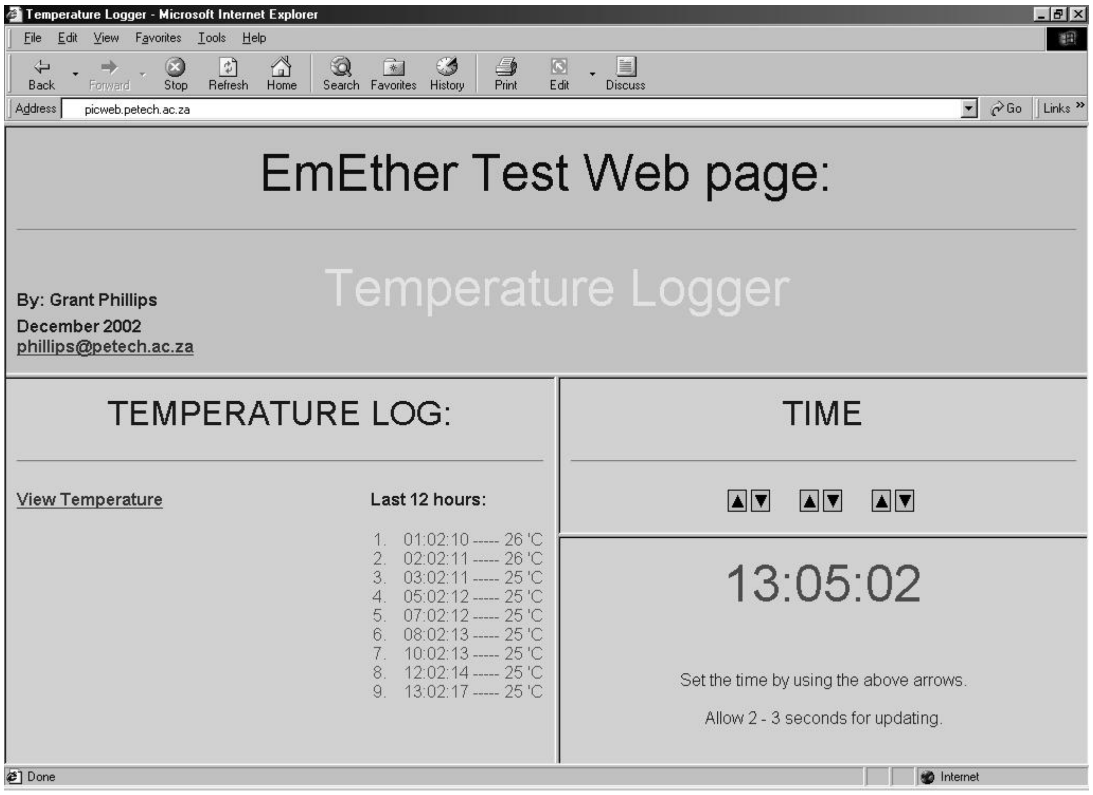
25 °C

TIME

13:02:28

Set the time by using the above arrows.

Allow 2 - 3 seconds for updating.


Temperature Logger - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print Edit Discuss
Address picweb.petech.ac.za
Go Links
Done
Internet

EmEther Test Web page:

Temperature Logger

By: Grant Phillips
December 2002
phillips@petech.ac.za

TEMPERATURE LOG:

View Temperature

Last 12 hours:

```
1.  01:02:10 ----- 26 'C
2.  02:02:11 ----- 26 'C
3.  03:02:11 ----- 25 'C
4.  05:02:12 ----- 25 'C
5.  07:02:12 ----- 25 'C
6.  08:02:13 ----- 25 'C
7.  10:02:13 ----- 25 'C
8.  12:02:14 ----- 25 'C
9.  13:02:17 ----- 25 'C
```

TIME

13:05:02

Set the time by using the above arrows.

Allow 2 - 3 seconds for updating.

Figure 9.10 Web page for the temperature logger application.

The user can remotely monitor the temperature measured by the LM35. The RTC time can be set using the up and down arrows on the Web page. Users can also view the logged temperatures of the last twelve hours. The directory structure for the Web page is indicated in Figure 9.11.

```
⊞
├─ 7seg_0.gif
├─ 7seg_1.gif
├─ 7seg_2.gif
├─ 7seg_3.gif
├─ 7seg_4.gif
├─ 7seg_5.gif
├─ 7seg_6.gif
├─ 7seg_7.gif
├─ 7seg_8.gif
├─ 7seg_9.gif
├─ down.gif
├─ index.htm
├─ log.egi
├─ main.htm
├─ temp.egi
├─ time.egi
├─ up.gif
└─ updown.egi
```

Figure 9.11 Directory structure for the temperature logger application.

The 7seg__.gif files are image files for indicating the temperature as it would be displayed on the 7-segment displays on the remote Internet I/O controller board. The up.gif and down.gif files are also image files for showing the arrows to set the time of the RTC. The index.htm file basically divides the Web page into four different frames: the main title frame (main.htm), the temperature display frame (temp.egi), the time display frame (time.egi) and the time set frame (updown.egi). If a user needs to view the temperature log, the temperature display frame will change to the log display frame (log.egi). A list of all the EGI characters for requesting the status of variables from the remote board is indicated in Table 9.6.

Character	Function
0	Returns status of the TENS 7-segment display
1	Returns status of the UNITS 7-segment display
2	Returns the current time from the RTC (hour:min:sec)
3	Returns the log entries as one string

Table 9.6 EGI request characters for the temperature logger application.

The EGI output characters for setting the time of the RTC remotely are shown in Table 9.7. Each of the arrow images are push button controls with identifier names starting with a “Q” followed by one of the EGI output characters. Allow two to three seconds after clicking on an arrow for the change to take effect.

Character	Function
0	Decrement seconds
1	Increment seconds
2	Decrement minutes
3	Increment minutes
4	Decrement hours
5	Increment hours

Table 9.7 EGI output characters for the temperature logger application.

Figure 9.12 to 9.16 shows the HTML code for the main.htm, temp.egi, time.egi, updown.egi and log.egi files.

```
<head>
<title>Temperature Logger</title>
</head>
<body bgcolor="#99CCFF">
<p align="center"><font face="Arial" size="7" color="#000080">EmEther Test Web
page:</font></p>
<hr>
<p align="left"><b><font color="#0000FF"><font face="Arial">By: Grant
Phillips&nbsp;</font>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</font></b>
<font face="Arial" size="7"
color="#FFFF00">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
Temperature Logger</font><br><b><font face="Arial" size="3"><font
color="#0000FF">December 2002</font>
<br><a href="mailto:phillips@petech.ac.za">phillips@petech.ac.za</a></font></b></p>
```

Figure 9.12 HTML code for main.htm

```
<head><meta http-equiv="refresh" content="3">
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">TEMPERATURE</font></p>
<hr>
<p align="left"><font face="Arial"><b><a href="log.egi">View LOG</a></b></font></p>
<p align="center">&nbsp;</p>
<p align="center"><img border="0" src="7seg_@0.gif" width="40" height="66">
<font face="Arial" size="5" color="#FF0000"><img border="0" src="7seg_@1.gif" width="40"
height="66">
</font><font color="#0000FF"><sup><font face="Arial" size="5">o </font> </sup>
<font color="#0000FF" face="Arial" size="6">C</font></font></p>
```

Figure 9.13 HTML code for temp.egi

```
<head><meta http-equiv="refresh" content="1">
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" color="#FF0000" size="7">@2</font></p>
<p align="center">&nbsp;</p>
<p align="center" style="line-height: 100%"><font face="Arial" size="3"
color="#0000FF">Set
the time by using</font> <font face="Arial" size="3" color="#0000FF">the above
arrows.</font></p>
<p align="center" style="line-height: 100%"><font face="Arial" size="3"
color="#0000FF">Allow
2 - 3 seconds for updating.</font></p>
```

Figure 9.14 HTML code for time.egi

```
<head>
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">TIME</font></p>
<hr>
<form method="SUBMIT" action="updown.egi">
<p align="center">
<input border="0" src="up.gif" name="Q5" width="18" height="21" type="image">
<input border="0" src="down.gif" name="Q4" width="18" height="21"
type="image">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<input border="0" src="up.gif" name="Q3" width="18" height="21" type="image">
<input border="0" src="down.gif" name="Q2" width="18" height="21"
type="image">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<input border="0" src="up.gif" name="Q1" width="18" height="21" type="image">
<input border="0" src="down.gif" name="Q0" width="18" height="21" type="image">&nbsp;</p>
</form>
```

Figure 9.15 HTML code for updown.egi

```
<head><meta http-equiv="refresh" content="10">
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">TEMPERATURE LOG:</font></p>
<hr>
<p align="left"><font face="Arial" color="#0000FF"><b><a href="temp.egi">View
Temperature</a>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
Last 12 hours:</b></font></p>
<p align="right"><font face="Arial" color="#008000">@3</font></p>
<p align="right">&nbsp;</p>
```

Figure 9.16 HTML code for log.egi

9.2.2.2 Code for MAX7000S

The purpose of the MAX7000S in this application is to drive the two 7-segment displays. If the displays were driven directly from the I/O controller, it will use 16 pins on the PIC16F877 (8 pins per 7-segment display). To minimize the number of pins needed, two 7-segment drivers (7449) are programmed into the MAX7000S which drives the two displays. The two 7-segment drivers are fed from a binary-to-double BCD driver. In this application the LM35 will rarely measure temperatures beyond 50°C and therefore the binary-to-double BCD driver only needs six inputs from the I/O controller ($2^6 = 64$).

Pin 79 on the MAX7000S (IO57) is used as the chip enable (CE) pin for the DS1305 and is internally connected to pin RA5 of the I/O controller. Once again all unused pins of the MAX7000S must be programmed as inputs.

Figure 9.17 shows the graphic representation for this configuration on the MAX7000S.

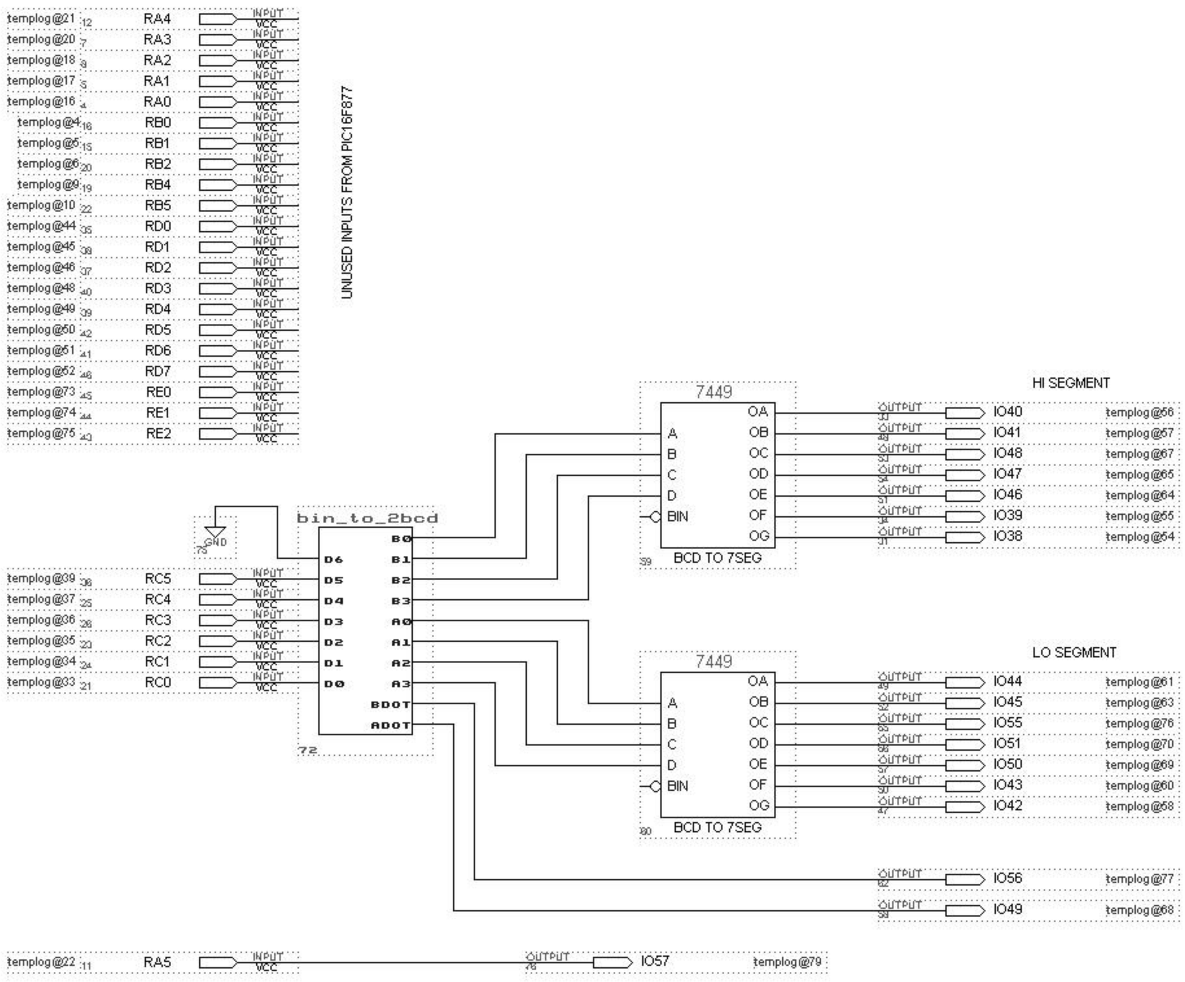


Figure 9.17 MAX7000S graphic entry for the temperature logger application.

9.2.2.3 Code for I/O controller

Figure 9.18 shows the source code for the I/O controller. The PIC16F877 uses its analog to digital converter to read the temperature measurement from the LM35. The I/O controller also uses port C to drive the MAX7000S which in turn drives the two 7-segment displays. Port E is used to access the SPI bus to communicate with the DS1305 RTC.

```
#include <16F877.H>
#use DELAY(CLOCK=20000000)
#include "ds1305.c"

// ********************** I/O Ports **********************
#byte   PORTA           = 0x05
#byte   PORTB           = 0x06
#byte   PORTC           = 0x07
#byte   PORTD           = 0x08
#byte   PORTE           = 0x09

long temp=0;                                        // current temperature
byte DAY, MONTH, YEAR, HOUR, MIN, SEC;              // current time
byte LOG_HOUR[12], LOG_MIN[12], LOG_SEC[12];        // time arrays for logging
long LOG_TEMP[12];                                  // temperature array for logging
signed int LOG_ENTRIES=-1;                          // current amount of logged temperatures

// ******************** Required Code ********************
#use DELAY(CLOCK=20000000)                                     // setup crystal frequency
#use RS232 (BAUD=115200, XMIT=PIN_C6, RCV=PIN_C7, ERRORS)  // setup RS-232 module
#define STOP   putchar('~')                     // STOP character for protocol

#int_rda                                        // interrupt routine for receiving a command / request
void serial_isr(void)
{
   char b;
   disable_interrupts(global);
   b = getch();                                 // read character describing a request or command
   if (b == '?')                                // request
   {
      b = getch();                              // read request number
      switch(b)
      {
         case '0': {
                      printf("%lu", temp/10);   // returns status of TENS 7-segment display
                      STOP;
                   }
                   break;
         case '1': {
                      printf("%lu", temp%10);   // returns status of UNITS 7-segment display
                      STOP;
                   }
                   break;
         case '2': {
                      printf("%X:%X:%X", HOUR, MIN, SEC);        // returns current time
                      STOP;
                   }
                   break;
```

Figure 9.18 I/O controller source code for the temperature logger application.

```
        case '3': {
                    if (LOG_ENTRIES < 0)                        // no log entries
                      printf("  ");
                    else
                      for (i=0; i <= LOG_ENTRIES; i++)
                      {
                        printf("%d.&nbsp;&nbsp;&nbsp;&nbsp;%X:%X:%X      -----      %lu   'C<br>",   i+1,
LOG_HOUR[i], LOG_MIN[i], LOG_SEC[i], LOG_TEMP[i]);          // displays log as several lines of text
                      }
                    STOP;
                  }
                  break;
        deault:   {
                    printf("X");                  // request not recognized, then reply with an 'X'
                    STOP;
                  }
                  break;
      }
   }

   if (b == '!')                                  // command
   {
      b = getch();                                // read command number
      switch(b)
      {
        case '0': {                               // decrement seconds
                    if ((SEC == 0x50) || (SEC == 0x40) || (SEC == 0x30) || (SEC == 0x20) || (SEC == 0x10))
                      SEC = SEC - 7;
                    else if (SEC == 0x00)
                      SEC = 0x59;
                    else
                      SEC--;
                    RTC_Set_DateTime(0,0,0,HOUR,MIN,SEC);
                  }
                  break;
        case '1': {                               // increment seconds
                    if ((SEC == 0x09) || (SEC == 0x19) || (SEC == 0x29) || (SEC == 0x39) || (SEC == 0x49))
                      SEC = SEC + 7;
                    else if (SEC == 0x59)
                      SEC = 0x00;
                    else
                      SEC++;
                    RTC_Set_DateTime(0,0,0,HOUR,MIN,SEC);
                  }
                  break;
        case '2': {                               // decrement minutes
                    if ((MIN == 0x50) || (MIN == 0x40) || (MIN == 0x30) || (MIN == 0x20) || (MIN == 0x10))
                      MIN = MIN - 7;
                    else if (MIN == 0x00)
                      MIN = 0x59;
                    else
                      MIN--;
                    RTC_Set_DateTime(0,0,0,HOUR,MIN,SEC);
                  }
                  break;
```

Figure 9.18 continued.

```
        case '3': {                                   // increment minutes
                    if ((MIN == 0x09) || (MIN == 0x19) || (MIN == 0x29) || (MIN == 0x39) || (MIN == 0x49))
                       MIN = MIN + 7;
                    else if (MIN == 0x59)
                       MIN = 0x00;
                    else
                       MIN++;
                    RTC_Set_DateTime(0,0,0,HOUR,MIN,SEC);
                 }
                 break;
        case '4': {                                   // decrement hours
                    if ((HOUR == 0x20) || (HOUR == 0x10))
                       HOUR = HOUR - 7;
                    else if (HOUR == 0x00)
                       HOUR = 0x23;
                    else
                       HOUR--;
                    RTC_Set_DateTime(0,0,0,HOUR,MIN,SEC);
                 }
                 break;
        case '5': {                                   // increment hours
                    if ((HOUR == 0x09) || (HOUR == 0x19))
                       HOUR = HOUR + 7;
                    else if (HOUR == 0x23)
                       HOUR = 0x00;
                    else
                       HOUR++;
                    RTC_Set_DateTime(0,0,0,HOUR,MIN,SEC);
                 }
                 break;
      }
   }

   enable_interrupts(global);
}

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

void main(void)                                     // main program loop
{
   int i;
   long a=0, sum=0;
   byte LAST_LOG=0;

// ******************** Required Code  ***************
   set_tris_c(0x80);                                // setup RXD and TXD pins for RS-232
   enable_interrupts(global);
   enable_interrupts(int_rda);                      // enable interrupt on RS-232 receive
// *******************************************************
   setup_adc_ports(RA0_ANALOG);                     // setup port A and ADC
   setup_adc(ADC_CLOCK_INTERNAL);
   set_adc_channel(0);
   set_tris_a(0x01);                                // set port direction registers
   set_tris_b(0x00);
   set_tris_d(0x00);
   set_tris_e(0x01);
```

Figure 9.18 continued.

```
    portc = 0;
    portd = 255;
    portb = 0;
    porta = 0;
    porte = 255;

    RTC_Init();                                // initialise DS1305
    RTC_Set_DateTime(0,0,0,0,0,0);             // reset time on DS1305

    while(1)
    {
       // ********** Get Temperature from LM35 on ADC channel 0 **********
       for (i=0; i<200; i++)
       {
          delay_ms(10);                        // allow to setle
          a = READ_ADC() * 500 / 1024;         // get value
          sum = sum + a;                       // add to sum so that average can be calculated
       }
       temp = sum / 200;                       // calculate average
       portc = temp;                           // drive MAX7000S with the binary value of the temperature
       sum = 0;                                // reset sum

       // ********** Get time from RTC **********
       RTC_Get_Time(HOUR, MIN, SEC);           // read time from DS1305

       // ********** Read sample every 1 hour **********
       if (HOUR - LAST_LOG >= 1)               // if an hour expired, log the temperature
       {
          LOG_ENTRIES++;
          if (LOG_ENTRIES > 11)
             LOG_ENTRIES = 0;
          LAST_LOG = HOUR;
          LOG_HOUR[LOG_ENTRIES] = HOUR;              // store temperature and time in the log arrays
          LOG_MIN[LOG_ENTRIES] = MIN;
          LOG_SEC[LOG_ENTRIES] = SEC;
          LOG_TEMP[LOG_ENTRIES] = TEMP;
       }
    }
}
```

Figure 9.18 continued.

9.2.2.4 Results

Table 9.8 indicates typical times that were measured to transfer the different source code files to the remote Internet I/O controller board and to program it in real-time.

File	Size	Transfer	Program
templog.jbc	21978 bytes	8.54 sec	12 min 11.98 sec
templog.hex	10077 bytes	1.55 sec	1 min 2.80 sec
Web page	16674 bytes	———	2.41 sec

Table 9.8 Typical loading and programming times for the temperature logger application.

9.2.3 APPLIC-71 UNIT

This application is based on the APPLIC-71 unit from Scientific Educational Systems (SES). It is used as an educational tool to teach students about stepper motors, DC motors, switches etc. The unit consists of various peripherals, but for this application only the switches, the stepper motor, the lamp, the temperature sensor and the potentiometer will be used. Figure 9.19 shows the required external hardware and the connection to the remote Internet I/O controller board.

A LM35 temperature sensor is used to measure the temperature. It provides an analog output voltage and is connected to the ADC1 pin, which in turn is connected to pin RA1 of the I/O controller. The potentiometer is used to set the speed of the stepper motor and is connected to the ADC0 pin (pin RA0 on the I/O controller). Jumpers JP10 and JP11 should be in the 1-3 position seeing that these pins are configured as an analog inputs and not as a digital inputs.

A LED is also connected as common anode and needs a LOW on the input to turn it ON. All the switches are connected to ground on the one side and pulled up by the PIC16F877's internal pull-up resistors on the other side. If a switch is closed, the input to the I/O controller will be LOW.

The inputs to the stepper motor's four windings first go through a ULN2803 octal driver to allow the stepper motor to be driven from a +12V supply and to provide the necessary current. PWM is demonstrated by using the lamp, which is driven by the ULN2803. By changing the PWM duty cycle, the lamp will be dimmer or brighter.

Figure 9.19   Required hardware for the APPLIC-71 application.

Using Table 6.1 (Chapter 6, p.79), the connections to the I/O controller are as follows:

Description	**Connector J10 pin**	**Connector J11 pin**	**I/O controller pin**
Switch	3		RB2
Pushbutton	2		RB1
Limit switch	1		RB0
LED	5		RB5
S1 (stepper motor)	25		RC5
S2 (stepper motor)	26		RC4
S3 (stepper motor)	27		RC3
S4 (stepper motor)	28		RC2
Lamp	24		RC1
LM35		30	RA1
Potentiometer		29	RA0

Table 9.9 I/O connections for the APPLIC-71 application.

9.2.3.1 Web page

Figure 9.20 shows the Web page for the APPLIC-71 application.

The user can remotely monitor the temperature measured by the LM35. By using the arrow controls, the brightness of the lamp can be changed. As the lamp gets brighter, the temperature will also increase seeing that the LM35 is positioned right next to the lamp.

The stepper motor can be started or stopped by clicking on the GO/STOP image and clicking on the arrow image can change the direction. As the stepper motor turns, the position of the shaft is indicated in degrees. The potentiometer on the APPLIC-71 unit sets the speed of the stepper motor. The lower the speed indicator on the Web page, the faster the stepper motor rotates.

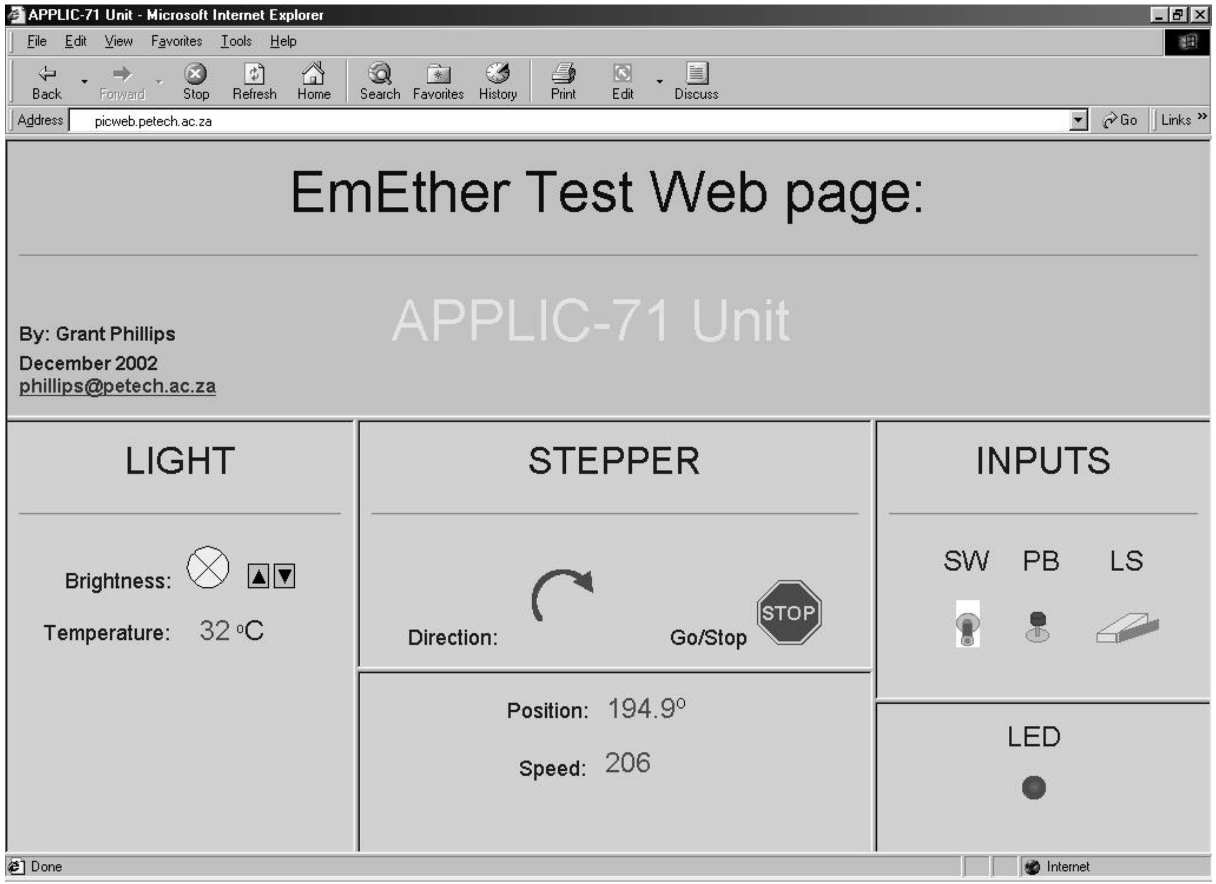


Figure 9.20 Web page for the APPLIC-71 application.

The directory structure for the Web page is indicated in Figure 9.21.

The light_.gif files are image files for indicating the brightness of the lamp. The ls_.gif, sw_.gif, pb_.gif and led_.gif image files just indicates the status of the switches and the LED. The index.htm file basically divides the Web page into six different frames: the main title frame (main.htm), the lamp display frame (light.egi), the stepper control frame (stepper.egi), the stepper position frame (position.egi), the inputs frame (inputs.egi) and the LED frame (led.egi).

The EGI output characters for controlling the stepper motor, the lamp brightness and the LED are shown in Table 9.10. The arrow images, the LED image and the stepper motor control images are push button controls with identifier names starting with a “Q” followed by one of the EGI output characters.

```
⊞
├─ ccw.gif
├─ cw.gif
├─ down.gif
├─ go.gif
├─ index.htm
├─ inputs.egi
├─ led.egi
├─ led0.gif
├─ led1.gif
├─ light.egi
├─ light0.gif
├─ light1.gif
├─ light2.gif
├─ light3.gif
├─ light4.gif
├─ ls0.gif
├─ ls1.gif
├─ main,htm
├─ pb0.gif
├─ pb1.gif
├─ postion.egi
├─ stepper.egi
├─ stop.gif
├─ sw0.gif
├─ sw1.gif
└─ up.gif
```

Figure 9.21 Directory structure for APPLIC-71 application.

Character	**Function**
0	Toggles LED ON or OFF
1	Increases the lamp brightness
2	Decreases the lamp brightness
3	Changes stepper motor direction
4	Toggles stepper motor ON or OFF

Table 9.10 EGI output characters for the APPLIC-71 application.

A list of all the EGI characters for requesting the status of variables from the remote board is indicated in Table 9.11.

Character	Function
0	Returns status of the potentiometer (speed of the stepper)
1	Returns the temperature measured by the LM35
2	Returns the status of the limit switch
3	Returns the status of the push button
4	Returns the status of the switch
5	Returns the status of the LED
6	Returns the brightness of the lamp
7	Returns the current stepper motor direction
8	Returns the status of stepper motor
9	Returns the position of the stepper motor’s shaft
a	Returns the speed of the stepper motor

Table 9.11 EGI request characters for the APPLIC-71 application.

Figure 9.22 to 9.27 shows the HTML code for the main.htm, light.egi, stepper.egi, position.egi, inputs.egi and led.egi files.

```
<head>
<title>DIP switch & LEDs</title>
</head>
<body bgcolor="#99CCFF">
<p align="center"><font face="Arial" size="7" color="#000080">EmEther Test Web
page:</font></p>
<hr>
<p align="left"><b><font color="#0000FF"><font face="Arial">By: Grant Phillips&nbsp;
</font>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</font></b>
<font face="Arial" size="7" color="#FFFF00">
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
APPLIC-71 Unit</font><br><b><font face="Arial" size="3">
<font color="#0000FF">December 2002</font>
<br><a href="mailto:phillips@petech.ac.za">phillips@petech.ac.za</a></font></b></p>
```

Figure 9.22 HTML code for main.htm

```
<html>
<head><meta http-equiv="refresh" content="10">
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">LIGHT</font></p>
<hr>
<form method="SUBMIT" action="light.egi">
<p align="center"><font face="Arial" size="4" color="#0000FF">Brightness:
</font>&nbsp; <img border="0" src="light@6.gif" width="36" height="36">&nbsp;&nbsp;&nbsp;
<input border="0" src="up.gif" name="Q1" width="18" height="21" type="image">
<input border="0" src="down.gif" name="Q2" width="18" height="21" type="image"></p>
</form>
<p align="left"><font face="Arial" color="#0000FF" size="4">&nbsp;&nbsp;&nbsp;
Temperature:
&nbsp;&nbsp;&nbsp;&nbsp;
</font><font face="Arial" color="#FF0000" size="5">@1</font><font face="Arial" size="4"
color="#FF0000">
</font><font color="#0000FF"><font face="Arial" size="3"><sup>o</sup></font>
<font face="Arial" size="5">C</font></font></p>
</body>
</html>
```

Figure 9.23   HTML code for light.egi

```
<html>
<head>
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">STEPPER</font></p>
<hr>
<form method="SUBMIT" action="stepper.egi">
<p align="center"><font color="#0000FF" face="Arial" size="4">Direction:
&nbsp;&nbsp;</font><input border="0" src="@7.gif" name="Q3" width="92" height="84"
type="image">
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<font color="#0000FF" face="Arial" size="4">Go/Stop</font>&nbsp;
<input border="0" src="@8.gif" name="Q4" width="55" height="55" type="image">
</p>
</form>
</body>
</html>
```

Figure 9.24   HTML code for stepper.egi

```
<html><meta http-equiv="refresh" content="1">
<title></title>
<body bgcolor="#CCCCFF">
<p align="left"><font color="#0000FF" face="Arial" size="4">
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
Position:&nbsp;&nbsp; </font><font face="Arial" color="#FF0000" size="5">@9</font>
<sup><font color="#0000FF" face="Arial" size="4">o</font></sup></p>
<p align="left"><font face="Arial" color="#0000FF" size="5"><sup>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</sup><sup>&nbsp;&nbsp;&nbsp;</sup><sup>&nbsp;&nbsp;&
nbsp;
</sup><sup>Speed:</sup></font><sup><font face="Arial" size="4"
color="#0000FF">&nbsp;&nbsp;&nbsp;
<b> </b></font></sup><sup><font face="Arial" size="4" color="#0000FF"><b>
</b></font></sup>
<font face="Arial" color="#FF0000" size="6"><sup>
@a</sup></font></p>
</body>
</html>
```

Figure 9.25 HTML code for position.egi

```
<head><meta http-equiv="refresh" content="2">
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font face="Arial" size="6" color="#0000FF">INPUTS</font></p>
<hr>
<p align="center"><font color="#0000FF" face="Arial" size="5">&nbsp;&nbsp;&nbsp;
SW&nbsp;&nbsp;&nbsp; PB&nbsp;&nbsp;&nbsp; &nbsp;
LS&nbsp;&nbsp;&nbsp;&nbsp;</font></p>
<form method="SUBMIT" action="inputs.egi">
<p
align="center">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<img border="0" src="sw@4.gif" width="20"
height="40">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<img border="0" src="pb@3.gif" width="27"
height="35">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<img border="0" src="ls@2.gif" width="60"
height="32">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
</p>
</form>
</body>
```

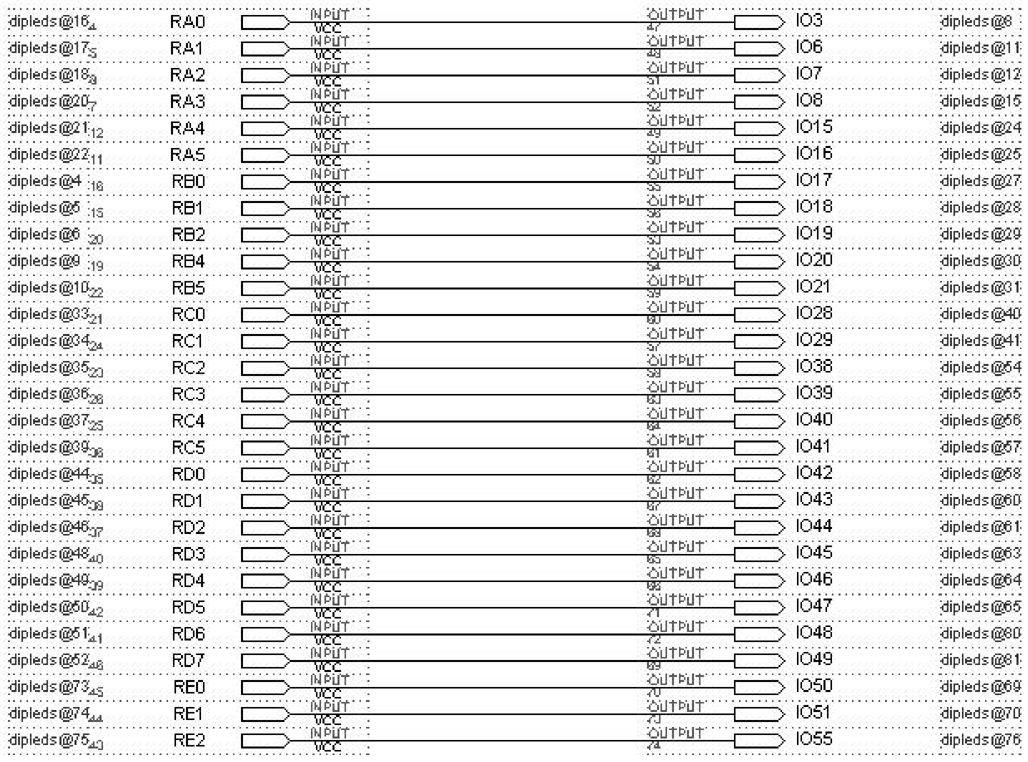
Figure 9.26 HTML code for inputs.egi

```
<head>
<title></title>
</head>
<body bgcolor="#CCCCFF">
<p align="center"><font color="#0000FF" face="Arial" size="5">LED</font></p>
<form method="SUBMIT" action="led.egi">
<p align="center">
<input border="0" src="led@5.gif" name="Q0" width="20" height="20" type="image">
</p>
</form>
</body>
```

Figure 9.27 HTML code for led.egi

9.2.3.2 Code for MAX7000S

As with the DIP switch & LEDs application, the reconfigurable hardware is not needed in this application. All unused pins on the MAX7000S that are connected to the I/O controller will be programmed as inputs. Figure 9.28 shows the graphic representation for this configuration on the MAX7000S. The same source file as for the DIP switch & LEDs application (dipleds.jbc) can be used to program the MAX7000S.



dipleds@16 RA0 INPUT VCC OUTPUT IO3 dipleds@8
dipleds@17 RA1 INPUT VCC OUTPUT IO6 dipleds@11
dipleds@18 RA2 INPUT VCC OUTPUT IO7 dipleds@12
dipleds@20 RA3 INPUT VCC OUTPUT IO8 dipleds@15
dipleds@21 RA4 INPUT VCC OUTPUT IO15 dipleds@24
dipleds@22 RA5 INPUT VCC OUTPUT IO16 dipleds@25
dipleds@4 RB0 INPUT VCC OUTPUT IO17 dipleds@27
dipleds@5 RB1 INPUT VCC OUTPUT IO18 dipleds@28
dipleds@6 RB2 INPUT VCC OUTPUT IO19 dipleds@29
dipleds@9 RB4 INPUT VCC OUTPUT IO20 dipleds@30
dipleds@10 RB5 INPUT VCC OUTPUT IO21 dipleds@31
dipleds@33 RC0 INPUT VCC OUTPUT IO28 dipleds@40
dipleds@34 RC1 INPUT VCC OUTPUT IO29 dipleds@41
dipleds@35 RC2 INPUT VCC OUTPUT IO38 dipleds@54
dipleds@36 RC3 INPUT VCC OUTPUT IO39 dipleds@55
dipleds@37 RC4 INPUT VCC OUTPUT IO40 dipleds@56
dipleds@39 RC5 INPUT VCC OUTPUT IO41 dipleds@57
dipleds@44 RD0 INPUT VCC OUTPUT IO42 dipleds@58
dipleds@45 RD1 INPUT VCC OUTPUT IO43 dipleds@60
dipleds@46 RD2 INPUT VCC OUTPUT IO44 dipleds@61
dipleds@48 RD3 INPUT VCC OUTPUT IO45 dipleds@63
dipleds@49 RD4 INPUT VCC OUTPUT IO46 dipleds@64
dipleds@50 RD5 INPUT VCC OUTPUT IO47 dipleds@65
dipleds@51 RD6 INPUT VCC OUTPUT IO48 dipleds@80
dipleds@52 RD7 INPUT VCC OUTPUT IO49 dipleds@81
dipleds@73 RE0 INPUT VCC OUTPUT IO50 dipleds@69
dipleds@74 RE1 INPUT VCC OUTPUT IO51 dipleds@70
dipleds@75 RE2 INPUT VCC OUTPUT IO55 dipleds@76

Figure 9.28 MAX7000S graphic entry for the APPLIC-71 application.

9.2.3.3 Code for I/O controller

Figure 9.29 shows the source code for the I/O controller. The PIC16F877 uses its analog to digital converter to measure the temperature and the potentiometer.

```
#include <16F877.H>
#use DELAY(CLOCK=20000000)

// ********************** I/O Ports **********************
#byte   PORTA           = 0x05
#byte   PORTB           = 0x06
#byte   PORTC           = 0x07
#byte   PORTD           = 0x08
#byte   PORTE           = 0x09

#define CW              1
#define CCW             0

long TEMP, POT, BRIGHTNESS = 0, STEPPER_SPEED=0;
short STEPPER_ON = 0;
short LED_ON = 0;
short STEP_DIR = CW;
float STEPPER_POS=0;
byte STEPPER_STATE = 0;
byte const STEPPER_POSITIONS[4] =  {0b0101,                 // stepper motor sequence
                                    0b1001,
                                    0b1010,
                                    0b0110};
void read_analogs(void);                                    // function prototypes
void Light(long BRIGHTNESS);
void LED(short VAL);
short SW(void);
short PB(void);
short LS(void);
void Stepper(short DIR, byte SPEED);


// ******************** Required Code ********************
#use DELAY(CLOCK=20000000)                                  // setup crystal frequency
#use RS232 (BAUD=115200, XMIT=PIN_C6, RCV=PIN_C7, ERRORS)   // setup RS-232 module
#define STOP   putchar('~')                  // STOP character for protocol

#int_rda                                     // interrupt routine for receiving a command / request
void serial_isr(void)
{
   char b;
   disable_interrupts(global);
   b = getch();                              // read character describing a request or command
   if (b == '?')                             // request
   {
      b = getch();                           // read request number
      switch(b)
      {
         case '0': {
                     printf("%lu", POT);     // returns analog value of potentiometer
                    STOP;
                   }
                   break;
         case '1': {
                    printf("%lu", TEMP);     // returns temperature
                    STOP;
                   }
                   break;
```
Figure 9.29   I/O controller source code for the APPLIC-71 application.

```
        case '2': {
                if (LS())                       // returns status of limit switch
                    printf("1");
                else
                    printf("0");
                STOP;
            }
            break;
        case '3': {
                if (PB())                       // returns status of push button
                    printf("1");
                else
                    printf("0");
                STOP;
            }
            break;
        case '4': {
                if (SW())                       // returns status of switch
                    printf("1");
                else
                    printf("0");
                STOP;
            }
            break;
        case '5': {
                if (LED_ON)                     // returns status of LED
                    printf("1");
                else
                    printf("0");
                STOP;
            }
            break;
        case '6': {
                 printf("%lu", BRIGHTNESS/255);      // returns brightness of lamp
                 STOP;
            }
            break;
        case '7': {
                if (STEP_DIR == CW)             // returns stepper motor direction; CW = 1; CCW = 0
                    printf("cw");
                else
                    printf("ccw");
                STOP;
            }
            break;
        case '8': {
                if (STEPPER_ON == 1)            // returns status of stepper motor
                    printf("stop");
                else
                    printf("go");
                STOP;
            }
            break;
        case '9': {
                printf("%5.1f", STEPPER_POS);       // returns stepper motor's shaft position
                STOP;
            }
            break;
```

Figure 9.29 continued.

```
         case 'a': {
                      printf("%lu", STEPPER_SPEED);      // returns stepper motor's speed
                      STOP;
                   }
                   break;
         deault:   {
                      printf("X");                   // request not recognized, then reply with an 'X'
                      STOP;
                   }
                   break;
      }
   }

   if (b == '!')                                     // command
   {
      b = getch();                                   // read command number
      switch(b)
      {
         case '0': {                                 // toggles LED
                      if (LED_ON == 1)
                         LED(0);
                      else
                         LED(1);
                   }
                   break;
         case '1': {
                      if (BRIGHTNESS < 1020)                      // increases brightness of lamp
                         BRIGHTNESS = BRIGHTNESS + 255;
                   }
                   break;
         case '2': {
                      if (BRIGHTNESS > 0)                         // decreases brightness of lamp
                         BRIGHTNESS = BRIGHTNESS - 255;
                   }
                   break;
         case '3': {
                      if (STEP_DIR == CW)                         // toggles stepper motor direction
                         STEP_DIR = CCW;
                      else
                         STEP_DIR = CW;
                   }
                   break;
         case '4': {
                      if (STEPPER_ON == 1)                        // toggles stepper motor ON or OFF
                         STEPPER_ON = 0;
                      else
                         STEPPER_ON = 1;
                   }
                   break;
      }
   }
   enable_interrupts(global);
}

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

Figure 9.29 continued.

```
void main(void)                              // main program loop
{

// ******************** Required Code  ***************
   set_tris_c(0x80);                         // setup RXD and TXD pins for RS-232
   enable_interrupts(global);
   enable_interrupts(int_rda);               // enable interrupt on RS-232 receive
// ******************************************************

   setup_adc_ports(ALL_ANALOG);              // setup port A and ADC
   setup_adc(ADC_CLOCK_INTERNAL);

   set_tris_a(0xFF);                         // set port direction registers
   set_tris_b(0xEF);
   set_tris_d(0xFF);
   set_tris_e(0xFF);
   port_b_pullups(TRUE);                     // enable internal pull-up resistors for switches
   setup_ccp2(CCP_PWM);                      // set PWM module
   setup_timer_2(T2_DIV_BY_1, 255, 2);
   portc = 0;
   portd = 0;
   portb = 0;
   porta = 0;
   porte = 0;
   LED(0);

   while(1)
   {
      read_analogs();                        // read temperature and potentiometer
      light(BRIGHTNESS);                     // set the brightness of the lamp
      STEPPER_SPEED = ((1024-POT)/5) + 7;    // stepper motor speed calculation according to pot
      stepper(STEP_DIR, STEPPER_SPEED);      // set stepper motor direction and speed
   }
}


// ***************** Function definitions ***************

void read_analogs(void)
{
   delay_us(20);
   set_adc_channel(0);                       // read temperature on ADC channel 0 (pin RA0)
   delay_us(20);
   POT = read_adc();

   delay_us(20);
   set_adc_channel(1);                       // read temperature on ADC channel 1 (pin RA1)
   delay_us(20);
   TEMP = read_adc();
}


void light(long BRIGHTNESS)
{
   set_pwm2_duty(BRIGHTNESS);                // set PWM duty cycle for brightness of lamp
}
```

Figure 9.29 continued.

```
void LED(short VAL)                            // toggles LED ON or OFF
{
   if (VAL == 1)
   {
      OUTPUT_LOW(PIN_B4);
      LED_ON = 1;
   }
      else
   {
      OUTPUT_HIGH(PIN_B4);
      LED_ON = 0;
   }
}

short SW(void)                                 // returns status of switch
{
   if (INPUT(PIN_B2) == 0)
      return 1;
   else
      return 0;
}

short PB(void)                                 // returns status of push button
{
   if (INPUT(PIN_B1) == 0)
      return 1;
   else
      return 0;
}

short LS(void)                                 // returns status of limit switch
{
   if (INPUT(PIN_B0) == 0)
      return 1;
   else
      return 0;
}

void Stepper(short DIR, byte SPEED)            // sets stepper motor direction and speed
{
   if (STEPPER_ON)
   {
      delay_ms(SPEED);

      if (DIR == 1)
      {
         if (STEPPER_STATE == 3)
            STEPPER_STATE = 0;
         else
            STEPPER_STATE++;                   // go to next stepper motor sequence
         if (STEPPER_POS >= 352.5)
            STEPPER_POS = 0;
         else
            STEPPER_POS = STEPPER_POS + 7.5;      // 7.5 degrees per step
      }
      else
```

Figure 9.29 continued.

```
      {
         if (STEPPER_STATE == 0)
            STEPPER_STATE = 3;
         else
            STEPPER_STATE--;                         // go to previous stepper motor sequence
         if (STEPPER_POS <= 0)
            STEPPER_POS = 352.5;
         else
            STEPPER_POS = STEPPER_POS - 7.5;
      }
      output_bit(PIN_C5, bit_test(STEPPER_POSITIONS[STEPPER_STATE], 0));    // set stepper motor
      output_bit(PIN_C4, bit_test(STEPPER_POSITIONS[STEPPER_STATE], 1));    // inputs according to
      output_bit(PIN_C3, bit_test(STEPPER_POSITIONS[STEPPER_STATE], 2));    // current sequence
      output_bit(PIN_C2, bit_test(STEPPER_POSITIONS[STEPPER_STATE], 3));    // number
   }
}
```

Figure 9.29 continued.

9.2.3.4 Results

Table 9.12 indicates typical times that were measured to transfer the different source code files to the remote Internet I/O controller board and to program it in real-time.

File	Size	Transfer	Program
dipleds.jbc	20933 bytes	8.16 sec	11 min 50.51 sec
applic71.hex	10130 bytes	1.65 sec	2 min 50.04 sec
Web page	24134 bytes	———	9.42 sec

Table 9.12 Typical loading and programming times for the APPLIC-71 application.

9.3 CURRENT LIMITATIONS

Unfortunately the project had some limitations. The main reason for these limitations is the lack of memory (RAM) of the Ethernet interface controller (PIC16F877) which only has 368 bytes of RAM. Various methods had to be performed to overcome this RAM issue, as explained in the previous chapters, but unfortunately not all of the limitations could be conquered. The current limitations are:

- The maximum file size that can be accessed from the remote Internet I/O controller board is 1.5 Kbytes. It is not a real problem, as long as the Web page does not include large images. The main idea of this remote system is to monitor and control applications, and therefore no fancy images and controls are really necessary.

- Only lower case and 8.3 format DOS file names are supported. This is a problem if small JAVA applets need to be executed on the remote system, because JAVA applets are case sensitive and not in the 8.3 format.
- The PIC16F877 has a program memory of 8K words, of which 7K is used by the TCP/IP protocol stack and the supporting embedded code. The ideal situation would be if the I/O controller and the Ethernet interface controller could be one controller, but unfortunately the PIC16F877's program memory does not allow this.
- Only the HTTP upper layer protocol is supported
- The system does not support dynamic IP address allocation through a dynamic host configuration protocol (DHCP) server and needs a dedicated IP address within the LAN.

9.4 FUTURE DEVELOPMENTS

The main two areas of the project that can be improved in the future are the speed and the program memory size of the controllers. Here are some of the proposed future developments or improvements:

- A faster programming processor. Currently an 8051 derivative is used and runs at 12Mhz. It might be worth looking at the new Dallas range of microcontrollers. They are also based on the very popular 8051, but runs at higher clock frequencies (up to 50Mhz).
- Combine the Ethernet interface controller and the I/O controller in one microcontroller. This will only be possible if the microcontroller has a large program memory area. Microchip's 18FXXX series has larger program memory than the 16FXXX series and might be worth looking at. Another approach would be to use the latest 8051 based network microcontroller from Dallas (DS80C400) that has its own built-in NIC.
- Combine the Ethernet interface controller and I/O controller and program them into the CPLD. This will reduce the component count considerably.
- Support more upper level protocols such as FTP and SMTP.
- Allow for larger files that can be accessed with long file name support. This will automatically make it easier to implement JAVA applets.

- Improve the file system's data access time. Currently I^2C EEPROMs are used which are relatively slow. At the time the design was done there were no 64K SPI EEPROMs available which would improve the situation significantly.

9.5 CONCLUSION

The project has shown how a normal 8-bit microcontroller (PIC16F877 in this case) can be used to establish a Web server and with the help of an I/O controller then to monitor and control remote processes. Through the use reconfigurable hardware (CPLD) the process can be reconfigured and reprogrammed from a remote location within minutes.

The aim of this research was to provide methods and ideas for combining embedded Internet and reconfigurable hardware in one project to accomplish total reconfigurability of a remote embedded Internet system.

APPENDIX A
REFERENCES

1. Altera (2001). Embedded programming using the 8051 & Jam byte-code. San Jose: Altera Corporation.

2. Altera (1999). In-system programmability guidelines. San Jose: Altera Corporation.

3. Altera (2000). In-system programmability in MAX devices. San Jose: Altera Corporation.

4. Altera (1999). Introduction to ISP. San Jose: Altera Corporation.

5. Altera (2001). MAX+PLUS II: Getting Started. San Jose: Altera Corporation.

6. Altera (2000). MAX7000 Programmable Logic Device Family. San Jose: Altera Corporation.

7. Arnett, F., Dulaney, E., Harper, E. (1995). Inside TCP/IP. Indianapolis: New Riders Publishing.

8. Bentham, J. (2000). TCP/IP lean: Web servers for embedded systems. Lawrence: CMP Books.

9. Blackwell, N. (1995). Handbook of data communications. Cambridge: Blackwell Publishers.

10. Bolton, M. (1990). Digital Systems Design with Programmable Logic. Cornwall: T.J. Press.

11. Bostock, G. (1996). FPGAs and Programmable LSI: A designer's handbook. Boston: Butterworth-Heinemann.

12. Carter, J. (1997). Digital Designing with Programmable Logic Devices. New Jersey: Prentice-Hall Inc.

13. Ladd, E. & O'Donnell, J. (1996). Using HTML, JAVA and CGI. Indianapolis: Que Corporation.

14. Microchip (2001). 24LC256 Datasheet. Microchip Technology Inc.

15. Microchip (2000). In-circuit serial programming for PIC16F8XX FLASH MCUs. Microchip Technology Inc.

16. Microchip (2001). In-circuit serial programming guide. Microchip Technology Inc.

17. Microchip (2000). MPLAB User's Guide. Microchip Technology Inc.

18. Microchip (2001). PIC16F87X data sheet. Microchip Technology Inc.

19. Microchip (2001). PICDEM.net User's guide. Microchip Technology Inc.

20. Socolofsky, T., Kale, C. (2001). "A TCP/IP Tutorial". Article available from Internet URL http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1180.html.

21. Stanek, W. (1996). Web publishing unleashed (1st Edition). Indianapolis: Sams.net Publishing.

22. Wilder, F. (1999). A guide to the TCP/IP protocol suite. London: Artech House.