IST BASIC RESEARCH PROJECT
SHARED COST RTD PROJECT
THEME: FET DISAPPEARING COMPUTER
COMMISSION OF THE EUROPEAN COMMUNITIES
DIRECTORATE GENERAL INFSO
PROJECT OFFICER: THOMAS SKORDAS

Global Smart Spaces

D12: CONSTRUCTION GUIDELINES: HARDWARE INFRASTRUCTURE DESIGN

ANDREW JAMIESON, DUNCAN SMEED, PADDY NIXON

IST Project Number	IST-2000-26070	**Acronym**	GLOSS
Full title	Global Smart Spaces		
EU Project officer	Thomas Skordas		

Deliverable	**Number**	D12	**Name**	Construction Guidelines: Hardware infrastructure Design		
Task	**Number**	T	**Name**			
Work Package	**Number**	WP	**Name**			
Date of delivery	**Contractual**			**Actual**		
Code name				**Version** 1.0 draft ☐ final ☑		
Nature	Prototype ☐ Report ☑ Specification ☐ Tool ☐ Other:					
Distribution Type	Public ☑ Restricted ☐ to: <partners>					
Authors (Partner)	University of Strathclyde					
Contact Person	Paddy Nixon					
	Email	Paddy.Nixon@cis.strath.ac.uk	**Phone**	+44 141 548 3588	**Fax**	+44 552 5330
Abstract (for dissemination)	In this document we describe the development of the hardware sensing infrastructure for GLOSS. We use a Motorola HCS12 to act as a sensor to network gateway and develop a bespoke board and micro web server to support this. We demonstrate the development process and guidelines for building such as device.					
Keywords						

1 INTRODUCTION

In this document we describe the development of the hardware-sensing infrastructure for GLOSS. We use a Motorola HCS12 to act as a sensor to network gateway and develop a bespoke board and micro webserver to support this.

1.1 THE HCS12 MICROCONTROLLER OVERVIEW

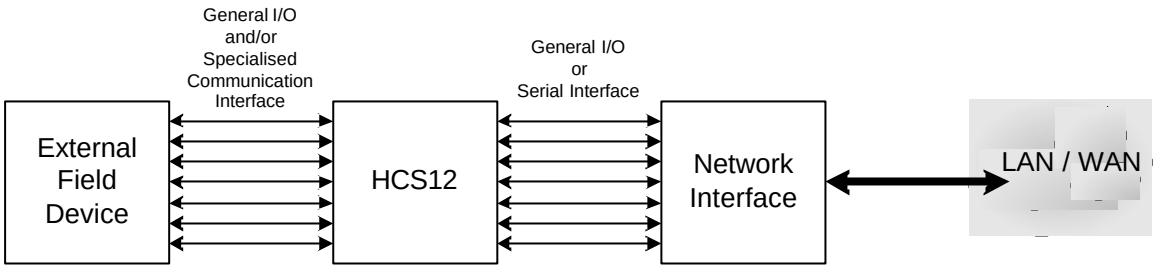
The Motorola HCS12 microcontroller is an industry leading 16bit device, supporting many superior functions over its competitors. The actual device that is being used for the basis of this project is a MC9S12DP256, which has the following main on-chip features

- 12K RAM
- 256K Reprogrammable Flash ROM
- 4K EEPROM
- Serial Communications Interface (x2)
- Serial Peripheral Interface (x3)
- Analogue to Digital Converters (x2)
- Controller Area Network (CAN) device (x5)
- Pulse Width Modulator (8bit)
- Various system timers

Having a fixed network interface to a controller of this sort can open up many possibilities in terms of control, monitoring and remote access. The device has a rich set of interface and communication options – most of the external pins associated with the devices mentioned above can also be configured for general I/O use via the on chip Port Integration Module.

1.2 INTERCONNECTED (GATEWAY) HCS12

We use propose that the HCS12 acts as a communication gateway between an external device and the network interface. The actual possibilities which arise with this configuration are endless, as devices can be connected both via specialised communication interfaces provided by the HCS12, and also via general I/O. For example, a piece of machinery in a large factory could be connected via the specialised Control Area Network (CAN) interface on the HCS12. With a web interface on the HCS12, a user could monitor and communicate with CAN based machinery from the comfort of an office that could be on the same site, or the other side of the world. Having a serial interface available on the HCS12 also opens up the possibility of communicating with many legacy devices that only support basic serial connectivity.



This report will go on to focus on the implementation of some specific networking hardware and the design of the related software. Some testing (both functional and performance) and conclusions.

1.3 MAIN REQUIREMENTS

For any device to participate as a node on a local area network or the Internet, there are a few main issues that must be considered.

1.3.1 PROTOCOL STACK

The system must be able to support a subset of the Internet Protocol (IP) suite of communications protocols. As a minimum, we must be able to check if the device is actually part of the network. This is commonly performed by the use of a network "ping", and is handled by a protocol called ICMP (Internet Control Message Protocol). ICMP is carried by IP, and supports a variety of different message types for error reporting, control and distribution of status information. The only ICMP message we will require to support is the "Echo Reply" message – the ability to respond to a network ping directed to us.

To be able to actually connect to and interact with the device, another protocol must be supported. Transmission Control Protocol (TCP) is a connection-orientated service that is designed for use over unreliable connectionless networks (again carried by IP). TCP builds in the reliability that is required to successfully transmit data over an unknown communication channel, and is the backbone protocol used in most Internet services.

1.3.2 APPLICATION SOFTWARE

Having the ability to test if our node is present, and the ability to negotiate a TCP connection is a start, but the system will also require some application software that can interact with the stack and use the communication facilities it provides.

For direct, interactive communication, there are really only two candidates in mainstream use today on the Internet – the slightly antiquated "Telnet" and the hugely popular Hyper Text Transfer Protocol (HTTP). Telnet is an application that comes bundled with most network aware operating systems, and allows a user to open a textual interface to a remote system. HTTP is the protocol that people use to access the web every day, and can carry a variety of different media, including text and graphics. Therefore, it would seem that implementing an http interface to the system would be the most user friendly and widely supported option for most client terminals. Web browsers are even available on palmtop devices and the latest 3G mobile phones.

1.3.3 PHYSICAL INTERFACE

There are a multitude of different network mediums in use today. For our system, we require two different types of interface.

Firstly, a direct minimal cable connection that can be used by as wide a variety of different host systems as possible would allow a single user access to the device. This interface will be used in a point-to-point configuration, but could be expanded for multiple remote use by connecting to a machine with appropriate routing software.

A most effective interface would be to connect the system directly to a local area network (LAN) infrastructure. A variety of different LAN standards exist, the most widely used today fall under IEEE category 802. These standards define technologies such as Token Ring and Token bus, and also the most popular of the LAN standards, CSMA/CD, the technology behind Ethernet networking.

Supporting both interface types (serial and Ethernet) described above will allow the device to be used with many client terminal interface types. We support SLIP connections of a dedicated serial port and develop an Ethernet capability.5 - Networking Design and Implementation

1.4 DEBUGGING DURING EMBEDDED DEVELOPMENT

One of the problems faced whilst developing in an embedded environment is the lack of user interpretable debug facilities. Whilst debugging with the HCS12 development kit, there are only two main options available

- A bank of 8 LEDs that can be used to indicate various output, such as a heartbeat, indication that a particular part of code has been reached, or the output of an 8bit number.
- The Real Time Debugger, which gives the ability to spy on memory and resources, and set breakpoints at specific routines in the code.

These two methods are very useful, but sometimes it is advantageous to be able to print out messages, numerical values and status information in a user readable format. Normally this would be done by inserting a `printf()` statement that would output the message to standard I/O, but this is not natively possible on an embedded microcontroller.

1.4.1 THE SERIAL DEBUG INTERFACE

As discussed earlier, the HCS12 supports two Serial Communications Interface devices among its communications features. One of these units is already being used in the project to handle SLIP traffic, however the second interface remains free. By utilising this interface to send (and possibly receive) plain ASCII bytes, it is possible to construct routines with similar functionality to `printf()` and `scanf()`, allowing the board to have a custom I/O interface. A screenshot of this functionality is included in Appendix VI.

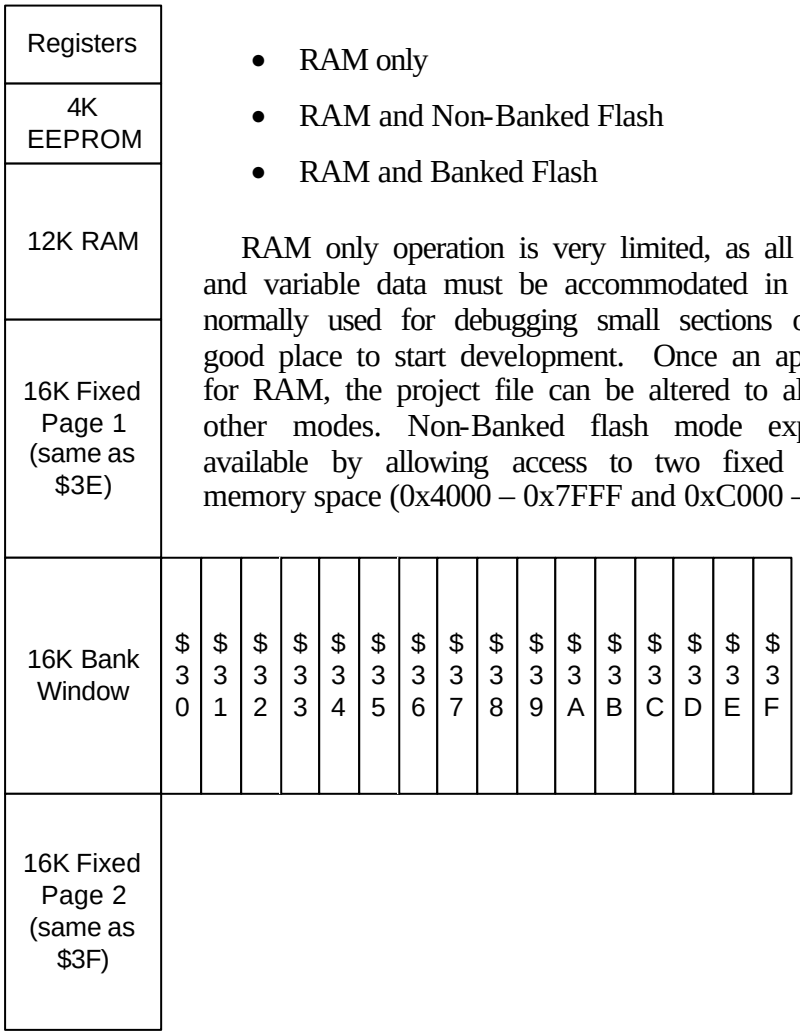
To implement the serial connectivity, the development board has a serial logic to RS232 converter chip that handles the voltage conversions necessary from 0 to 5v logic into +12 to -12 RS232 signals. This chip can handle two simultaneous serial interfaces, however only one of these interfaces can be connected to the external RS232 socket. Additionally, there is no on-board support for hardware based flow control, so any external communication with the interface must disable flow control for successful communication. Two jumper pins on the development board can select the interface that is connected to the external RS232 connector. Use of the second interface can be

made possible by creating a custom serial cable that can connect to the two unused jumper pins. These pins are connected to the Tx and Rx outputs of the serial chip. For successful communication, the ground pin of the serial cable must also be connected to the ground on the board (this connection is already provided on the external header socket).

1.4.2 MEMORY AND ASSOCIATED MODES OF THE HCS12

As mentioned previously, the HCS12DP256 device we are using has a total of 256Kbytes of flash programmable ROM. This ROM is normally programmed via the BDM interface, but can also be programmed selectively in blocks via application code running on the controller.

There are three main memory configurations that can be used



- RAM only
- RAM and Non-Banked Flash
- RAM and Banked Flash

RAM only operation is very limited, as all application code, constant and variable data must be accommodated in 12Kbytes. This mode is normally used for debugging small sections of code, and is usually a good place to start development. Once an application becomes too big for RAM, the project file can be altered to allow access to one of the other modes. Non-Banked flash mode expands the storage space available by allowing access to two fixed windows into the paged memory space (0x4000 – 0x7FFF and 0xC000 – 0xFEFF respectively).

These two address spaces correspond to the top 2 16Kbyte areas in banked Flash, and can be used for both application code and fixed constant storage This is shown in the memory map diagram opposite. Banked memory mode is the most flexible mode

available, and is recommended for most projects. This mode allows access to the same memory space as the Non Banked model, but also allows access to the rest of the 256Kbyte Flash space via a 16Kbyte paging window, from 0x8000 to 0xC000. This window can be configured to point at any one of the 16 areas available in the 256Kbyte Flash block, including the two areas that are accessible through the fixed windows, by setting a page pointer register. Having the two fixed areas accessible through both fixed addressing and paged mechanisms is invalid, so one method must be chosen. Usually having 32Kbyte of flash accessible in the fixed address space is chosen, as this allows both these areas and the flash window to be accessed simultaneously without manipulating the window pointer register.

Although the banked memory model suits most projects, it does have some restrictions. If a function residing in paged memory requires access to constant data, this data must either be located in the same physical page, or reside in a fixed area in memory. This shows the benefit of having 32Kbyte of flash accessible from a fixed address space.

During code development, specific functions and associated constants can be forced into particular pages in Flash. This is achieved by inserting **#pragma** statements into the code before definitions. If these are not used, the linker will use its own discretion to place objects. This is usually fine for most code, but the linker is not always intelligent enough to spot certain subtle use of variables and pointers.

The project linker file contains all declarations for the different memory areas, and is discussed in Appendix III. Full documentation of memory placement is available in both the compiler reference manual on the accompanying project CD, and Metrowerks specific detail is contained in Motorola document AN2216/D, which is also available on the CD.

1.5 PORTING UIP TO THE HCS12

As the full source for uIP is freely available, very little of this source has to be changed to allow a basic stack to be implemented. The main issues that need to be addressed are:

- Creation of a network driver
- Creation of the main polling code loop which will execute on the HCS12
- “Endianness” issues
- Compiler type issues

The creation of the network drivers are covered elsewhere in this report – this section will concentrate on the polling loop, and some other more general issues.

At the highest level, the polling loop has responsibility for the following

- Check if new incoming data is present
- Process the data
- Invoke any outgoing responses
- Periodically polling active connections to invoke timeouts

To implement the periodic polling function above, the "Modulus Down Counter" on the HCS12 is used. This device is part of the Enhanced Capture Timer (ECT) module, and is fully described in the associated Motorola document. A pre-scaler is used which gives an interrupt approximately 8 times a second. The associated ISR scales this down by a factor of 8, and will increment a global system time counter approximately once a second. The polling loop can then monitor this variable every code cycle, and if it increments, the periodic functions can be invoked. Other uses of the time count variable are discussed in the application section of this report.

1.5.1 ENDIAN ISSUES

When porting any code to a new target architecture, one of the most important issues that must be taken into account is the endian configuration of the target architecture. Most processors nowadays are "little endian" based. However, the Motorola range of processors all use big endian byte ordering. For networked applications, using a processor which is natively big endian can actually yield large performance increases over a little endian architecture, as the byte ordering on wired networks is also big endian based (also known as "network order").

uIP comes with some built in macros that can compensate for different endian schemes. However, for our implementation, these macros will effectively do nothing, as we do not require changing byte ordering at any time. As the code that has been developed for this project is targeted specifically at the hardware of the HCS12, specific attention has not been paid to byte ordering issues. However, this will affect the portability of the code to a different architecture using little endian byte ordering.

1.5.2 COMPILER TYPE ISSUES

As this project is centred round an embedded system, we want to be as conservative as possible while declaring data types. For this reason, it is useful to have some standard data types defined which we know will allocate us a specified space in memory. The following types are declared in the Motorola `s12_common.h` header file, and all uIP declarations are changed to use these types.

Typedef	Compiler type	Range

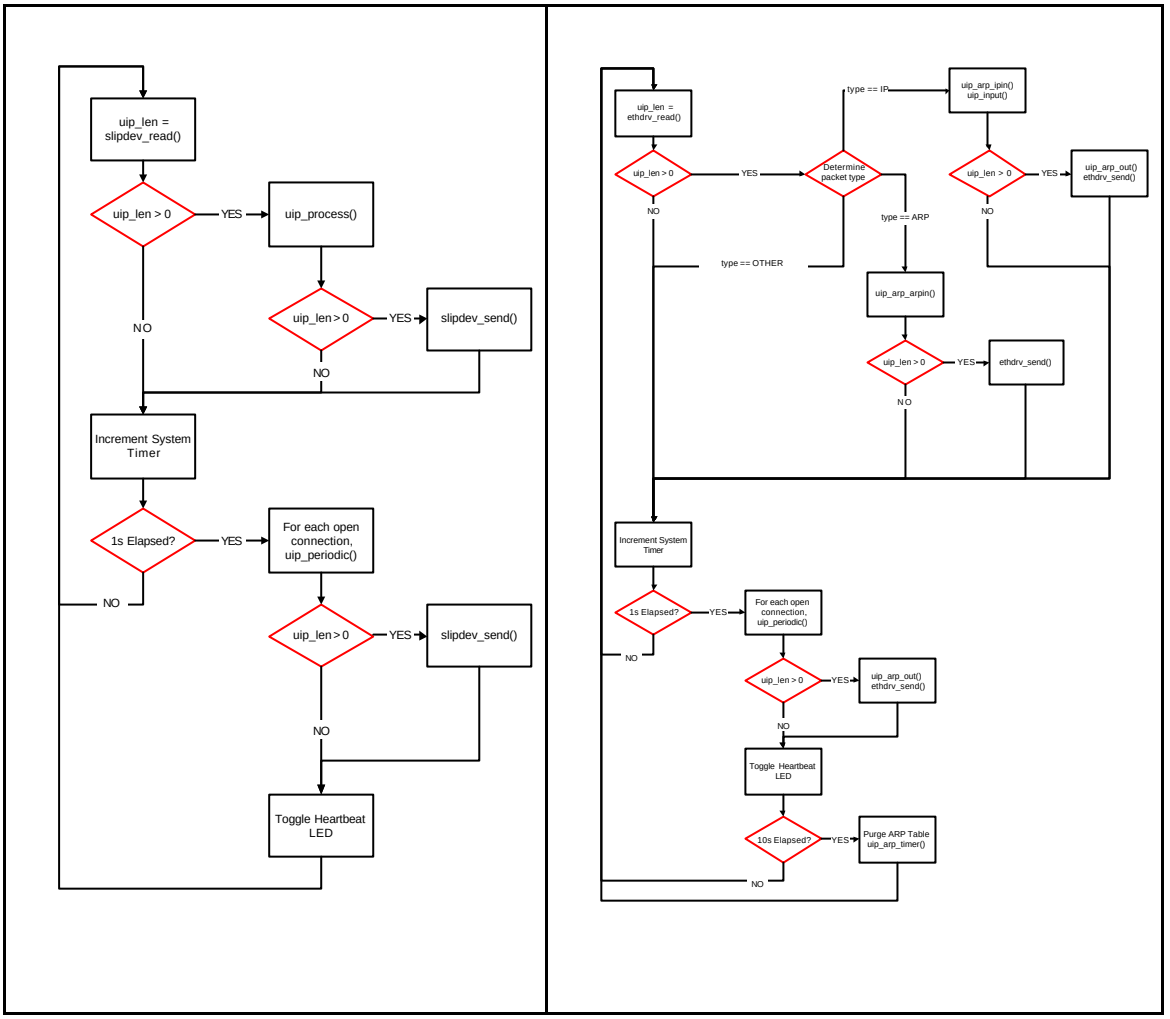
TU08	unsigned char	0 to 255
tU16	unsigned int	0 to 65535
tU32	unsigned long	0 to 4294967295
tS08	signed char	-128 to 127
tS16	signed int	-32768 to 32767
tS32	signed long	-2147483648 to 2147483647

1.5.3 POLLING LOOPS

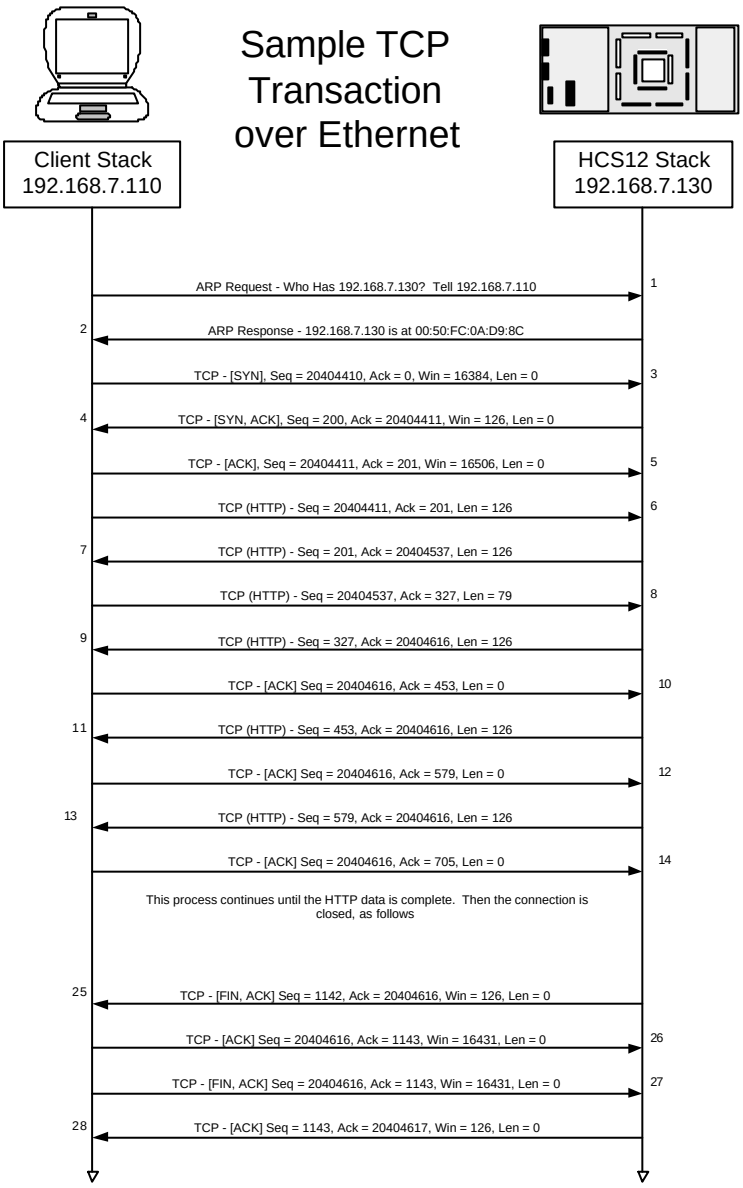
As we need to have the functionality to potentially support more than one network interface (though not concurrently), there will be subtle differences to the content of each polling loop. This problem is catered for in the code by the use of **`#ifdef`** statements that determine which network interface is to be supported in the build of the software. The required interface, and various other project specific configurable parameters can be set in the **`uipopt.h`** header file.

The basic functionality of each polling loop is fundamentally the same, and is shown by the following two diagrams.

SLIP Main Loop	Ethernet Main Loop



A diagram showing the flow of data and different messages used best shows the actual functionality of a TCP transaction with the stack. The diagram on the next page shows a sample TCP transaction carrying http traffic between the board and a windows client node.



The initial part of the transaction shows the ARP (Address Resolution Protocol) lookup that must be performed whilst using Ethernet. This takes the form of a broadcast packet that is issued from the client node, asking who has the IP address "192.168.7.130". As this is the IP of our node, the uIP ARP engine replies directly to the client machine informing that "192.168.7.130" is the IP address attached to our MAC address (0x0050FC09AD98C). This then allows the client to address ethernet packets directly to our network interface.

The next packet that is sent is a TCP Synchronise (SYN) request. This indicates that the client wishes to establish a TCP connection with us. As this packet contains IP, it is passed to **`uip_process()`**, which determines what to do with it. As you can see from the diagram, uIP has determined correctly that this is a valid TCP transmission destined for a port that we are listening on (http port 80), as it has generated a response packet that has both the SYN flag and the ACK flag set. This indicates to the client that we are happy to accept the connection. The client replies by sending a further acknowledgement back to us, and at this point the connection is in the ESTABLISHED state. This handshaking occurs for every TCP connection, and a similar negotiation is carried out to pull the connection down when it is no longer required.

There are some other important observations to note about the connection initiation shown above. The first is the significance of the Win field. This field is contained in all TCP headers and determines the maximum size of data (in bytes) a host is able to accept. In the example above, the windows client host is able to accept a maximum packet size of 16506, whereas our packets state that the Maximum Segment Size (MSS) that we are able to receive is 126 bytes. This value is governed by the size of the data buffer which uIP is configured to use.

The diagram above also shows the use of sequence numbers in TCP transactions. These 16bit numbers are contained in every TCP header, and are used to control the flow of data in the connection. It is defined that these numbers can be initialised to any value at the start of a TCP connection. Whilst the connection is set up, the client sends an initial number, 20404410 in the example above. When uIP responds to this, it returns the value of the received sequence number plus one as the ACK field in the header. uIP also sends a sequence number of it's own in the response packet. This is repeated in all further negotiations. Once payload data is sent in packets, this also affects the sequence number that is sent in the ACK field. Instead of increasing the number by one, it is increased by the number of bytes that have been received in the packet that is being acknowledged. For example, the first data payload in transaction 6 above is 126 bytes long (the MSS of our connection). When we acknowledge that data, we increase the ACK field by 126 taking it from 20404411 to 20404537. As TCP is a full duplex protocol, we can also send a data payload back with the acknowledgement, as indicated by the LEN field being set to 126 in transaction 7 above.

1.5.4 UIP SEQUENCE NUMBER CALCULATIONS

Calculating sequence numbers and checksums is a mathematically intensive operation, especially for a low powered microcontroller. In uIP, this functionality has been removed from the main uIP source to the uip_arch module, allowing specific routines to be easily optimised for a particular architecture. Although these calculations will be fairly intensive, our system is not intended to handle large numbers of connections, and the generic code has been left "as is". This could however be optimised in the future, possibly by implementing in raw HCS12 assembler.

The uIP stack also has to deal with invalid incoming connections, and also send a suitable response if there are no free connection slots left on the board. In both cases, uIP will send a TCP Reset – a packet with the RST flag enabled. This indicates to the remote host that a connection cannot be established.

1.6 IMPLEMENTATION OF UDP FOR UIP

While many IP based services rely on TCP as a transport mechanism, there are other services that require a lightweight connectionless protocol for data transmission. User Datagram Protocol (UDP) is such a protocol, designed to send connectionless data packets over IP. Compared to TCP, UDP itself employs no acknowledgement or retransmission mechanism. If a datagram packet is lost or determined to be corrupt, it is up to the higher-level applications to realise this and invoke an appropriate retransmission. An optional 16bit checksum is available to validate integrity of packets.

Having the ability to send and receive UDP datagrams with our system will open up various options that are described later in this report. As uIP is primarily designed to support TCP traffic, there are some fundamental changes that have to be made to allow parallel support of UDP.

1.6.1 MAIN UIP CODE CHANGES FOR UDP

There are some similarities between TCP and UDP packet reception, hence it is a fairly methodical process to add UDP support to the stack. Both protocols use the concept of source and destination ports, and we therefore need to replicate some of the TCP port functionality for UDP. For incoming datagram reception, this necessitated the creation of

- An array of UDP listening ports
- A **`uip_udplisten(port)`** function
- A structure representing a UDP header to map to the incoming packet buffer

For outgoing datagrams, the following was required

- Ability to alter the initialised position of uip_buf pointer. Normally this is set to allow space for a combined IP/TCP header at the start of the buffer. A corresponding UDP header is considerably smaller. The **`uip_presendUDP()`** function addresses this issue.
- Similar to TCP, we require a function that actually sets up the header data ready to send the packet. The **`uip_sendUDP()`** achieves this, and calls the appropriate checksum calculation routines.

The **`uip_process()`** call is designed to check for TCP and ICMP packets carried as payload on IP packets. To be able to receive incoming UDP, a clause must be inserted into the uIP code to allow these packets to be broken out from the function.

As already mentioned, UDP has an optional checksum field. If this field is unused, it will be set to zero. In our implementation, UDP checksums are unsupported, and as a result, the **`uip_udpchksum()`** function returns zero by default. This function has been created to allow UDP checksum code to be implemented at a future date if necessary.

1.6.2 PROCEDURE FOR AN APPLICATION TO SEND A UDP DATAGRAM

As previously mentioned, if we are going to write application data directly to the stack buffer, we must ensure that the **`uip_appdata`** pointer is set to the start of the UDP payload area. **`uip_presendUDP()`** sets the pointer 28 octets into the buffer – the length of a combined UDP and IP header. An application also has the option of setting **`uip_appdata`** to a buffer of its choice, if for example a static buffer in flash requires to be sent.

Once the buffer has been populated (or data pointer set), a send function similar to TCP must be invoked.

```
uip_sendUDP(tU16* ripaddr, tU16 srcPrt, tU16 destPrt, tU16 len)
```

This function takes a pointer to a 32bit IP address buffer, source port (zeroed if not used), mandatory destination port and the length of the datagram payload. The function will populate the correct UDP and IP header fields, and calculate any checksums. The value of **`uip_len`** will be set accordingly, and the length check in the main loop will invoke the network driver to send the packet.

1.7 IMPLEMENTATION OF TCP AND UDP MULTIPLEXING FUNCTIONS

One of the objectives of porting an IP stack to the HCS12 is to allow I/O access to both hardware connected to the device, and to other applications executing on the microcontroller. The native uIP stack API is designed to support a single transport protocol - TCP, and is really only targeted for a single application.

As we now need to support more than one transport protocol, and potentially multiple applications, we need to find a way of differentiating between TCP packets and the newly implemented UDP datagrams. These incoming packets must subsequently be directed to the correct application.

Differentiation of protocol type is handled in the **`uip_process()`** function, and it is here where the fundamental changes are made. In both TCP and UDP cases, a packet will only be accepted if

- It has passed all checksum and length checks
- It is destined for our IP address (the accepting of broadcast packets can also be enabled)
- It is destined for a port on which we are actively listening

Any packets arriving that do not conform to the above checks will be dropped by uIP. Once we have determined that we are definitely interested in a packet, we have to make a choice as to what to do with it. This process is slightly different for each protocol, so we will now consider each in detail.

1.7.1 TCP

As TCP is a connection oriented protocol, any arriving TCP packet may be for either an already open connection or it may be a SYN packet trying to open a new connection. uIP first checks to see if there is a match for the source and destination IP of any open connections, if this is found the source and destination ports are also checked. If a match is found, the packet is deemed part of an open connection and the stack checks what next needs done with the connection. If an open connection is not found, the destination port of the packet is checked against our array of TCP listening ports. If no

match is found, a TCP Reset packet is sent in response. If a match is found, the stack will start to negotiate the new connection.

Once a connection is in progress, when new data arrives, a function is called to invoke the correct application. This is where a fundamental change is made to the stack to allow application multiplexing - the TCP Multiplexer module is invoked to make the decision.

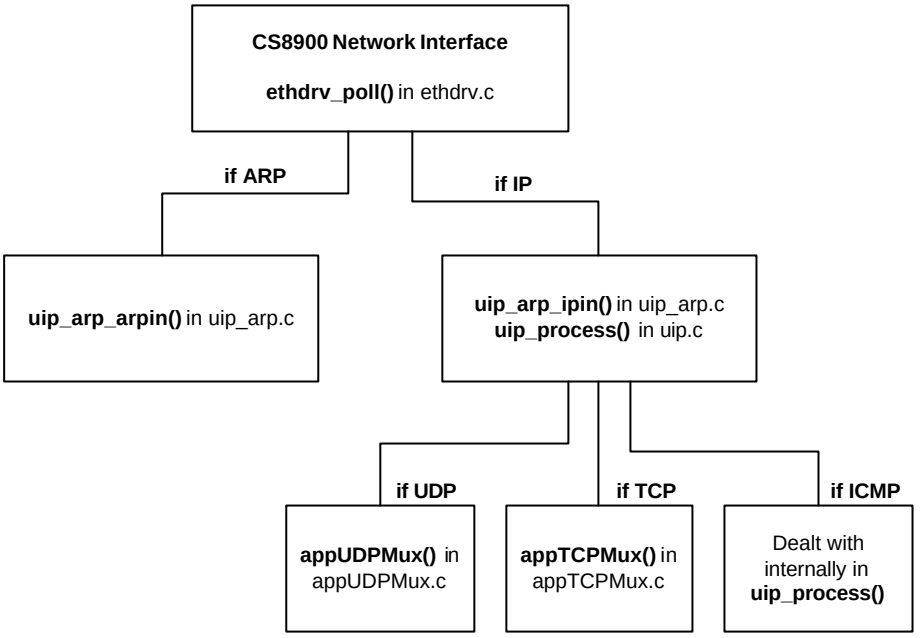
1.7.2 UDP

As UDP is a connectionless protocol, the stack does not need to worry whether the packet is part of an ongoing data transfer. The packet is simply checked against an array of listening UDP ports, and if a match is found, the UDP Multiplexer is called. If no match is discovered, the packet is simply dropped – no response needs to be sent.

1.7.3 TCP AND UDP MULTIPLEXERS

By the time a packet reaches either of these routines, it has been determined that it is definitely for our node on the network, and it is also destined for a port on which we are listening. The function of the multiplexers is to determine which application the packet is for, and subsequently to invoke that application. In most cases this is achieved by checking the port the packet is destined for. In some specialised cases, a check can also be made on the port the packet has been sent from, or the remote IP address. This is especially important when dealing with broadcast packets. For example, with DHCP (discussed later) a response can be received from more than one listening server and it is the responsibility of either the multiplexing function (or in our case, the client itself) to determine if the packet is valid. As the multiplexer functions are called for **every** IP packet that makes it past the checks in the stack, these functions should be as small and efficient as possible – any CPU intensive operation here could cause a bottleneck in terms of overall throughput.

The diagram below further illustrates the flow of data from the network interface to the main functional components in the system.



1.7.4 IP SECURITY CONSIDERATIONS

As security is not a big concern in this "proof of concept" project, the only checks that are performed are against destination fields in IP headers – determining if a packet is destined for our node. However, there are some apparent places in the chain of events where some security features could be implemented. The most obvious location would be to implement a security module directly after the network interface, where every packet could be scrutinised before passing to the stack for consideration. This would be the most secure solution, and would allow filtering on various fields, such as an "allowed" or "denied" Source IP Range. Obviously this sort of scheme would require some planning, and to be effective would have to be easily updatable. Also, on a resource-limited microcontroller, this type of scheme would raise some performance issues.

1.7.5 SUMMARY OF 3RD PARTY APPLICATION SUPPORT API

The discussion above has detailed the implementation of both the base protocol stack, and also further measures to ease integration of other TCP/UDP based applications.

To summarise, for an incoming packet to reach an application the following must be implemented

- If the application receives data via a broadcast mechanism (DHCP for example), the broadcast functionality must be enabled in the stack by calling the `EnableBroadcastTCP()` or `EnableBroadcastUDP()` routines during initialisation.

- If the application requires initialisation, a function call must be added to `initTCPApps()` or `initUDPApps()` in the corresponding multiplexer module. The init routines are called by default when uIP initialises at system startup.
- The correct listening port must be enabled (during application initialisation) for a packet to reach the multiplexer stage.
- A check must be added to the multiplexer function that calls an application function on reception of a packet destined for it.

1.8 NETWORK INTERFACE TYPES

The physical network interface for the project is responsible for actually sending a specified stream of bytes – a packet, over a network medium. During the construction of the HCS12 server, two different network access methods are investigated and implemented – Serial Line IP and Ethernet. Whilst having many differences in terms of byte coding, speed and physical connection, both network access methods share some common functionality in terms of the format of data sent over the external communication link.

1.9 ETHERNET AND SLIP DATA STREAMS

Although both access mechanisms send and receive the same IP Protocol Data Units (PDUs), each system packages these PDUs in a different manner. Both require the data to be split and sent in well-defined blocks. As SLIP is a direct connection between the system and a host it requires less header and control data to be included. Conversely, Ethernet is a multiple access broadcast system and therefore requires use of a Media Access Control (MAC). The points are summarised in the following table.

	Serial IP	**Ethernet**
Address Range	None	48-bit MAC Address
Transmit Availability	Any Time	When Network Idle
Reception	Assumed Reliable	Assumed Unreliable
Format	Stream with Escape Bytes	Frame
Data Length	Potentially Unlimited	1500 Byte Max Payload
Addressing	None	Source, Destination

The differences between the two packet formats can also be observed in the following diagram.

SLIP	NO MAC or Protocol Ident	Esc	IP Header	Data	Esc
		1 byte	20 – 60 bytes	Variable Length	1 byte

Ethernet	Dest MAC	Source MAC	Protocol	IP Header	Data
	6 bytes	6 bytes	2 bytes	20 – 60 bytes	Variable Length

Both drivers share the common features that they will require three main functions to be accessible to the main program loop

- An initialisation routine.
- A polling read routine to receive new data if present.
- A send routine to send out pre formatted data over the network interface.

1.10 SERIAL LINE INTERNET PROTOCOL (SLIP)

Before considering the software driver aspects of the SLIP implementation, it is important to consider the physical connection required between the host PC and the HCS12 development board.

A serial connection at its most basic level will normally consist of a pair of shift registers, controlled by a common clock signal. The output of one shift register is fed to the input of the other, and a single bit is shifted between them every clock tick. While this is perfectly acceptable for two devices that are physically close together (on the same PCB for example), it is not suitable for a link between devices. To overcome this problem, the RS232 specification was developed. Data in an RS232 connection is sent asynchronously, without the use of a common clocking signal – the formatting of the data on the wire allows for synchronisation between the two devices.

The HCS12 contains two Serial Communications Interfaces (discussed earlier) – we are going to use the SCI0 device for our SLIP connection. By making sure that jumper pins J12 and J13 are both in the 0-1 position, we ensure that the board's external RS232 connector is connected to the SCI0 device. As we are connecting the board to a host PC, we require a simple male to female straight serial cable – this will connect the Tx from the board to the Rx of the host, and vice versa. As already mentioned, the HCS12

development board has no support for hardware flow control, and this must to be catered for when configuring the client PC.

1.10.1 SLIP DRIVER DESIGN

To isolate complexity from the main driver routines, we first need to construct two low level routines – one to receive a byte from the SCI device, and the other to send a byte. The following two functions are created for that purpose.

```
u8_t sci0Get(u8_t *byte)
u8_t sci0Put(u8_t byte)
```

Both functions work with individual bytes, therefore allowing the driver to send and receive raw bytes from the port. The get function must be non-blocking, and will return a value of 0 if there is no new data at the port.

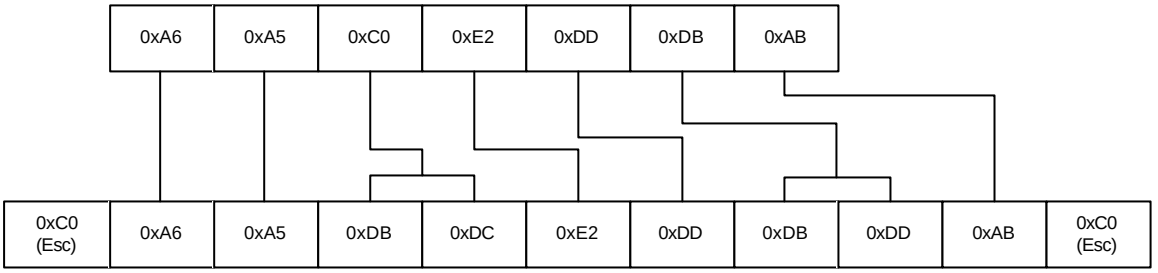
Now that we have the functionality to send and receive raw bytes, we need to construct an initialisation function, along with functions to deal with transmission and reception of SLIP frames. An example SLIP driver for the Commodore 64 is included with the uIP distribution – this will be used as a high level basis for the overall driver design.

1.10.2 SLIP DATA FORMATS AND CONTROL CHARACTERS

As the SLIP protocol sends streams of raw bytes across a serial link, there must be a way to define the start and end of individual data blocks. This is achieved by using a special code, known as the Escape Character. As the escape character could potentially be a legitimate part of a data stream when transmitting binary information, special functionality must be built into the send and receive routines to cope with this. A byte-stuffing sequence is used in the following manner.

- The end of a block is signified by the escape character 0xC0.
- Most SLIP devices will send the escape character immediately before a frame if the line has been idle to help with sequencing (this is not however required).
- If a byte in the data stream is equal to 0xC0, the values 0xDB and 0xDC are substituted, and subsequently converted back at the receiver
- If a byte in the data stream is equal to 0xDB, the values 0xDB, 0xDD are sent, and again substituted back to a single 0xDB at the receiver

These measures ensure that a frame of data can be transmitted easily and correctly over a serial link, and are illustrated by the following byte coding sequence.



1.10.3 MODEM EMULATION & DIRECT CABLE CONNECTIONS

Whilst designing a SLIP deriver for the system, it became obvious from looking through various manuals and settings for the Windows operating system that a SLIP connection was primarily designed to work via a modem link. Obviously, in most cases this would be the case, but we wish to connect the project board directly to the host PC. To overcome this problem, extra functionality must be built into the SLIP driver to emulate a modem connection. The situation is slightly different for Windows 2000 and Unix – these are discussed separately below.

1.10.4 WINDOWS 98/ME

When configuring a windows client, the best option is to choose a SLIP connection using a standard 28000bps modem. This is the most basic SLIP connection that windows will support. In addition to the standard SLIP frames, windows will try to initiate various 'AT' commands to the modem – these are standard modem command strings. When one of these commands is sent, the windows client will wait until an 'OK' response is received. Therefore, every time we receive an 'AT' string from the host, we should immediately respond with an 'OK' string.

1.10.5 DIRECT CABLE CONNECTION IN WIN2000

Windows 2000 operates in a slightly different manner, and simple 'AT' modem command set emulation is not sufficient to ensure a reliable connection. However, Windows 2000 supports a modem device known as "Direct Cable Connection" which removes the need for modem emulation altogether. This device does require some simple negotiation at start-up. When the windows client tries to open a connection, it will send the string "CLIENT" to the board. Once the board has received this command, it replies with the string "CLIENTSERVER" and the SLIP connection is then free to start. This initial negotiation is the only non-SLIP commands that need to be catered for.

1.10.6 UNIX/LINUX CONFIGURATIONS

Connection methods vary between different distributions, but most setups will allow you to create a simple SLIP connection requiring no negotiation or modem emulation.

However, if this is not possible, configuring to use a standard modem similar to the standard windows setup, and enabling 'AT' emulation should be sufficient.

All of the emulation schemes above are supported, and can be selected in the **`uipopt.h`** configuration file.

1.10.7 SLIP & SCI INITIALISATION

The Initialisation function that must be called at start-up is defined

```
void slipdev_init(void)
```

This function initialises the HCS12 SCI device by enabling the transmitter and receiver, and setting an appropriate Baud rate. The port baud rate is set in a register associated with the device. It is a clock divider ratio, and is determined by the following function.

$$SCIBaudRate = \frac{SCIModuleClock}{(16 * SCIBR[12:0])}$$

The SCI Module clock is defined as being half the development board oscillator clock, therefore 8MHz. Using the formula, a baud rate of 34000bps can be obtained by setting the SCIBR[12:0] field to the value 13 (0x0D). Due to the limitations of the oscillator on the development board, we cannot clock the SCI device much faster than 34000bps as bytes start to be dropped from the serial line. To overcome this limitation, the HCS12 itself would have to be clocked at a faster rate by a faster oscillator. Further information on the SCI device and its modes of operation can be found in the specialised Motorola documentation, available on the project CD.

If we are going to be communicating with a Windows 2000 host, the appropriate request and response initialisation functions will be executed during initialisation, blocking until a response is received, as described above.

1.10.8 SLIP READ OPERATION

This function is non-blocking, and will return 0 if no data has been received at the serial port. It is defined

```
u8_t / u16_t slipdev_read(void)
```

The function will first poll the serial port to see if a data byte has arrived. If one has, the function will continue reading bytes and storing them in its own buffer until either

the buffer limit has been exceeded, in which case the slip frame will be dropped, or the end escape character has been received. If the special escape character 0xDB is received, this is not placed in the buffer – instead the next byte is received and if the two bytes form the special escape sequences discussed above, the appropriate decoded byte will be placed into the buffer.

Once a complete frame has been received, it is copied from the driver buffer to the uIP stack buffer, and the appropriate received length is returned. The AT Modem command emulation is also built into this read operation, and if it is detected the 'OK' response is sent back to the host.

1.10.9 SLIP SEND OPERATION

The send operation is defined

```
void slipdev_send(void)
```

This function is fairly simple in operation – when it is called, it first sends the SLIP escape character 0xC0. It will the iterate through the uIP buffer, checking each byte as it sends it to make sure that it does not equal the special escape characters. If this is the case, the appropriate substitution bytes will be sent, otherwise the byte itself is sent. Once the full buffer has been sent, the escape character 0xC0 is sent again, determining the end of the frame.

These simple functions allow a simple SLIP connection to be supported, and will therefore allow IP traffic to be sent from the HCS12 device to a client terminal.

1.11 IEEE802.3 - ETHERNET

While SLIP allows a TCP/IP network connection to be established to the development board, it's main limiting factor is that it is a single point to point link between the board and host, therefore the host machine is the only point of access to the board. Although this problem could be overcome by configuring some IP routing software on the host PC, this is not ideal, as it does not allow a direct connection to the board from a remote network station.

A much cleaner and more manageable solution is to enable the system to participate independently on a Local Area Network, interfacing directly to a Network Interface Controller (NIC).

Whilst considering a suitable NIC, some important considerations had to be taken into account, as the controller would have to be connected to the HCS12 by general I/O lines. Specifically, we were interested in

- Data and address bus widths (number of I/O interface pins required)

- Power consumption
- On board RAM / Buffering Capabilities

Taking these points into account, there were really only two suitable interfaces found - the Cirrus Crystal LAN CS8900A and Realtek RTL8019AS. Both interfaces are 10BaseT compatible, and although they are primarily designed for 16bit ISA bus systems, both can run in a reduced 8bit mode, reducing the external I/O capacity and therefore the overall pin outs required from the chip. Both chips are also functionally identical, although there are some differences in terms of internal addressing and driver requirements.

The Integrated Circuits described above are self-contained in Surface Mount Thin Quad Flat Pack (TQFT) packaging. While this makes the ICs very small and compact, it also makes them virtually impossible to mount onto a PCB without specialised equipment.

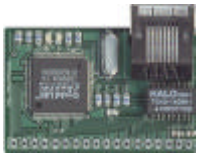
This point has been recognised by some electronics manufacturers, and both interfaces are available ready mounted onto small development PCBs, with the relevant pins required for 8bit operation brought out to accessible solder points. The development boards also have other necessary components on board such as an oscillator, RJ45 connector, link and data LEDs, some required resistors and capacitors, and an isolation module that matches the impedance of the Tx and Rx signals from the chip to the RJ45 connection.

A Norwegian company named Systor Vest manufactures the Ethernet board chosen for the project. It contains the Cirrus CS8900A chip described above, and conforms to all our requirements. The board will draw a maximum power of 80mA. When combined with the microcontroller power requirements, the total is well within the 1.2A maximum we can draw from the development board power supply.

1.11.1 ETHERNET BOARD PHYSICAL CONNECTION

When designing the connection between the embedded Ethernet board and the microcontroller, we require the following pin connections

- (Input) Power and Ground
- (Input/Output) 8bit Data Bus
- (Input) 4bit Address Bus
- (Input) Read Enable (Active Low)
- (Input) Write Enable (Active Low)
- (Input) Chip Select (Active Low)
- (Output) Interrupt Request (Optional)



The Embedded

The CS8900A is capable of operating in either polled or interrupt driven modes. Normally interrupt driven operation would be desirable, as this would remove the additional overhead which polling places on the host microcontroller. However, the CS8900A does not support interrupt driven operation whilst being operated in 8bit mode. As the development board does bring out the IRQ pin, we will connect it to the controller for completeness, but the pin will not be actively used in our system and could easily be omitted.

This results in a total required pin count of 18. As many of the HCS12 pins are multi purpose, we want to try to maximise availability of the on chip components, while at the same time keep wiring and driver coding simple. It is decided for this project to use general I/O ports A and H for communication with the Ethernet module. The physical pin connections used are listed in Appendix IV.

We have now covered all of the physical connection design to the board. When the development board is powered on, and an Ethernet connection is made to a hub (or straight to a host using a cross wired cable), the link light on the card will illuminate green. Data being received or transmitted will result in the link LED flashing orange.

1.11.2 ETHERNET DRIVER DESIGN

Successful communication with the Ethernet module is based around the process of asserting the correct address and data line levels, and then toggling the read or write pin momentarily low accordingly. Most ISA based systems will achieve this goal by memory mapping the relevant address and data bus range of the card to an unused addressable range of system memory, thereby allowing Direct Memory Access to the card and simplifying access. To the software driver, the card will be seen as an area of system memory. In our case, we do not have a free area of mapable system memory that can be dedicated to I/O, and rely only on manually asserting general I/O lines for operation. For reliable operation, it is desirable to construct the driver in a robust manner, isolating the assertion of I/O lines from the main code as much as possible.

When communicating with the CS8900A in 8-bit mode, we have access to 8 data ports, each 16bits wide. For ease of understanding, this can be represented as eight paired 8bit locations. As the controller is primarily designed for 16bit operation, one of the restrictions that we must comply with is that both the low and high bytes of each port must be accessed during any communication with the card.

The CS8900 ports and associated address range are summarised in the table below

1.11.2.1 CS8900A PORTS

Address Offset	Type	Description	
0x00 : 0x01	Read/Write	Receive/Transmit Data (Port 0)	
0x02 : 0x03	Read/Write	Receive/Transmit Data (Port 1)	*Not Used*
0x04 : 0x05	Write Only	TxCMD (Transmit Command)	
0x06 : 0x07	Write Only	TxLength (Transmit Length	
0x08 : 0x09	Read Only	Interrupt Status Queue	*Not Used*
0x0A : 0x0B	Read/Write	PacketPage Pointer	
0x0C : 0x0D	Read/Write	PacketPage Data (Port 0)	
0x0E : 0x0F	Read/Write	PacketPage Data (Port 1)	*Not Used*

As we are operating in 8bit-polled mode, we do not need to access Receive/Transmit Port 1, Interrupt Status Queue or the PacketPage Data Port 1.

The CS8900 employs a very neat and efficient system to give access to internal status and control registers – the PacketPage system. Access to packet page data is gained by the following two-step procedure.

- Set the PacketPage Pointer port to the correct register offset.
- Access the corresponding data via the PacketPage Data port.

The following table summarises the high level layout of the PacketPage address space.

PacketPage Address Offset	Contents Summary
0x0000 – 0x0045	Bus Interface Registers
0x0100 – 0x013F	Status and Control Registers
0x0140 – 0x014F	Initiate Transmit Registers
0x0150 – 0x015D	Address Filter Registers
0x0400	Receive Frame Location
0x0A00	Transmit Frame Location

For our driver operation, we are really only concerned with the Status and Control registers section, all other card accesses are via the main address space.

1.11.3 CS8900A LOW LEVEL I/O ROUTINES

At the lowest level in our software driver, we require functions that can access the ports on the CS8900. Two functions are constructed,

```
u8_t cs8900_reg_read(u8_t offset)
void cs8900_reg_write(u8_t offset, u8_t value)
```

Both of these functions take the register-offset value as an argument, and use this to assert the address lines accordingly. Port A direction register is set to input or output, and the value is read/written accordingly by momentarily toggling the RD or WR pin low.

By manipulation of the two functions above, we can perform all of the operations necessary to operate the CS8900A. For ease of coding and clarity, a corresponding pair of functions are constructed which will access the 16bit PacketPage register bank. They are defined as,

```
u16_t cs8900_ppreg_read(u16_t offset)
void cs8900_ppreg_write(u16_t offset, u16_t value)
```

The functions above can be thought of operating a level above the basic register read and write functions. These methods take a 16bit PacketPage offset as an argument.

This value is written to the PacketPage Pointer port byte at a time – LSB then MSB. The corresponding data value is then either written to or read from the PacketPage Data Port 0.

These two pairs of functions, whilst fairly complex and time critical themselves, isolate this complexity from the rest of the driver code, and allow access to both main registers and PacketPage registers easily and efficiently.

Now that we have the low level functionality covered, all that remains is to construct initialisation, read and write functions (similar to SLIP) that the main program loop on the microcontroller can actively use.

1.11.4 CS8900A INITIALISATION

Before we can start sending and receiving ethernet packets on the CS8900A device, we need to perform some routine housekeeping tasks. These are all carried out by the initialise function, defined

```
void ethdrv_init(void)
```

In this function, firstly we need to set the port direction of portH to Output. This is fixed, as all of the lines connected to port H (apart from the unused IRQ line) are used for sending signals to the device. We also have to pull the Write Enable (!WR) and Read Enable (!RD) pins high to disable them, and finally we pull the Chip Select (!CS) pin low to enable the device. We are now in a position to start communicating with the CS8900A IC.

Before we try to initiate any initialisation commands on the device, it is helpful to check that we have reliable communication. A standard way of achieving this is to read the EISA identification number from the card. This is a read only chip identification number contained in the very first 16bit PacketPage register, and for our CS8900A device it should be equal to 0x630E. If we manage to match the correct ID number, we know that we are definitely communicating successfully with a valid CS8900A device.

The next operation to execute is the self-test call. This resets the chip, and verifies that it is functioning correctly.

1.11.5 IEEE MAC ADDRESSING

Another important setting that **must** be performed before frame transmission and reception can occur is the Media Access Control (MAC) address setting. The MAC address consists of a 48bit number that must be unique to every Ethernet network card manufactured, and is used for host identification on a CSMA/CD broadcast LAN. It is

important to realise that this number is specific to the network interface, not the CS8900A device itself. If the CS8900A were mounted on an ISA card, this address would be hard coded into on board EEPROM, and would be set in the device at power on. As we do not have any on board firmware on our embedded Ethernet PCB, it is up to us to set the address to a suitable value in the driver initialisation. A 48bit MAC address is split into four fields, arranged as follows

1bit	1bit	22bits	24bits
0 : Unicast 1 : Multicast	0 : Universal 1 : Locally Managed	IEEE Allocated Manufacturer ID	Manufacturer Allocated Device ID

The first two bits are usually set to 00 as most LANs are unicast, and though locally managed MAC addressing is possible it is not widely used.

Using the scheme above, every manufactured network interface is given a unique MAC address. Ideally, we should give our interface a new unique address, but this is not practical as the IEEE charge a fee for a manufacturer ID allocation. For the purposes of this project, a MAC address from an old non-functioning PCI network interface is used. The value, in hexadecimal is

00–50–FC-0A-D9-8C

Once this crucial value is set, the final settings (manipulated through PacketPage accesses) are

- (possibly) Enable Full Duplex (if supported by the physical Ethernet connection)
- Set to accept valid addressed frames (for our MAC address) and broadcast frames (required for ARP and IP broadcast operation)
- Enable receiver and transmitter operation
- Force interface to 10baseT operation

1.11.6 POLLING OPERATION

As we have to operate the CS8900A in polled mode, we require a polling function that the main control loop can use to check for the presence of new data. This function is defined as follows

```
u8_t / u16_t ethdrv_poll(void)
```

This function simply checks the correct PacketPage status register to determine if a receive event has occurred. If it has, the **`ethdrv_read()`** function (discussed below) is called to retrieve the packet from the interface, and the received packet length is returned.

If no receive event has occurred (or the packet is too large), 0 is returned.

1.11.7 READ OPERATION

As the CS8900A is in 8bit mode, the read operation must follow a strict procedure whilst receiving Ethernet data, taking care to read low and high bytes from the data register in the correct order. The function is defined

```
u8_t / u16_t ethdrv_read(void)
```

It is called from the polling function when a packet is found to have arrived. The sequence of events is as follows. Similar to the SLIP driver, we have a fixed driver buffer space available, and any packets that are larger than the allocated buffer space are dropped. It is important to note that if we wish to drop a packet, to retain sequencing in the CS8900A, the entire packet must still be read from the device.

The first 16bit value that must be read once we know a packet has been received is the RxStatus 16bit value. Although we do not use this value, it still has to be read from Receive/Transmit Port 0, high byte first. The next value is the RxLength value (also high byte first). This value is very important, as it determines both how many bytes we must sequentially read, and also whether the packet will fit within our allocated buffer space.

Once these two values have been read, the data packet itself is retrieved from the card, byte by byte from Receive/Transmit Port 0. During this transaction, we must read the low byte first. The following table shows the start of a sample frame reception, indicating the sequence of associated register offsets required.

Value	Port	Port Offset	Direction
RxStatus MSB	Rx/Tx Port 0 MSB	0x01	Read
RxStatus LSB	Rx/Tx Port 0 LSB	0x00	Read
RxLength MSB	Rx/Tx Port 0 MSB	0x01	Read
RxLength LSB	Rx/Tx Port 0 LSB	0x00	Read
Data Byte 0	Rx/Tx Port 0	0x00	Read

	LSB		
Data Byte 1	Rx/Tx Port 0 MSB	0x01	Read
Data Byte 2	Rx/Tx Port 0 LSB	0x00	Read
Data Byte 3	Rx/Tx Port 0 MSB	0x01	Read
Data Byte 4	Rx/Tx Port 0 LSB	0x00	Read
etc	etc	etc	Read

The sequence above is continued until the correct number of data bytes (determined by RxLength) is read.

In terms of coding, a boolean value 'keep' is defined, and is used to determine if received bytes are buffered or dropped as they are read. For completeness, it is helpful to include the following code snippet that shows the loop that actually reads the data bytes from the CS8900A. Note the use of the modulus operator that determines if we are reading an even or odd byte, and calls the read operation on the LSB or MSB register accordingly.

```
for(i=0; i < recLen; i++) {
  if(i%2 == 0)
    c = cs8900_reg_read(CS_DP0_LSB);
  else
    c = cs8900_reg_read(CS_DP0_MSB);
  if(keep)
    eth_buf[i] = c;
}
```

Once the packet is completely transferred, it is copied from our driver buffer to the uIP stack buffer, and the received packet length is returned. If the packet is larger than our buffer, a value of 0 is returned.

1.11.8 WRITE OPERATION

Sending a frame to the CS8900A is achieved via register manipulation in a similar manner to frame reception. First of all, the driver has to bid to send the frame, by issuing the correct transmit command to the TxCommand port, and subsequently writing the required transmission length to the TxLength port. Once this is done, we have to poll the appropriate PacketPage register until we are given permission to start

writing the frame. This sequence is necessary, as the CS8900A has internal buffer space that may already have been used for incoming frames, or may be holding outgoing frames that have not yet been sent successfully. Once write permission has been granted, the frame is written in exactly the same way as reception – LSB to 0x00 and MSB to 0x01. Again, the overall sequence of register accesses is summarised in the following tables.

Value	Port	Port Offset	Direction
TxCommand LSB	TxCMD LSB	0x04	Write
TxCommand MSB	TxCMD MSB	0x05	Write
TxLength LSB	TxLength LSB	0x06	Write
TxLength MSB	TxLength MSB	0x07	Write

Poll PacketPage BusStatus register until we have write permission. Then write

Value	Port	Port Offset	Direction
Data Byte 0	Rx/Tx Port 0 LSB	0x00	Write
Data Byte 1	Rx/Tx Port 0 MSB	0x01	Write
Data Byte 2	Rx/Tx Port 0 LSB	0x00	Write
Data Byte 3	Rx/Tx Port 0 MSB	0x01	Write
Data Byte 4	Rx/Tx Port 0 LSB	0x00	Write
etc	etc	etc	Write

Once the complete frame has been transferred to the CS8900, it is automatically sent. If the frame is smaller than the minimum frame length of 64 octets, it is automatically padded by the CS8900A with extra garbage bytes to comply with the Ethernet CSMA/CD minimum frame size specification.

An issue was discovered when polling the BusStat register during development. In the CS8900A Reference manual (Cirrus Logic document AN083), it states that permission to transmit the frame should always be given, though not always immediately. It was found however that this is not always the case, and the driver loop

was getting stuck in an infinite loop waiting for the correct permission bit to be set. To combat this issue, a count variable is included as a timeout. If this variable reaches its maximum value of 255, the send operation is aborted. This will however result in the packet being lost, but should be compensated for by a TCP retransmission, or UDP timeout.

1.12 OTHER NETWORK INTERFACE POSSIBILITIES

The two network interfaces that have been discussed in this section were chosen mainly for their standardised use and compatibility with a large range of systems. However, the uIP stack can easily use other IP aware network technologies by simply constructing an appropriate software driver.

1.13 DYNAMIC NETWORK CONFIGURATION VIA DHCP

The problem of uniquely identifying and configuring a collection of nodes on a network is not new. Every node must have a unique identifier - in the case of IP networks, this is the IP address. When communicating at the hardware level on an Ethernet network, hosts are differentiated by their hardware MAC address. This address consists of a Manufacturer ID and an equipment ID, as discussed earlier. The IEEE allocates the manufacturer portion to companies who wish to produce network products. The equipment ID is assigned by the manufacturing company, and must be unique for every item manufactured.

Local IP address configuration and allocation is the responsibility of the local network manager, who must ensure that the network is organised in such a way that no two nodes can ever have the same address. Having more than one node with the same IP causes many problems, and can lead to data loss and overall confusion.

There are a few different ways in which the allocation of addresses can be managed. The first is to use **static** IP addressing, where each node has to be configured manually. This is obviously a large administration task, and must be carried out very methodically to ensure there are no conflicts.

Another option is to use some sort of **dynamic** allocation. Using this type of scheme, a node will boot without any address, and must query the network to determine a suitable address. There are a few different protocols that can be used to achieve this, notably

- BOOTP, and its more recent counterpart
- DHCP

Both protocols above rely on using initial broadcast messages to contact a centralised database server that will broadcast a suitable address back to the client.

BOOTP is a protocol that is designed to allow a diskless node to boot via an IP network. Firstly, a BOOTP server will be discovered that will allocate an IP address, and this then allows the node to subsequently remotely load a file into its own memory in order to complete the boot process. BOOTP can work on a variety of network media, and is fully defined in RFC 951.

DHCP is a newer extension to BOOTP, and is designed for use on machines that have some form of local storage media. There are some fundamental differences between BOOTP and DHCP. DHCP introduces the concept of acquiring a network lease for a finite period of time. It is the responsibility of the client to ensure that any acquired lease is renewed with the originating DHCP server within the allocated time. Also, DHCP allows the client to acquire other IP parameters that it requires to operate, for example Subnet Mask and default router details. DHCP can also provide lists of DNS servers, although these are not needed for operation of the HCS12 IP stack. DHCP is designed to be backward compatible with BOOTP, and as a consequence uses fairly large messages containing many redundant fields. The operation of DHCP is fully defined in RFC 2131, and extra extensions are defined in RFC 2132. As the protocol operates with variable length packets, and variable options fields, there are many other vendor dependant options that servers and clients can optionally support.

1.13.1 DHCP SERVER CONFIGURATIONS

There are a few different schemes in which a DHCP server can be configured to allocate leases to different client hosts. The simplest setup is to set aside an address range that can be used solely for dynamic allocation. This "pool" of addresses can then be given out on a first-come first-served basis. For a lab of client PC's, this is a perfectly adequate solution, as most IP traffic will be outgoing, and therefore it doesn't matter which address a specific machine is allocated. However, for our system, we require to know the IP address a given node is allocated in order to communicate with it.

DHCP can solve this problem in a number of ways. As each client must have some sort of unique hardware address (48bit MAC address in Ethernet), the DHCP server can use this as an identifier, and a specific IP can be allocated accordingly. While this obviously increases network administration at the DHCP server, all administration will take place in a centralised location. All client nodes will still boot without an address, and any IP address range changes on the network can be coordinated centrally.

Another, slightly more complex method is to have an integrated DHCP and DNS server solution. In this case, the hostname given whilst discovering the DHCP lease is used as the client identifier. Once a DHCP lease is allocated, a corresponding DNS

record is created, and from this point on all communication with the node can be carried out via its hostname rather than IP address. This approach is slightly more flexible, as a fixed node to IP mapping is not required. However, the complexity lies with the communication and synchronisation required between the DHCP server and network DNS server, and also each host must store a unique hostname in firmware.

1.13.2 DHCP IMPLEMENTATION

As DHCP requires not only acquiring an IP lease, but also subsequently renewing it, the most straightforward implementation for our system is to implement the client as a state machine, which is periodically polled by the main code loop. It is important to realise at this point that until we have been allocated a lease (and subsequently received a suitable acknowledgement from the server), the stack cannot accept or send any IP packets, except broadcast DHCP messages. This is accomplished by setting the system IP address to 0.0.0.0 until a lease is obtained.

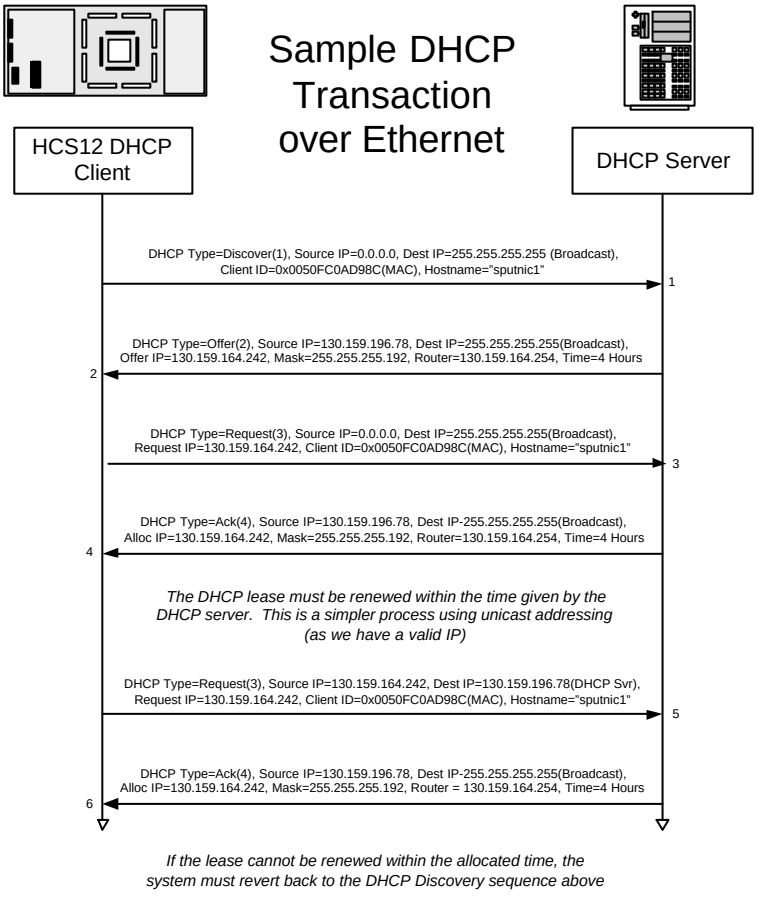
As already discussed, DHCP is an enhanced version of BOOTP which is carried as the payload of a UDP packet running on IP. DHCP uses the following UDP ports for normal operation

- Client : UDP port 67
- Server : UDP port 68

The diagram to the right shows a sample DHCP discovery transaction, and subsequent renewal of a DHCP lease. This diagram was constructed from a session between the project board and departmental DHCP server. The captured traffic is contained in Appendix VI.

As mentioned previously, the main state machine for the DHCP client is carried out by a polling function. This function takes care of

- DHCP Discovery
- Lease Request
- Lease Renewal



Before looking at the state machine implementation in detail, it is helpful to examine the DHCP send and receive functions, and the basic structure of a DHCP message.

1.13.3 DHCP MESSAGE LAYOUT

The message format of DHCP looks rather strange from the outset, and contains a large amount of redundant data fields. This is due to its backwards compatibility with the older BOOTP protocol. A DHCP message has a standard fixed format header that maintains compatibility with BOOTP, and also contains one or more variable length options fields. The header has the following format

Field Size	**Field Name**	**Description**
8BIT	Op Code	
8bit	Hardware Address type	This is Ethernet in our case
8bit	Hardware address length	6 bytes for Ethernet
8bit	Number of hops	Normally unused, and set to 0
32bit	Transaction ID	A unique number that is used in all further correspondence between a client and DHCP server
16bit	Elapsed Time	Normally unused. It defines the number of seconds that have elapsed since the node was first powered on
16bit	Flags	Also redundant - would normally contain the BOOTP message flags
32bit	Client Address	Unused in DHCP
32bit	Your Address	Contains allocated IP
32bit	Server Address	May contain server IP
32bit	Relay Agent IP	May contain a relay server IP
16byte	Hardware Address field	We populate the first 6 bytes of this field with our Ethernet (MAC) address
64byte	BOOTP Server Hostname	BOOTP specific field - we leave it blank
128byte	BOOTP Boot File Name	Again, left blank

After this standard header, a 32bit "Magic Cookie" value is inserted. This value is standard to DHCP, and identifies that the rest of the variable options in the message are to be interpreted as DHCP. The DHCP magic cookie is defined as

- 0x63825363

1.13.4 VARIABLE OPTIONS

The content and type of options used are discussed in the sections below. Each option has a standard format, allowing variations that are unknown to a client or server implementation to be safely ignored.

EACH OPTION CONSISTS OF

- 8bit option code
- 8bit option length, specifying the number of payload bytes
- Option payload data

A special op-code byte of 0xFF is reserved to indicate the end of the variable options, and hence the end of the DHCP message.

1.13.5 DHCP SEND

As we are developing the client for a low power embedded system, we are aiming to make the DHCP client implementation as simple as possible to conserve processing power and network bandwidth. Therefore, we only support the minimum number of messages to make the system work. These are

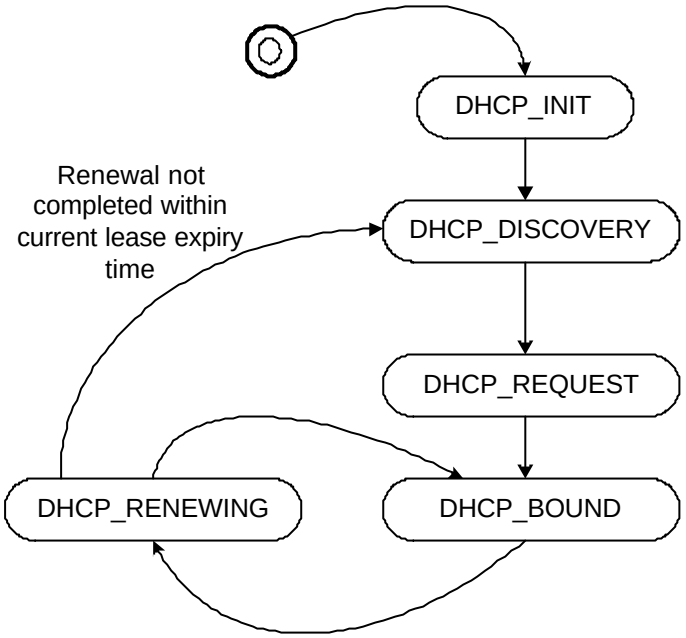
- DHCP Discovery
- DHCP Request

Most systems also support the ability to release an IP address on system shutdown and to renew an address by user intervention, but these are not mandatory in the DHCP specification, and therefore not necessary for our minimal implementation.

1.13.6 DHCP RECEIVE

Again, our DHCP client implementation requires only support for a subset of the full range of possible server generated messages and related options that may be sent to us. We specifically require reception and interpretation of

- DHCP OFFER
- DHCP Acknowledgement



1.13.6.1.1.1.1.1 DHCP State Machine

THE STATE MACHINE FOR THE DHCP CLIENT IS DESIGNED TO BE POLLED ONCE A SECOND AND IS CALLED FROM THE MAIN APPLICATION LOOP. THE STATE MACHINE OPERATION IS SHOWN IN THE DIAGRAM ON THE LEFT.

TRANSITIONS BETWEEN STATES ARE ACCOMPLISHED BY CONDITIONAL EXPRESSIONS IN THE RECEIVE FUNCTION WHICH PARSE THE CONTENTS OF INCOMING DHCP MESSAGE PACKETS, AND ALSO IN THE POLLING LOOP WHEN LEASE EXPIRY IS NEAR.

2 SYSTEM APPLICATIONS DESIGN AND IMPLEMENTATION

2.1 6HTTP SERVER

A small web server application is included with uIP. It is very limited in functionality and does not provide all the options that we require. For our system, the web server application must

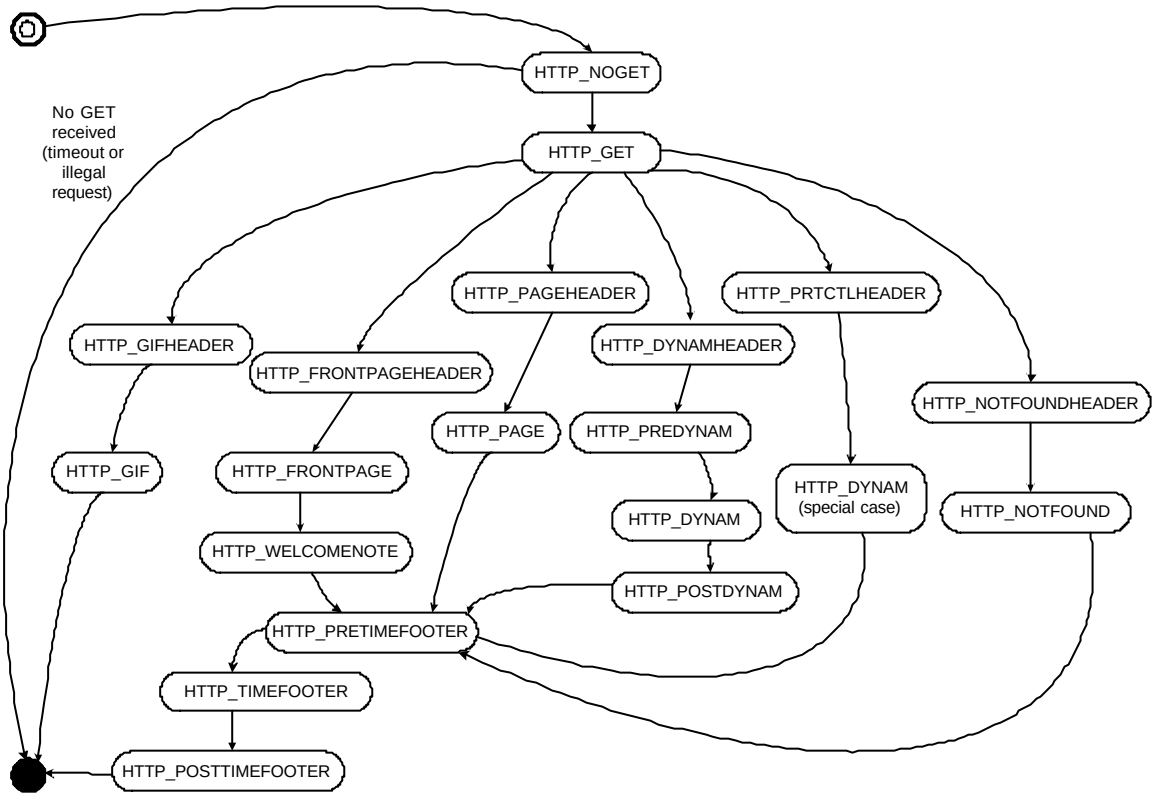
- Be HTTP 1.0 compliant
- Be non-blocking
- Be capable of publishing dynamic content
- Support a general API to allow other applications to publish content

There are two major versions of the http protocol – http/1.0 specified in RFC 1945, and more recently http/1.1, which is fully specified in RFC 2616. There are various enhancements in version 1.1 that are mainly centred on improving the performance of the protocol, for instance the operation of persistent connections. However, as we are designing our server for a minimal system having persistent connections will tie up TCP connection slots, and will add complexity to the server operation. RFC 2616 specifies that any application supporting the newer http/1.1 protocol **must** be backwards compatible with a server only supporting version 1.0. Therefore when replying to http requests, our server will report that it only supports http/1.0, and will close the connection at the end of every request. For a client to retrieve a html page containing two images from the server, three separate TCP transactions will take place (unless the image is already cached).

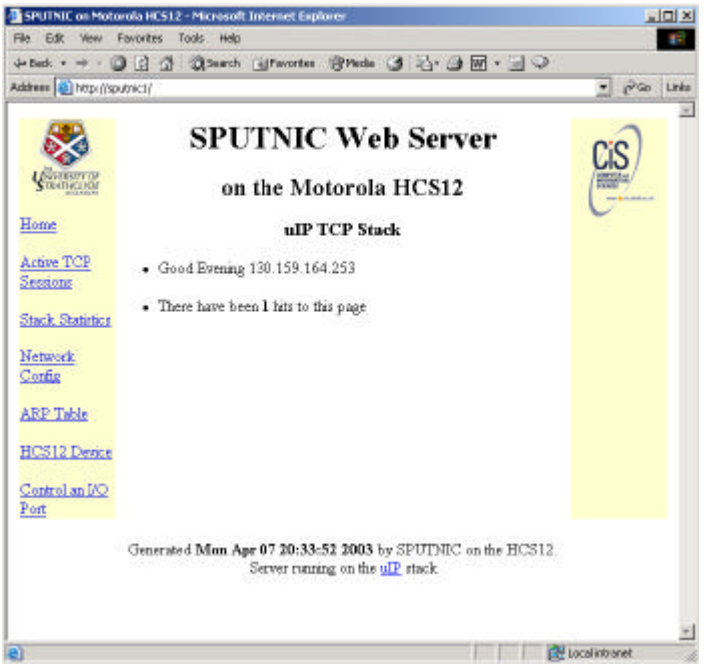
2.1.1 BASIC SERVER OPERATION

As with all modules in this project, the web server application must be able to serve content to a connection on a packet-by-packet basis, and must be able to support multiple connections simultaneously. Obviously, this requires various states to be held for each connection. This functionality is handled by a connection specific state structure, and an array of these structures are kept by uIP in a static array. The definition of the connection state is defined in `httpd.h` and as such is only applicable to the http server application. This seems restrictive, but other 3rd party applications are very unlikely to require state information to be held on a connection basis – most will hold their own application specific states. If this functionality is required for future applications, the connection state declaration should be managed by the appTCPMux module. In the http server, a connection pointer is used to pull out the application state. It is initialised to the correct array location by referencing the global connection pointer at the start of each http application call.

The operation of the http server state machine is shown in the following state diagram.



Transition between each state is dependant initially on the payload content of incoming packets. The server has to wait in the HTTP_NOGET state until a valid http GET request is received. The subsequent path through the state machine is then determined by the content of the GET request. State transitions are governed by the reception of acknowledgement packets for TCP packets sent. For static content that is larger than the MSS of the connection, more than one packet may have to be sent per state, and if no ack is received within 1 sec, the periodic timer will poll the application and a retransmission packet will be sent. To stop broken connections taking up connection slots infinitely, connections are reset if no ack is received after ten retransmissions.



HTML AND IMAGE STORAGE ON THE HCS12

With no access to any external storage medium, some thought has to go into where content that the http application requires to send out is stored. As the server is embedded and designed to be as small as possible, the content should also be kept to a bare minimum. With this in mind, each page is served out with identical html content starting and finishing each page. Every page consists of a html table containing three columns. The left column contains the university logo and links to other pages that the server supports, and the right hand column simply contains the departmental logo. The content of the middle column in the table is the responsibility of whatever function has been requested by the client. Constructing each page in this manner both saves the quantity of html code that we need to store, and gives each page a similar appearance. Also, if the content of ether of the side columns requires to be changed, this change will only need to be made once rather than in an html definition for every function. An example of this layout is shown in the screenshot on the previous page.

Bearing in mind the restrictions which the Banked memory model imposes, some planning is required to determine where both constant html data and application code is stored. As the fixed content is going to be relatively small compared to code which actually generates the pages, the fixed constants will be stored in Non Banked Flash, and application code can then be consigned to a Banked (paged) location.

2.1.2 SERVING OF IMAGES

Normally images are served from a web server either from disk or they may be cached in the server's memory. With the HCS12 acting as a standalone controller, we do not have access to any sort of storage medium apart from system memory. In order to serve out a binary image, we first must do some processing to convert it into a format that we can put into a standard C character array.

There are many utilities that allow the Hexadecimal representation of a binary file to be displayed, one such package is OctalDump, a utility found on most UNIX environments. The utility allows a file to be viewed in many formats, including hexadecimal shorts. However to put into an array we require the data to be arranged in a specific format, for example

```
char anImage[] = {0x23, 0x45, 0x02, 0x05, 0x32, 0x38};
```

To easily encode binary images in the above format, a script has been created for the Awk pattern-matching interpreter found in UNIX based systems. The script is designed for use with the OctalDump utility that can convert a binary file into hex bytes. This script will place the initialiser on the first line of its output, and insert commas between each hex byte definition on all subsequent lines and finally close the image definition on the last line.

This script is basically a time saving tool, allowing images on the server to be changed or added easily. Its use is described in Appendix II.

2.1.3 DYNAMIC HTML CONTENT GENERATION

Serving of fixed content pages and images as described above demonstrates the potential of an embedded web server, but to utilise the server to its maximum we must be able to produce html pages (or parts of pages) dynamically. Being able to produce content "on the fly" in a limited embedded environment requires some careful thought as we are constrained by both buffer sizes and CPU cycles, which we wish to keep to a minimum.

In our implementation, all dynamic content is produced via functions that are contained in the httpDynamGen module. Each function acts as a small state machine independently from the main http application, borrowing the count variable that is contained in the application state of each connection to hold its current state. When the http state machine reaches the DYNAM state, control of the connection is passed to one of the dynamic functions.

Dynamic functions will send a chunk of dynamic data each time they are called. On the next call to the function (resulting from an ack arriving, or a poll), we check the `uip_acked()` test function to see if the data that was sent the last time round has been acknowledged. If it has, then the count variable is increased. Once the variable reaches a threshold determining that all data has been sent, the function will return true, which indicates to the http server that the function is finished and the next main server state will be entered. As the function checks the `uip_acked()` test function on entry, it is important to call `uip_reset_acked()` before the first call to any dynamic function. If this is not performed, the first state in the function will be missed, as the count will be increased prematurely.

The method for actually sending the data from a dynamic function is slightly different from the fixed page states. In this situation, rather than calling the `uip_send()` function, passing a pointer to the data to be sent, content is written directly to the stack buffer, and is then sent by specifying the buffer address as the pointer. While this will copy over any incoming data that is contained in the buffer, this is not a problem in this case as we have already retrieved the contents of the GET request to be in the dynamic function. As most of the dynamic content is centred around statistical data, the `sprintf()` function is used extensively to handle the formatting of data. This allows us to easily output data in different number formats with specialised precision.

2.2 PORT AND APPLICATION CONTROL VIA HTTP

One of the main applications of having a http server running on a microcontroller such as the HCS12 is control - whether it be of applications executing on the chip or for I/O control. Built into the functionality of the http server is the ability to control port B on the controller. This port is linked to the bank of LEDs on the evaluation board.

To simplify operation of the http server, the actual port control functionality is placed into a module named httpPortControl. At present, this module contains a single function,

```
tU08 portControl(char*)
```

This handles the actual control of a port. This function takes a pointer to a URL containing a command string specifying what operation is to be performed. At present, this command must contain the following specifiers

- Port (or device) to control
- Pin levels for each pin of the specified port

The function will return true for a successful request, and false for an unknown or unsupported command string.

The command will parse the start of the command string (in a similar manner to the http server), looking for a supported command, terminated by a '?' character. If one is found, the function will assume that all content after the '?' is related to that command. At present the function only supports a command string for a single device per request. This is a sensible restriction to impose however, as there is a limit to the length of a http GET request which we can support. At present, the http server will only act on the contents of the GET request contained in the first packet of a http transaction, and is therefore related to the Maximum Segment Size (MSS) which uIP supports. This is defined by uIP as

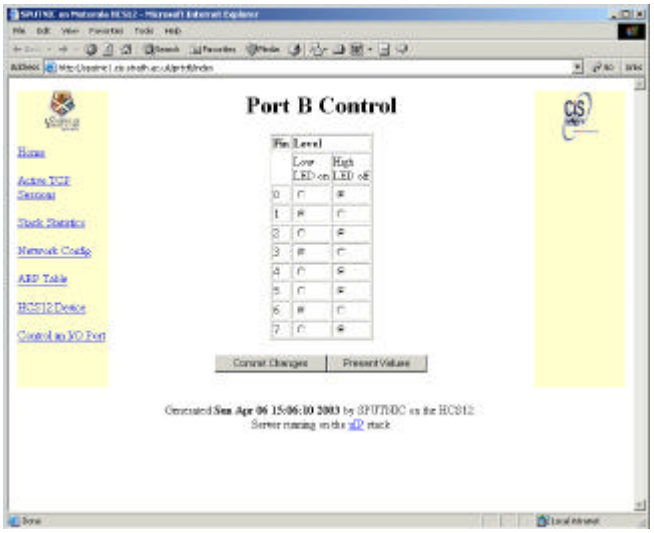
TOTALBUFFERSIZE - LINKLAYERHEADER – TCP/IP HEADER

For Ethernet, the link layer header is 14 bytes, and a standard IP header is 40 bytes.

The portControl function in our system has been constructed only to provide support for general I/O Port B, but could easily be expanded for any other system port, or more importantly for any of the on chip peripheral devices. This could allow direct control of a device via a specialised html form.

2.2.1 COMMAND GENERATION

As the port control command must be passed to the server via a http request, the easiest method of generating this command string is via a html form. The URL below is an example of what is generated from the following html form.



`http://130.159.164.242/prtctl/portB?p0=0&p1=1&p2=1&p3=0&p4=0&p5=1&p6=1&p7=0`

Constructing html forms and implementing the appropriate functionality in the portControl module can allow remote control of virtually any device or software component associated with the HCS12.

2.2.2 OTHER METHODS OF HTTP CONTROL

Although control via a web browser would be the most common method of controlling the system, in some circumstances it may be desirable to have this control performed from a different application. For many applications, this would be fairly simple, as all that is required is to open up a TCP connection on the correct listening port, and pass an http request with the correct command string implemented as a GET request. Many programming languages will provide support for opening and using http connections. For example, JAVA contains the **`HttpURLConnection`** class to handle http requests.

2.2.3 GENERATION OF THE HTML FORM CONTENT

The html form could be stored in flash, and served out as static content, similar to an image for example. However, this has many limitations, as in most cases it is useful to know the current state of the port that we are trying to control. To overcome this problem, the actual form html content generation is handled as a dynamic function, and served out line by line, with the contents of each line being determined by the current port state. This allows us to output a form which has pin levels pre selected and allows the user to be able to pre determine the status of a port before committing any changes. Having a reset button on the form also allows a user to cancel any preliminary changes made on the form before any changes are committed.

One of the other problems associated with serving a static form is that the form action tag must specify the host that the response of the form is to be sent. Having the form served statically would require the static content to be loaded onto the board with the IP address or hostname pre defined. This problem is addressed in our system by generating the form action tag dynamically, and using the current IP of the system as the host parameter. Therefore, even if the system IP address were to change (by acquiring a different DHCP lease for example), the form would still be generated with the most up to date address.

2.3 UIP STATISTICS GENERATION

uIP comes with some basic functionality to support simple telemetry. This support is built into a macro defined in **`uip.c`** called UIP_STAT. This macro provides access to the statistics structure, and if statistics are compiled into the build will allow a call to increment count values during uIP operation. The values of all supported telemetry counts are accessed by the httpDynamGen module and can be displayed on the statistics web page.

2.3.1 HCS12 DEVICE AND MEMORY STACK UTILISATION

In addition to uIP statistics, it is useful to be able to view some information about the HCS12 device itself. Natively, without an operating system there is no standard way of telling what resources the system is using, and how loaded it is. As an example, a method is constructed which allows the user to view some hard coded system parameters, and also to gain a rough idea of memory stack utilisation when the device is running.

To enable us to know how much stack we are using, we must be able to gain access to the stack pointer (SP) register, which is part of the core system block. There is an assembler instruction STS that will copy the value into a 16bit variable. To be able to use this instruction, we define it as a macro, as follows

```
#define READ_SP(x) asm sts x
```

When this macro is called, the value of SP is copied to x. Now that we have a way of determining the value of SP, we need to be able to probe and store this at specified times in order to gain an overall picture of how the stack is varying during normal system operation. As we already have an operational TimerISR, this would seem a good location to place this functionality. Two global 16bit variables are declared to represent the high and low water mark values of the pointer. When the TimerISR is called (8 times a sec), the SP value is obtained and the variables are updated if necessary.

The address and usage values are then output in the HCS12 statistics web page. In order to tell the actual amount of stack that is being used, we need to know the base address of the stack itself. This value is contained in the **`_startupData`** structure that is initialised by the linker and defined in **`basic.h`**. It is a simple matter of subtracting (as

the stack works down from the defined base value) the SP value from this base address to determine the quantity of stack bytes being used. With the high and low water mark values accessible from the dynamic html generator, the system maximum and minimum stack usage can be calculated and displayed.

2.4 TIME SYNCHRONISATION

With a standard PC, we take it for granted that the system will keep the time of day. When the system is powered down, the time is kept running on the motherboard by a battery and is therefore still correct when the PC is next switched on.

On an embedded microcontroller such as the HCS12, there is no concept of time. The processor cycles at the speed of the oscillator that is driving it, and application code will run as fast as the chip will allow. When implementing an IP stack, there are some very important timing considerations that we must consider.

While running a protocol such as TCP, retransmissions must be catered for to allow recovery of sessions when packets are lost. If we are using Ethernet as a communications medium, a local table containing IP to Hardware address mappings must to be held. If we are participating on a network that supports dynamic addressing, this table must be cleared periodically to allow for the re-mapping of IP addresses to different hardware. Also, if our node has to acquire a dynamic address, this lease must be renewed within a specific time interval.

The problems above can be resolved by creating a timer interrupt – a routine that is called at a specific time interval. This type of solution is effective, and will cater for the basic timing issues in our system. However, from an application perspective it would be advantageous to be able to determine the current time of day, and as we are working in a networked environment, several possibilities exist to allow this functionality.

Before looking at these different methods of acquiring an accurate time, it is important to understand how time is normally represented in the computer world.

In most computer systems, a 32bit unsigned number that represents the number of seconds that have elapsed since midnight on January 1st 1900 represents time. Some systems (notably UNIX) hold the time locally as the number of seconds since 1st Jan 1970. Most systems will also store a second 32bit value that contains the number of micro or nano seconds since the last second increment. The system must be able to convert these values into a format that is consistent with our everyday time. This conversion must take into account

- Time zone offsets (if applicable)
- Daylight saving time (if applicable)

- Leap Years

We already have a 32bit global variable which is incremented once a second to cater for uIP timing issues – this will be synchronised with a network time server.

2.4.1 TIME ENQUIRY PROTOCOLS

There are various standard protocols that can be used to retrieve the time of day.

Protocol	Transport	Port	Associated RFC
TIME	TCP or UDP	37	868
Daytime	TCP or UDP	13	867
NTP	UDP	123	1769
SNTP	UDP	123	1165 / 1305

NTP is a very sophisticated time protocol, and is mainly used between large servers and time critical systems. Its counterpart SNTP is simpler, but still offers some of the specialised features such as compensation for network delays between a client node and time server.

The Time and Daytime protocol operate in a simple manner – the client sends a single message and receives a single response. Time returns a 32bit numerical quantity representing the number of seconds that have elapsed since January 1st 1900, while Daytime returns a string representing the current time, such as

SAT MAR 16 20:40:59 1900

The daytime protocol does return the time value in a format which we could subsequently use on web pages, however it would be impractical to perform a time enquiry for every web page served.

As we are not implementing a time critical clock, accuracy to within a few seconds is perfectly adequate. Therefore, we will implement our time synchronisation process using the basic Time protocol.

2.4.2 TIME SYNCHRONISATION IMPLEMENTATION

A separate application module clockSync has been developed to handle both transmission and reception of time enquiry packets. This module is initialised as a UDP application, and during this initialisation it will register to receive UDP packets on a particular port. The value of this port does not matter a great deal – to be sure we are not going to potentially conflict with any other UDP service, the value 65000 is chosen. This is the value that is used as the source when sending the enquiry, and therefore the response from the time server will be destined for this port.

The module contains two functions – **`getNetworkTime()`** that is called by the main loop at specified intervals, and **`clockSync()`** which is the application call invoked by the UDP multiplexer on reception of a UDP packet for port 65000.

The outgoing function simply sets up a 32bit buffer, initialises it to zero, and sends it as the payload of a UDP packet to port 37 (Time) of the server specified in the uipopt.h file.

The incoming function copies the 32bit response from the time server into a local 32bit variable. This time is then checked against the system time and the difference is output to the debug interface. This can give an indication of how accurate the TimerISR is at giving us a 1sec increment. Finally, the system value is updated with the received time.

ANSI C defines several functions for use with time values, however the Metrowerks compiler does not provide implementations of these functions, so these were created as part of this project. Implementations were coded for

- asctime
- ctime
- gmtime
- localtime

and are contained in the fully documented **`time.c`** file.

Time updates are initialised by the main loop – the frequency of these updates can be changed in the **`uipopt.h`** file. The server IP used can also be configured. Whilst using a SLIP connection, remote time synchronisation will not normally be possible unless a routable UNIX host is available, and should be disabled.

3 TESTING AND EVALUATING THE SYSTEM

Due to the complexity involved whilst completing a project containing both hardware and software elements, a modular design with testing at every possible stage has been followed during design and implementation. This approach has therefore kept the number of variable factors that could cause potential problems to a minimum, and has allowed any problems that have arisen to be detected and fixed easily.

A number of different strategic testing methods and utilities have been used, including

- Utilisation of the developed serial debug interface
- Using the initial SLIP connection to determine the correct operation of the ported uIP stack. This connection also allowed parallel development of both the Ethernet driver software and application software – with correct operation of application modules determined via SLIP.
- Ethernet traffic capture using Ethereal, a network protocol analyser
- ApacheBench – http server benchmarking tool used for evaluating server efficiency and operation
- The http interface statistics and monitoring pages to view internal connection and ARP tables, and live configuration details.
- nmap – Network Mapper, a port scanning utility

We will now consider each of the above methods, highlighting any useful and interesting points.

3.1 THE SERIAL DEBUG INTERFACE

The development and general requirement of this interface has been discussed previously in this report. During the software development, extensive use of this feature has been made, and in some cases progress would have been virtually impossible without it.

While developing the network drivers for both SLIP and Ethernet, it was essential at times to see visually the format of byte streams and packets that were being received. Having the ability to print out values in different number bases was also extremely useful – when working at network data level we almost always wish to view data as Hex, whereas when debugging other sections of code it was more convenient to view decimal values.

Having the ability to output a buffer of null terminated ASCII data was also used extensively. For example, debugging of the http server was complicated during

development due to changes in what turned out to be a fairly large state machine. Being able to output text during each of these states allowed a greater understanding for what was actually happening in the system. Use of this debug interface is demonstrated in Appendix VI.

3.2 THE SLIP CONNECTION

Although the SLIP connection was a major objective of the project from the outset, it also served as an evaluation tool, allowing a fully functional IP connection to the board to be established before both application software or Ethernet connectivity was designed. Having this connection served as a gateway to the functionality of the http server, before the Ethernet solution was fully completed.

3.3 TESTING THE UDP IMPLEMENTATION

Along with using serial debug functions, the best method of testing that the implemented UDP transmission and reception was working was to actually make the device communicate with another node by sending UDP packets. Some small Java applications were constructed - one to transmit test packets to the device and another to receive packets. Using these two utilities along with serial debug, the UDP functionality could be shown to be working correctly. The java source code is included on the project CD.

3.4 THE ETHEREAL NETWORK PROTOCOL ANALYSER

During development of all Ethernet based areas of the project, the Ethereal analyser was used for debugging, testing and evaluation. Ethereal is an open source application which allows a user to capture traffic which is passing through the network interface of the machine it is running on. It has two main modes of operation

- Fixed interface capture
- Promiscuous mode

Fixed interface functionality was used to monitor active network connections between the host running Ethereal and the development board. This allowed the capture of full http and related TCP transactions, and also allowed monitoring of initial ARP requests and responses.

Promiscuous mode capture allows all traffic that is being broadcast on the network to be monitored. This allowed transactions between the board and other hosts to be identified, and was used to a great extent whilst developing the DHCP client. The functionality of the Time protocol transaction was also verified in this mode.

Various sample transactions captured with Ethereal are shown in Appendix VI.

3.5 APACHEBENCH

While utilities such as Ethereal can accurately determine any errors, which may be present in transactions with a remote host, they are not able to determine any performance characteristics for the system. Network latency can be determined by the "ping" utility, but what we are really interested in is the performance of both the stack in general, and the main http application.

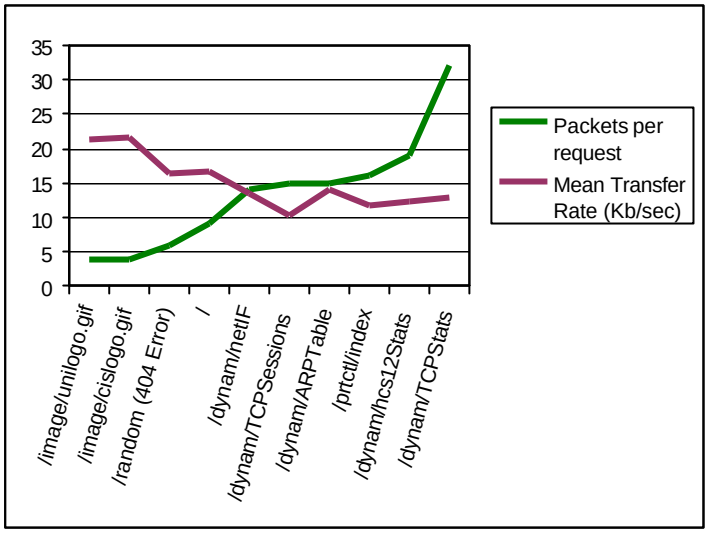
One utility that can be used to determine some simple performance data about a web server is ApacheBench. This utility comes bundled with most versions of apache – a popular open source web server. The utility can send off a specified number of page requests (at a specified concurrency level) for a given URL, and will report some simple statistics on completion. As the results of a test such as this are going to be dependant on network traffic levels and also the number of hops between source and destination, the tests were run on a Linux host connected directly to the board by a crossed Ethernet cable.

3.5.1 SYSTEM URL TESTS

Tests were performed on each of the main objects that the http server supports. The main objective of these tests is to determine if some general relationships can be observed between any of the main variable factors between each request. Each URL was tested with 100 hits at a concurrency level of 2. The table below shows various statistics that were collected from this series of tests.

URL	Bytes Per Request	Total Time Taken (sec)	Mean Requests (sec)	Mean Transfer Rate (Kb/sec)	http Data Packets Sent per request
/ (default)	797	5.31	18.83	16.68	9
/image/unilogo.gif	954	4.979	20.08	21.23	4
/image/cislogo.gif	982	5.056	19.78	21.46	4
/dynam/TCPSessions	1687	17.36	5.76	10.28	15
/dynam/TCPStats	2263	18.5	5.41	12.76	32
/dynam/netIF	1249	9.959	10.04	13.47	14
/dynam/ARPTable	1439	10.98	9.1	13.92	15
/dynam/hcs12Stats	1301	11.19	8.93	12.42	19
/prtctl/index	2030	18.36	5.45	11.59	16
/random (404 Error)	797	5.4	18.52	16.31	6

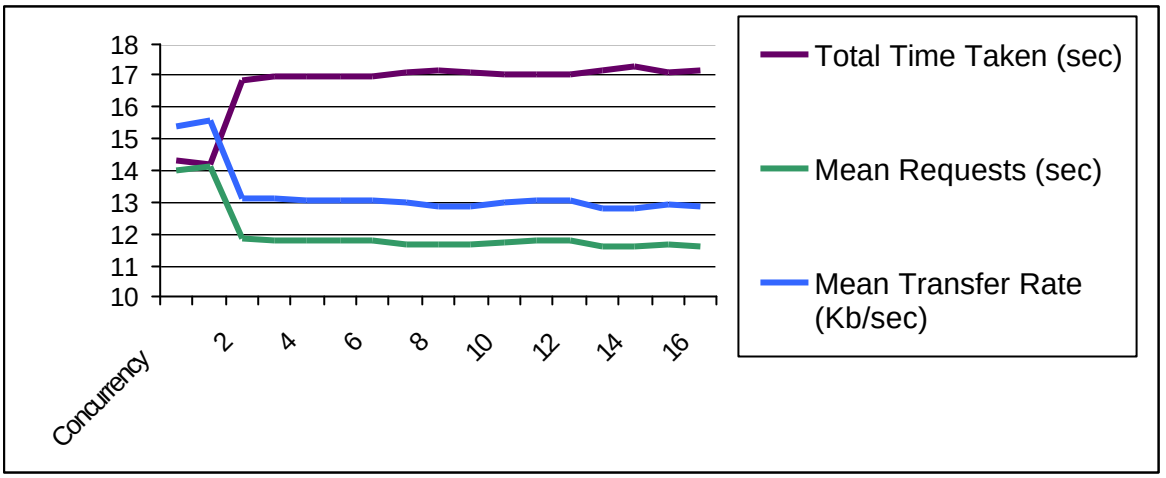
From the table above, it is helpful to compare some of the sets of data graphically. For instance, the graph below shows the packets per request against the overall mean transfer rate.



This comparison is very important, as it demonstrates how much the number of packets and associated sizes can affect the overall performance of our TCP stack which is governed by the inefficiencies caused by the "Stop And Wait" ARQ scheme imposed by not supporting TCP sliding window acknowledgement functionality. As expected, the more packets that need to be sent, the more time is spent waiting for acknowledgements, and therefore the lower the overall data transfer rate becomes.

3.5.2 CONCURRENT CONNECTION TESTS

The second type of testing which was carried out using ApacheBench was concurrent connection testing. The tests were carried out with a normalised number of connections, and the board supporting a maximum of 8 simultaneous TCP connections.



The graph above shows the relationship between the number of concurrent connections opened to the board, and the effects this has on performance.

It is surprising to note that although the board was configured to support 8 connections, the performance levels drop as soon as more than one concurrent connection is established. It is also interesting to note that when more than one connection is in operation, the performance levels remain fairly constant – even if many more connections are sent than the board supports. This seems strange at first, but actually is the expected result. If all connection slots on the board are in use, the stack will send a TCP Reset packet to the client to indicate that a TCP connection is not available. The client TCP stack will retry any failed connections until a specific time threshold is reached, in which case the connection is deemed to have failed. If this threshold is large enough for all connections to be processed within the given time, the average time and transfer rate should be approximately equal across all connections.

3.6 HTTP INTERFACE MONITORING AND STATISTICS

Once the functional operation of both the protocol stack and http serving application were implemented, a series of dynamic status pages were added to the server. This allows a user to access live statistics and configuration data from the device. Having this feature allows access to internal stack statistical data and tables that cannot normally be viewed. This feature was used to check the correct operation of the ARP table, and also gave an indication that TCP session states were being handled correctly. Network checks were also performed which the board correctly rejected. For example, attempts to connect to ports that the board was not actively listening to, and also connections to open ports with protocols other than http over TCP. The stack correctly denied all of these illegal requests, and the telemetry data on the stack statistics page correctly indicated that the illegal attempts had been made.

3.7 NETWORK MAPPER TESTING

The nmap utility was used to check that the system was only listening on TCP port 80 (http). The test was successful, showing only http port 80 to be accepting TCP connections. The utility results are shown below.

```
Starting nmap V. 2.54BETA31 ( www.insecure.org/nmap/ )
Interesting ports on sputnic1.cis.strath.ac.uk (130.159.164.242):
(The 1553 ports scanned but not shown below are in state: closed)
Port       State       Service
80/tcp     open        http


Nmap run completed -- 1 IP address (1 host up) scanned in 23 seconds
```

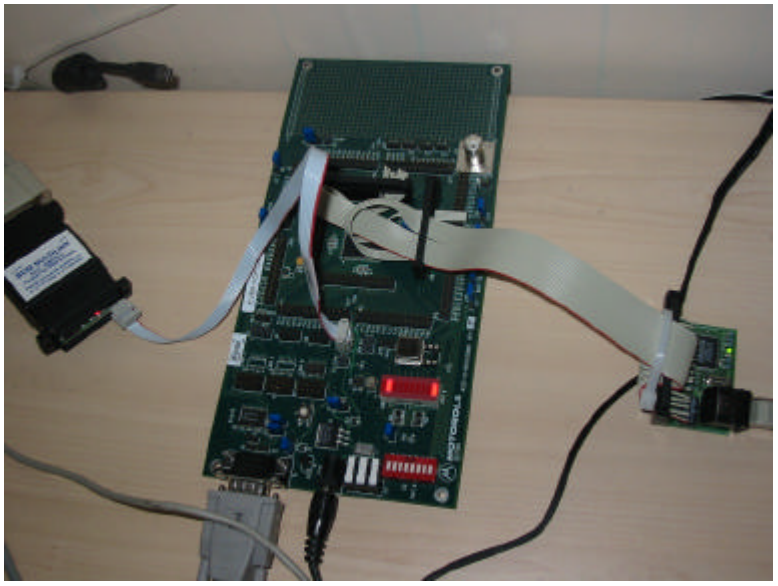
4 CONCLUSIONS

In summary, we have developed a stand-alone interface between a standard IP network and a sensor system (See figure below). This is supported by a Motorola HCS12 microcontroller with serial line and Ethernet interfaces. A micro webserver has been deployed on this device to support configuration and access to the I/O pin set of the board. This has been deployed and tested (as described in deliverable D11) in a sensor system supporting some 100 sensors. This has been operating continuously for over 2 months.

We have subsequently bought the necessary hardware elements and housings to package this system for deployment to other partners. This process is delayed due to limited availability of the Ethernet boards.

However, this device provides a very simple, low cost route to deploying a sensor system for *smartening spaces*. We estimate a cost of £250 per box that compares very favourable to the high cost alternatives (also investigated in D11) such as LonWorks. In this document we describe the development process. We have already had over 20 request for this system and specification from other researchers in the Pervasive Computing domain.

One of the limiting factors in developing many of the projects within DC has been access to sensor systems. This work complements that of Smart-ITS and provides the community with a coherent sensor interface.



The Sensor Gateway.

Glossary

ADC	Analogue to Digital Converter
ARP	Address Resolution Protocol
ARQ	Automatic Repeat reQuest
ARPA	Advanced Research Projects Agency
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
BDLC	Byte Data Link Control
BDM	Background Debug Mode
BOOTP	Bootstrap Protocol
CAN	Control Area Network
CGI	Common Gateway Interface
CSMA/CD	Carrier Sense Multiple Access / Collision Detect
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EEPROM	Electrically Erasable Programmable Read Only Memory (E^2PROM)
EISA	Extended Industry Standard Architecture
EVB	Evaluation Board (Development Board)
FTP	FILE TRANSFER PROTOCOL
HTML	HYPER TEXT MARK-UP LANGUAGE
HTTP	HYPER TEXT TRANSFER PROTOCOL
ICMP	Internet Control Message Protocol
IEEE	Institution of Electrical and Electronic Engineers
IP	Internet Protocol
I/O	Input / Output
ISO	International Standards Organisation
LAN	Local Area Network
LED	Light Emitting Diode
MAC	Media Access Control
MSCAN	Motorola Scaleable Control Area Network

MSS	Maximum Segment Size
NFS	Network File System
NIC	Network Interface Card
NTP	Network Time Protocol
OSI	Open Standards Interconnection
PDU	Protocol Data Unit
PoE	Power over Ethernet
PPP	Point to Point Protocol
PWM	Pulse Width Modulator
RAM	Random Access Memory
RFC	Request For Comments
ROM	Read Only Memory
RTOS	Real Time Operating System
SCI	Serial Communications Interface
SDI	Serial Debug Interface (Earlier version of the BDM)
SLIP	Serial Line Internet Protocol
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SNTP	Simple Network Time Protocol
SPI	Serial Peripheral Interface
SPUTNIC	Single Processor Unit Together with Network Interface Chips
TCP	Transmission Control Protocol
TQFT	Thin Quad Flat Pack
UART	Universal Asynchronous Receiver / Transmitter
UDP	User Datagram Protocol
uIP	Micro (µ) Internet Protocol
URL	Uniform Resource Locator
UTP	Unshielded Twisted Pair
WAN	Wide Area Network
WWW	World Wide Web