MATRIX | EBLOCKS 2

Embedded Internet Communications

CP4895

MATRIX

www.matrixtsl.com

CP4895

Embedded Internet

Course Notes

Contents

1 Getting started

The following information is designed to aid you in getting the E-blocks2 internet trainer up and running.

1.1 Required hardware

This document is designed for use with the BL0531 or BL0535 E-blocks2 internet training kits.

1.2 Test routine

Test routines are available for the E-blocks2 boards on the E-blocks2 support section of the Matrix website: www.matrixtsl.com

1.3 Required software

The example programs supplied on the Embedded Internet training CD require Flowcode V8 or later to be installed on the host PC.

1.4 Additional software

It is suggested that a network traffic analyzer be used to aid in learning TCP/IP, such as Wireshark. The latest version can be freely downloaded from the internet.

1.5 Documentation

Documentation for the various E-blocks2 boards and other items supplied as part of the training kit is provided on the E-blocks section of the Matrix web site: www.matrixtsl.com

1.6 Examples and exercises

The Embedded Internet training CD contains a folder of example and exercise programs that can be used in conjunction with this document

1.7 Disclaimer

The information contained here is correct at time of going to press, but may be superseded or changed at a later date. If this information is superseded a revised Getting started document will be issued with the E-blocks2 Embedded Internet training kit.

2 Introduction

The Embedded Internet training course is designed to introduce you to the concepts required to understand the communication protocols generally referred to as TCP/IP, including the Ethernet, TCP, IP, UDP and other protocols.

This course is carried out using Flowcode V8 or later and uses the TCP/IP component which has a number of properties and macros to allow the student to build up the system they require. This allows students to learn about TCP/IP without getting bogged down in the problems of programming in C or a lower level language.

2.1 Using this course

This course goes through the various layers of TCP/IP starting with the lowest physical communication layer and moving up to application data communications. The course covers the various TCP/IP layers and protocols and the Flowcode TCP/IP component modes used to implement them.

A number of exercises are included along with notes on how implement the exercise in Flowcode. For the first few protocols the notes are relatively extensive, forming a kind of walkthrough of the exercise. For later exercises the notes become more geared towards explaining specifics and how the protocol differs from those previously covered. By that stage the student should have sufficient understanding of the basics involved to complete the task without extensive notes. Example solutions are provided for the exercises for demonstration, discussion and use as a starting point for further programming.

2.2 Who this course is aimed at

This course is aimed at two main groups: Electronics technicians seeking to use TCP/IP communications and Network technicians seeking to understand TCP/IP data structure.

- Electronics technicians will generally have some experience with Microcontrollers, and will be mostly concerned with implementing TCP/IP communications from a practical point of view – i.e. how to create programs and send data. The general goal of the Electronics technician will be to send messages.

Network technicians and Computer will be more concerned with gaining an understanding of the communications process itself i.e. datagram’s and frameworks, and how this relates to the data viewable with network analyzer tools. The general goal of the Network technician will be to understand and debug messages on the network, and to troubleshoot communications errors.

Both groups are catered for in equal measure, with both the theory and the practical aspects being covered. It will be up to the teacher involved to place the emphasis during teaching.

3 TCP/IP basic concepts

3.1 What is TCP/IP?

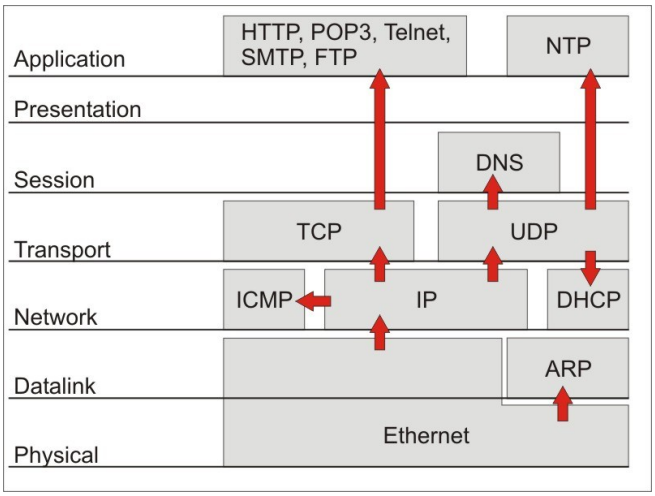
TCP stands for **T**ransmission **C**ontrol **P**rotocol. And IP stands for **I**nternet **P**rotocol. Some other common acronyms that we will be using throughout the course include: MAC – **M**edia **A**ccess **C**ontrol, UDP – **U**ser **D**atagram **P**rotocol, ICMP – **I**nternet **C**ontrol **M**essage **P**rotocol and ARP – **A**ddress **R**esolution **P**rotocol. Other acronyms will be introduced as they occur, and can also be found in the glossary at the end.

We can refer to TCP/IP in both a general and a specific way. The general term TCP/IP is used here to refer to the messaging system that transfers data between computers via the internet. The end result can be Email, web pages, or a data file. The process of transferring the message will involve a number of layered communication protocols. One of these can be TCP. One of them could be IP. And there are others – Ethernet, ICMP, UDP etc. This is where the specific definition of TCP/IP comes in, or rather the specific definition of TCP and the specific definition of IP. TCP and IP are two of the more important protocols used in internet communications, hence the general term TCP/IP to describe the whole process.

Note: Here we will use *TCP/IP* to refer to the general system, and *TCP* and *IP* to refer to the specific protocols.

3.2 OSI layers

TCP/IP is a multi-layered system with higher layers becoming increasingly abstracted from the serial data communications of the first Physical Layer. At the Physical and Datalink layers the prime consideration is shunting the data to and fro between specific nodes – usually a PC and a hub or router. The Network level introduces systems to specify the end receiver thus allowing the network to decide where data goes and how it gets there. Built on top of these the higher level protocols, such as TCP and UDP, are more concerned with getting the data to the computer applications that use the data. Even higher up the level structure are the applications that can either handle the data for us, or start the procedure of sending or requesting a message for us.



In the higher level layers more and more of the data transmission process becomes automated. For example users do not need to know how the routers will find the correct recipient for an IP data transmission – they only need to know that the routers can. Provide the correct IP address and the data will get there whether it's across the room, or across the world.

In the higher level layers more and more consideration is given to error detection and transmission sequencing, and it is likely that the receiving system will expect a two-way dialogue with data being

passed in a specific sequence, and with error codes being returned at specific times. Checking for these codes and dealing with any errors means a corresponding increase in data processing.

The Application layer protocols are the parts that we normally associate TCP/IP with – web browsers and email programs etc. Many would think of them as TCP/IP, but in truth they just manipulate and display the raw data sent by TCP/IP. What you see in an email program is not SMTP (Simple Mail Transfer Protocol), but a visual representation of the data sent via TCP/IP in the SMTP format.

3.3 TCP/IP component modes

The TCP/IP component has four modes that can be used as entry points into the OSI model.

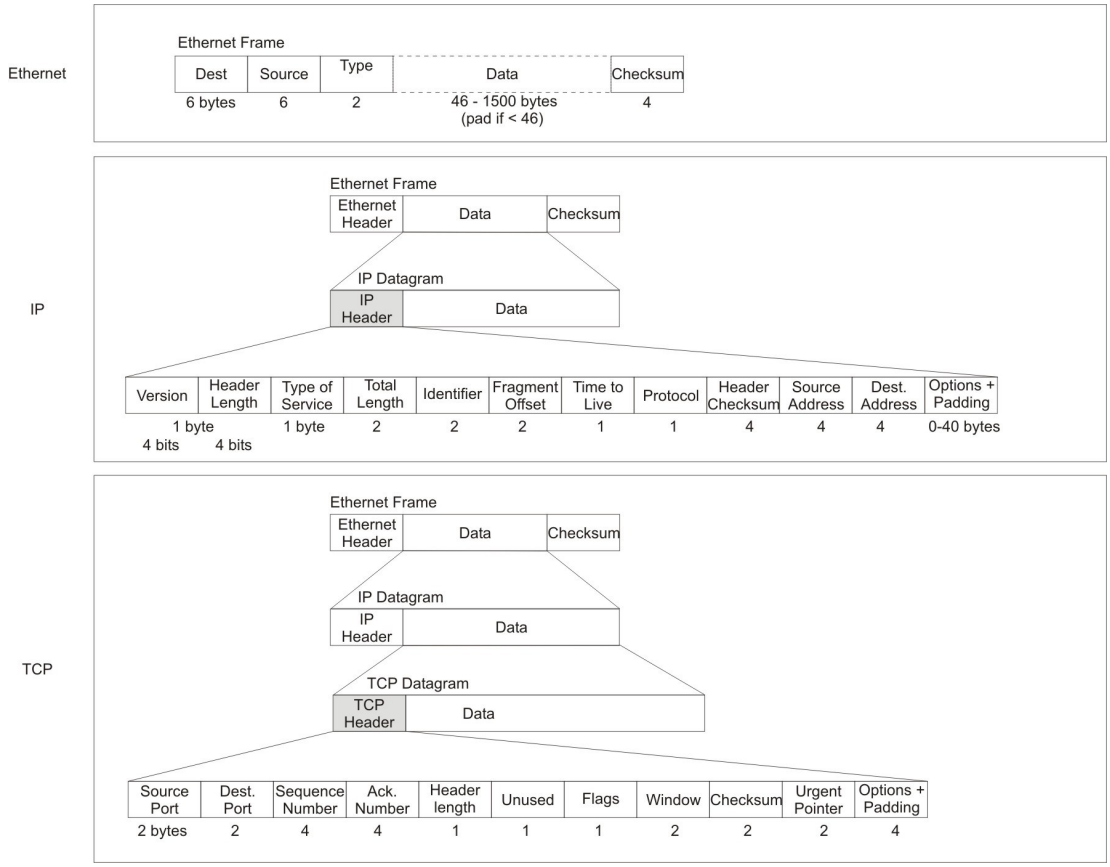
- MAC mode Brings the user in at the bottom physical level.
- IP mode Brings the user in at the Network level. Automatically handles the Physical layer procedures.
- UDP mode Brings the user in at the Transport level using the UDP protocol. Useful for applications such as DNS or DHCP that are built on UDP.

TCP mode Brings the user in at the Transport level using TCP datagram's. Useful as a starting point for applications such as SMTP or HTTP that are built on TCP.

The modes automate the levels below them, so implementing the TCP mode for instance will handle the Network (IP) and Physical (MAC) layers automatically. Headers are also handled automatically in all but MAC mode allowing you to concentrate more on the data processes.

3.4 Frames and datagram's - Messages within messages

The most basic message consists of a stream of serial data with a header section giving information about the data. As all messages sent in TCP/IP need this base level, a mechanism is needed to adapt this to serve the many purposes and formats needed for the different protocol layers. The mechanism chosen was to put the message meant for a higher protocol layers inside the data section of a lower layer. A TCP message is put into the data section of an IP message which is put inside the data section of an Ethernet section. As the message gets passed up the layers these outer wrappings can be discarded until the message finally gets to the level that is meant to handle it.



The graphic shown above displays the format for a number of different protocols. The graphic also nicely shows how datagram's can be layered inside other datagram's.

The Ethernet physical layer

With the Ethernet mode we are dealing with direct communication between one device and another. Using a mail system as a metaphor the Ethernet communications would be the equivalent of office pigeonholes where you can put mail for others and collect mail sent to you. However for many TCP/IP systems we need to be able to communicate to anywhere on the network no matter where it is. IP mode comes in here providing the stamped addressed envelope, which can be delivered to the address written on it, as long as that address exists. The routers are like the post office, with its sorting houses delivery vans and postmen. They receive the post, check the address and send it on to the next part of the system and so on, until it reaches its destination. Whether it goes by train, by van or by plane will be up to the post office and how it thinks the mail should best be handled. Similarly the routers will decide which route to send the messages. It may differ from message to message, but just like it doesn't matter if a letter went by train or not, we don't need to know how the routers did their job, only that they did.

Onto the network with IP

The IP datagram (the word 'datagram is derived from 'telegram' no doubt, adding to the mail system metaphor) is nestled inside an Ethernet frame to allow the physical sending of the message onto the network. The IP datagram contains a header file and a data packet. Just as the IP datagram (the envelope) is tucked inside an outer wrapper Ethernet frame, the data section is in itself a message (the letter itself). This message can be in a number of different formats e.g. TCP, UDP, and ICMP etc.

A real life packet would have details such as address to be delivered to, return address, postage type (Airmail, 1st class etc.), customs clearances, post stamps etc. An IP datagram has the same kind of information contained in its header section: Source IP address, Destination IP address, Protocol etc.

IP differs from Ethernet in that it is not limited to a direct connection. Ethernet can only pass the message on to another MAC device; generally this will be one in the same Local Area Network. IP however goes out into the wilds. The IP address is examined and the datagram passed on either locally or globally depending on where the IP address is. The receiving devices know how to decipher the IP

address and how to determine where to send it to next. There will be a whole host of systems swinging into action to aid the packet on its journey, but we won't see them, just like we don't see the baggage handlers and drivers that shuffle the mail around the country. These background details don't concern us, only the fact that the message will get through – unless there is some kind of problem.

Servers can be down. Addresses can be wrong. The message may not always get through on such occasions. The senders address is part of the data sent so should the destination be blocked or missing an error message can be returned to the sender to inform them of what the problem is.

3.4.3 Delivering data with TCP

At the TCP layer the contents of the message are finally delivered to the recipient; in this case an application that can read, understand and reply to the data sent. The data may be an email message, or a web page. Whatever the data contains it's the job of TCP to deliver it into the right hands. At higher levels like TCP this will often be in the form of a two-way dialogue with distinct communication sequences. Kind of like a courier having to go to reception and fill in a delivery form.

An important point to note here is that the contents of the message are irrelevant; it's the communication that counts: what the email or web page actually says doesn't matter to TCP – the important issue is getting the data to the application, and letting the application worry about content, fonts, formatting and all the other things that need doing to the data. It's the job of TCP to get the data there, that's all.

3.4.4 Applications and content

The application layer includes protocols such as SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol) and HTTP (HyperText Transfer Protocol). These are the protocols for formatting the raw data that becomes email and web pages etc. in the hands of the applications. These protocols may also contain transmission sequences and error codes for the TCP protocol to follow or act upon. To finish off our mail analogy this is the point at which the letter is opened and read, the brain receiving and understanding the contents of that letter. TCP/IP has done its job and the message has got through.

3.5 Other Protocols

TCP is the main protocol used for communicating as it generally contains application data, as opposed to Ethernet and IP which are generally the wrapping paper not the final message. But there are a number of other protocols that we can use, including a couple that are used with Ethernet and IP communications: these allow us to demonstrate the lower level protocols in action. These lower level protocols are often specialized protocols designed to aid in certain common TCP/IP tasks such as address verification. Thus they are useful to understand in general. The other protocols we will be dealing with in this course include:

3.5.1 ARP – Address Resolution Protocol

ARP is used with Ethernet mode to help verify MAC connections. This makes it a useful tool for physical level error checking. As it forms the data packet of an Ethernet frame it is a useful protocol for demonstrating the Ethernet mode.

3.5.2 ICMP – Internet Control Message Protocol

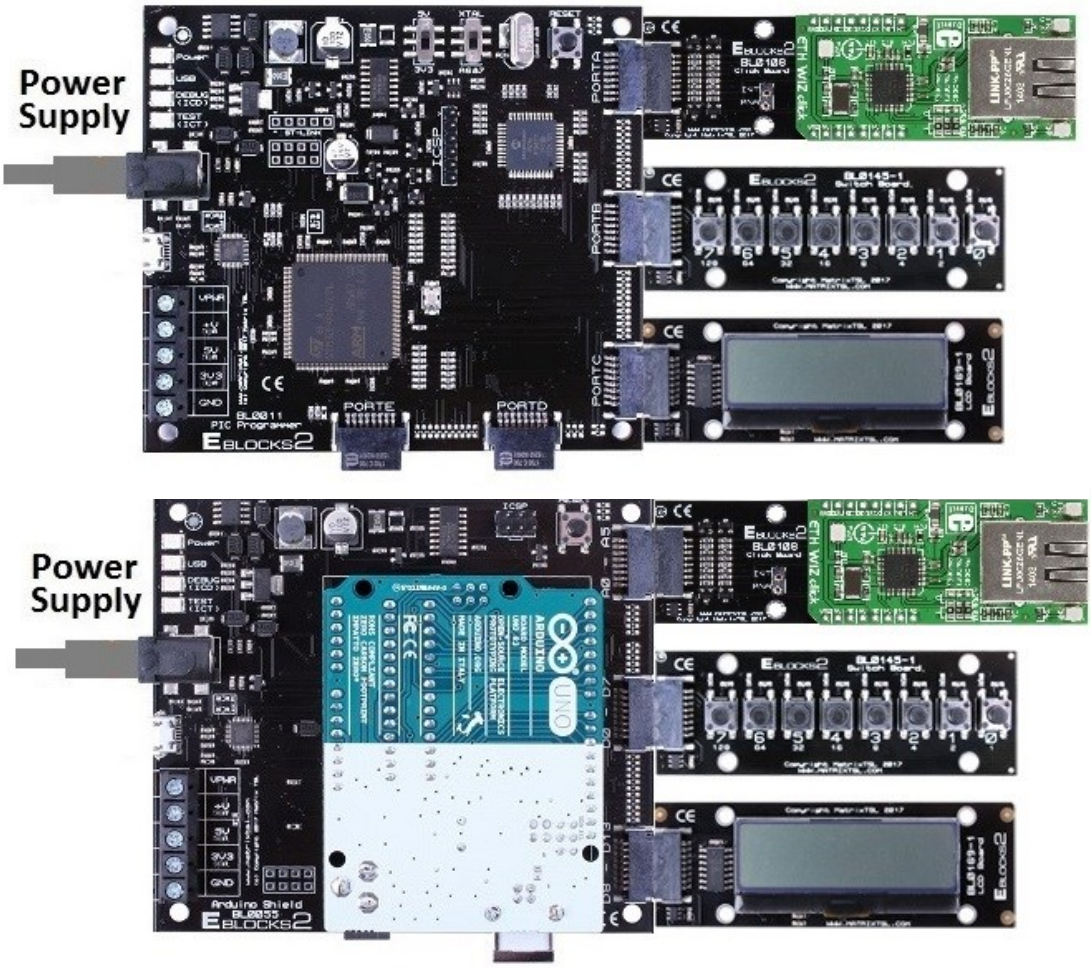
ICMP is often used to verify IP addresses as it can contain error checking information. The message is sent out and, if received, echoed back like a sonar ping. You may have already come across a basic implementation of the ICMP protocol in the command line program PING.

3.5.3 UDP – User Datagram Protocol

UDP is similar to TCP in that it contains application data. But it lacks the dialogue and error checking systems used by TCP applications. It is a more direct form of communication sending the message straight to the recipient. UDP can be very useful for fast communications between two known systems. However if the recipient isn't there, or isn't listening, UDP transmissions will be lost.

E-Blocks2 **setup**

The typical system set up of the E-Blocks2 internet trainer is illustrated below for both PIC and Arduino versions



Port connections

BL0011 (PIC) Port	BL0055 (Arduino) Port	E-blocks2 board
A	A0-5	BL0106 Click board with ETH WIZ Click
B	D0-7	BL0145 Switches board
C	D8-13	BL0169 LCD board

Practical limitations for the microcontroller

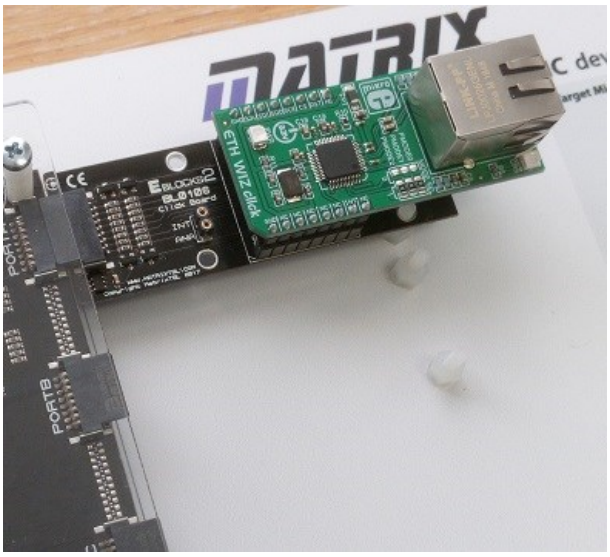
You can use the microcontroller to send and receive TCP/IP communication data such as Email or a web page. However, there is a limit to the size of the data that can be sent. Large files and attachments or images would simply be too big to store in the microcontroller's own memory. Whilst adding memory devices would give some extra storage space you must accept the microcontroller's memory limitations when developing your projects. In a similar way you will need to accept the display limitations of the microcontroller system. A standard 20 character 4 line LCD display is just not as graphically sophisticated as a modern PC monitor. Whilst email messages, for example, can be displayed they may need to be shown line by line, and without any formatting.

Also it may not be possible to perform multiple functions and use multiple protocols and levels of communications in the same program due to memory limitations. The microcontrollers have limited memory and needs some spare capacity for the program to actually function. The size and complexity of the code generated to support the TCP/IP component macros can take up a large part of the ROM and RAM space, leaving you with a much reduced amount of memory in which to develop your program. Unfortunately you may only learn that your program is just too big to fit into the memory when the assembler tells you that you have run out of space.

The internet board

The E-blocks2 Click adaptor and ETH WIZ Click internet board is designed to be used in conjunction with the range of Matrix E-blocks2 programmer boards.

The Embedded Internet training kit is available with either Microchip PIC processor or Arduino Uno processor boards, BL0531 and BL0535 respectively.

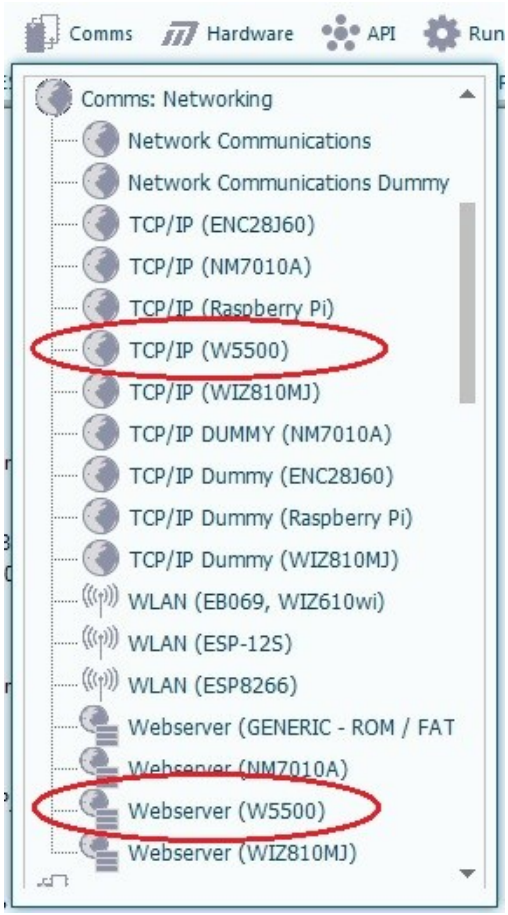


Internet board setup

For most examples and exercises in this manual the internet interface board is attached to Port A of the processor board and is required to be connected via the Ethernet cable to other equipment such as a PC or network switch. The Firewall example uses two internet interface boards.

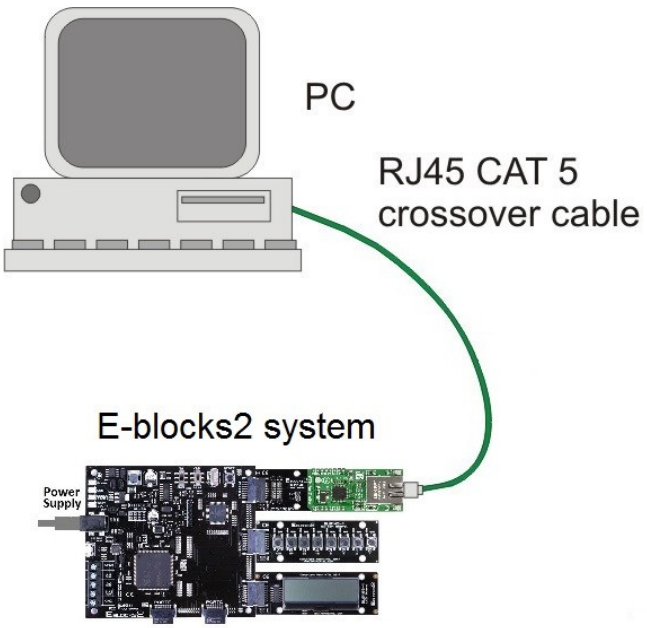
The development centre panel has support pillars fitted to both port locations to prevent movement of the board due to attachment of ethernet cables. Care must be taken to avoid damage when fitting and removing the
E-blocks2 Click board.

4.5 Flowcode and the TCP/IP and Webserver components



The TCP/IP and Webserver components can be found in the Comms section of the Component toolbar. These are identified by the W5500 device that is used on the ETH WIZ Click module.

A reasonable level of competence in creating Flowcode programs is assumed. It will be assumed in this document that the user is familiar with adding and editing macros and dealing with component properties. If the user requires further training with Flowcode we recommend that they review the tutorial files, and work through a free Flowcode course (available at www.matrixtsl.com).



4.6 Basic direct connection setup

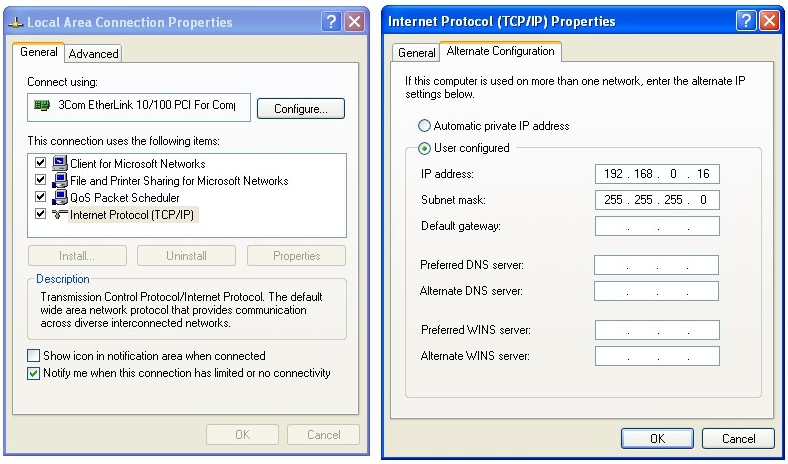
The internet board can be connected direct to a PC using a standard RJ45 CAT 5 cross-over cable which is supplied in your internet solution.
.

You may need to configure the PC to have a static IP address, and to not use proxy servers.

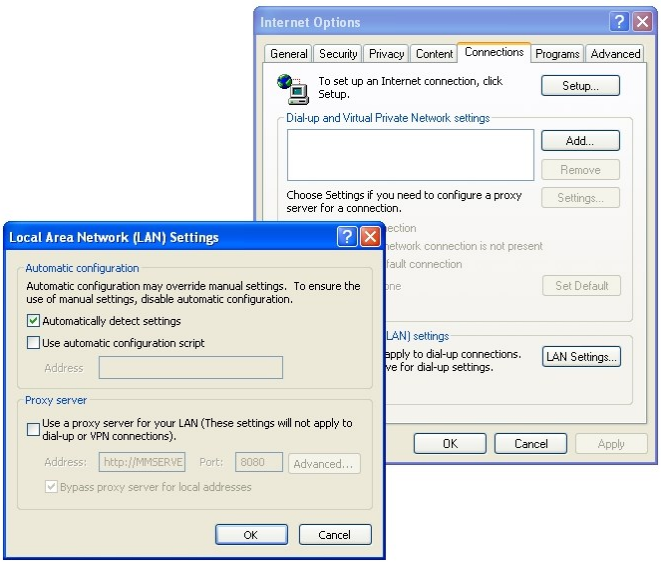
The IP address can be set up in the Network Connections Dialog screen. A 192.168.0.xxx address such as 192.168.0.16 is generally recommended for use with the internet board. Please note down the original IP address details before making any changes. If in doubt ask a network technician for aid.

Note that example files will require the programs target IP address setting to one that you have selected here.

Here we have set the IP address for this PC to 192.168.0.16.

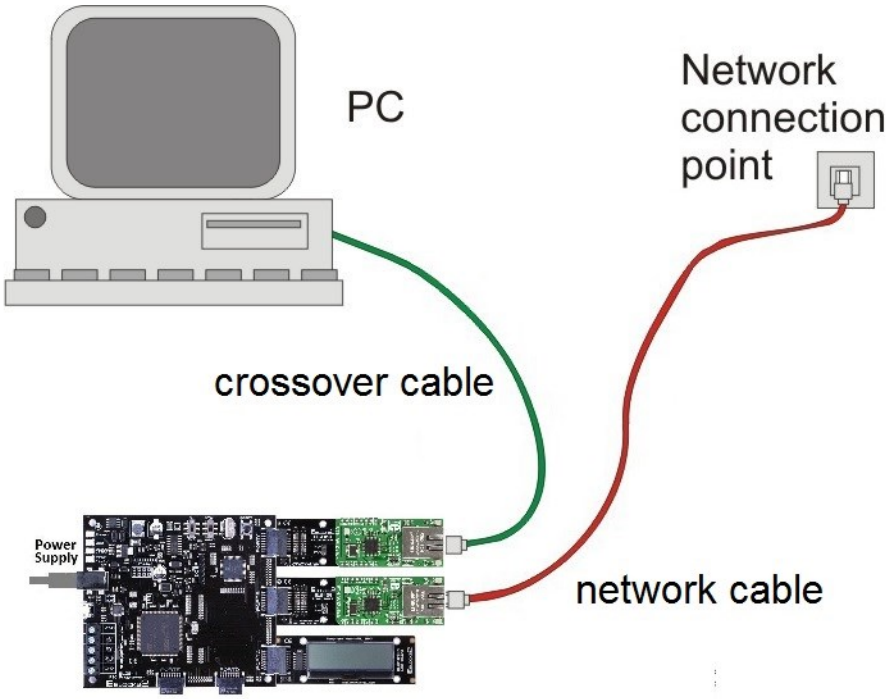


The LAN proxy server setting can be found in the Internet Options Control Panel.
Ensure that Use a proxy server is not ticked.



Dual Ethernet board system

For some applications, such as Firewalls you can connect two internet boards to the programmer board.

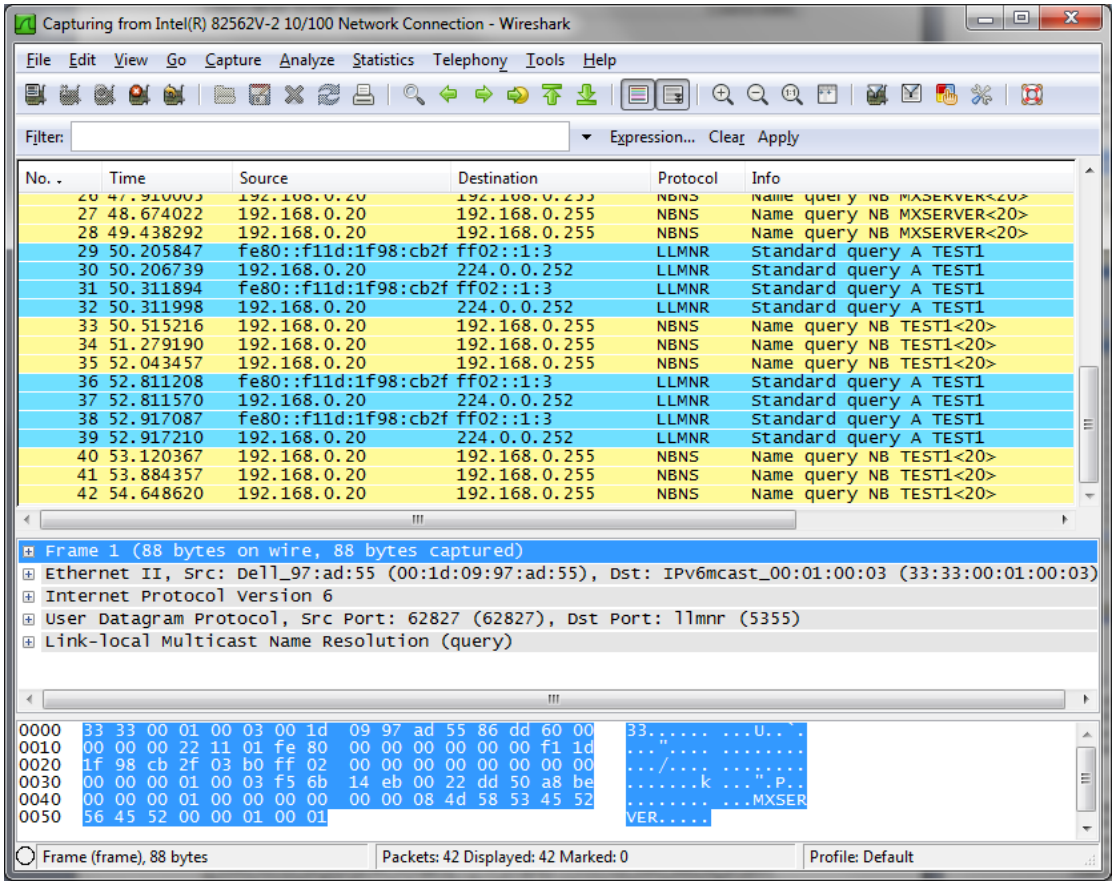


4.8 Network monitoring software

A useful tool for developing and debugging TCP/IP programs is a network traffic analyzer program. These are programs that allow you to monitor the network for messages being sent using the various TCP/IP protocols such as TCP, IP, UDP etc. They can allow you to check the data that your program is sending or receiving to see if it is correct. They may also provide useful error information about any malformed messages that you are sending.
There are a variety of programs available such as Wireshark.

We will be discussing using Wireshark to help debug programs later in the document.
The latest version is available from the Wireshark website: http://www.wireshark.org/download.html

The following image is taken from Wireshark. Note that the various sections allow you to select messages to examine and to see the various protocols contained in the message.
Data is shown about the various elements in the protocol e.g. “Type: IP (0x0800)” or the “Src Addr” IP address details. The various levels and sub categories for the data involved can be expanded as well.
The original raw data is also available at the bottom in both Hex and ASCII format.

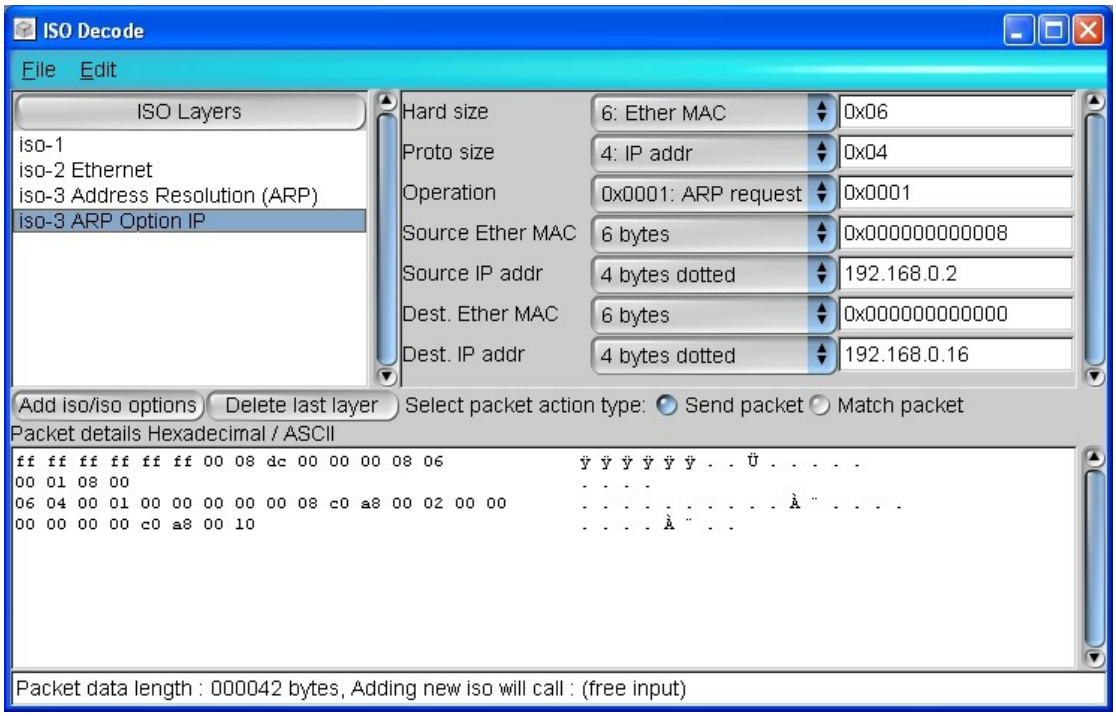


Debugging with Network traffic analysis tools

If the program works first time, then we are all happy. However should there be an error, whether it's our fault or not, then things get tricky. We are pumping data onto the network (or think we are) but cannot see it. This is where Network traffic analyzers such as Wireshark come into play. These tools can monitor record and display network traffic.

Packet injectors

The opposite of a network monitoring tool, these software applications allow us to create and send custom packets. This allows us to create test messages which our programs can receive and process. This can also help with debugging as we can modify the messages sent to see what affect they have on the program, useful when trying to figure out if it's the wrong data, or a faulty program.
There are a variety of programs available; the link below offers a variety of possible injection tools:
http://wiki.wireshark.org/Tools#Traffic_generators



Debugging with Packet injectors

Packet injectors allow you to create a packet or message that can be sent onto the system. Packets can be saved for future use, and can be edited and sent again. This allows you to create test packets that can be sent to test programs. Should you suspect an error you can then modify the packet to test out the error and to test potential solutions.

Packet injectors require a good working knowledge of TCP/IP as all stages of the packet require building. However a network traffic analyzer such as Wireshark can be used in conjunction with the Packet injector to find much of the required information for packet building.

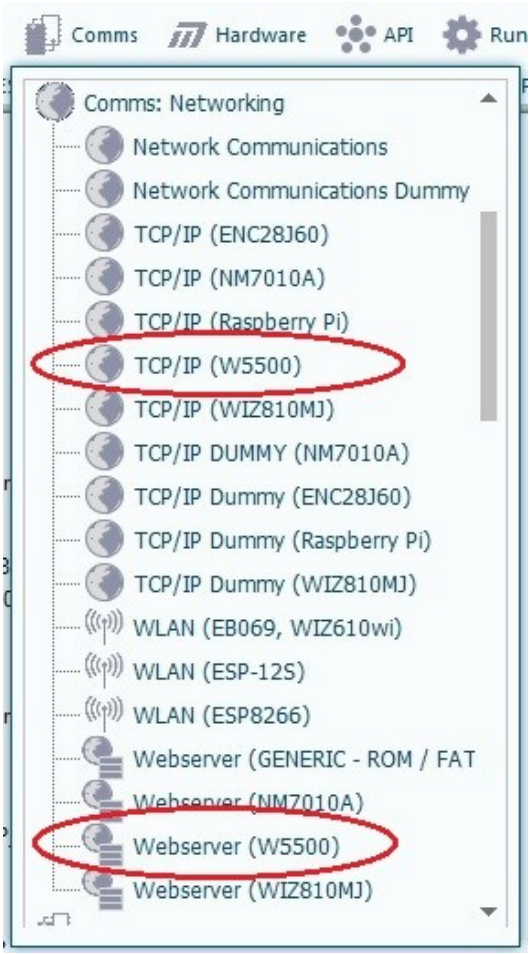
Server configuration

If you are connecting to your local network and encounter difficulties you may need to configure the network to accept the messages produced. You will need to consult your network supervisor in order to make any configuration changes.

Areas that may need configuring include:

- Checking IP settings for the internet boards to avoid conflicts
- Test email addresses for SMTP messaging.
- Firewall settings may need checking if they affect the messaging. This may involve settings on the PC, the Server and on any software or hardware firewall.
- Authorization may be required for installing Network analyzer tools
- Configuration settings to allow the internet boards (with their own IP addresses) to be visible to the outside world.

5 The TCP/IP component



To add a TCP/IP or Webserver component to a flowchart click on the Comms menu header in the Component toolbar and select the appropriate icon from the drop-down list. Look for the W5500 based versions.

TCP/IP component icon

Webserver component icon

The TCP/IP and Webserver components have Connection properties that need setting up in Flowcode and these are listed on page 12 of this document.

TCP/IP and Webserver properties

Details of the TCP/IP and Webserver component properties are covered in the component Help files, available via the Help button on the components properties pages.
Most properties can be left at the default values unless the default values clash with other hardware settings – e.g. an IP address conflict. For examples involving more than one board the MAC address and IP address settings will need to different for each board.

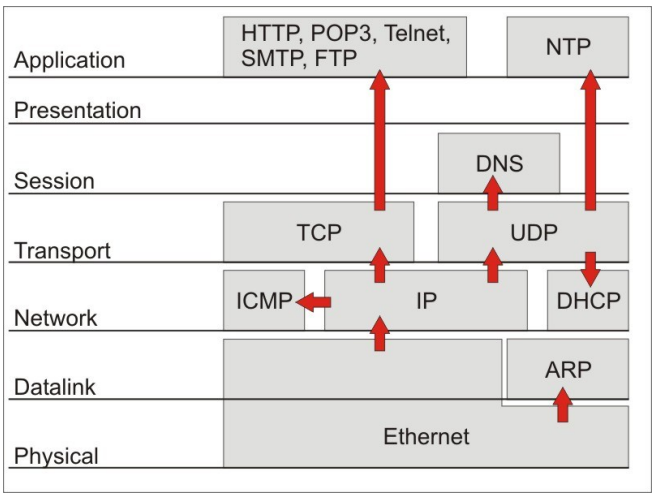
TCP/IP and Webserver component macros

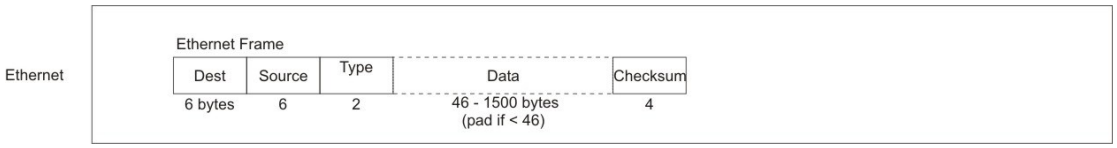
The bulk of the work is done by a series of macros. The macros, their parameters, and a description of how they work are covered in the component help files. You will need to refer to the help files for further information on the macros.

6 Ethernet layer

Overview

The Ethernet layer is the physical layer of the OSI model. It handles communications between the originating device and the other physical devices in the network such as hubs or switches. All communications on an Ethernet based network are done as Ethernet frames. Higher level protocols transmit their message inside the data section of an Ethernet frame. This message in a message is stripped out and processed by the receiving node. But the fundamental basic level of communication is at the physical level from MAC device to MAC device.





The Ethernet frame consists of a header section, a data section and a checksum section.

The Ethernet layer is also known as the MAC layer after the Media Access Controller devices that handle the communications. When working with the Ethernet layer the terms MAC mode and Ethernet mode can be used interchangeably. Each MAC device has a built in unique 6 byte identifier known as a MAC address. This ensures that no two devices on the network have the same MAC address. By passing the source and destination MAC addresses along with the data the message can be error checked and responded to. The first three bytes are generally used as a company identifier, and the final three bytes are used to give each device from the company a unique ID.

Normally the MAC identifier is hard coded into a device at manufacture and cannot be changed. However the E-Blocks internet boards have a user definable MAC identifier. This is so that you can set up specific MAC identifiers for testing and debugging. The default MAC address for the internet board is: 0.8.220.0.0.0, with 0.8.220 being the company ID for the makers of the particular device used on the internet board.

> When using more than one internet board you need to ensure each board has a unique MAC address.
> Otherwise the Ethernet layer protocols will be unable to differentiate between boards, which will lead to problems.

The Ethernet frame consists of the following elements:

Ethernet Frame

	Explanation	**Bytes**
Destination	The Destination MAC address can be the 6 byte MAC address of a known node, or a general address such as the broadcast address 255, 255, 255, 255, 255, 255.	6
Source	The Source MAC address is the MAC address of the sending node.	6
Type	The type section can vary depending on what protocol you are using. For example the ARP protocol has the type data: 8, 6.	2
Data	The data section can be from 46 to 1500 bytes in length. If the data section is less than 46 bytes it will need to be padded with extra bytes to bring it up to 46 bytes.	46 to 1500
Checksum	The Ethernet frame is rounded off by a 4 byte checksum number. Fortunately this is generated by the TCP/IP component so you will not normally need to deal with it.	4

7 Address Resolution Protocol

With the TCP/IP component we can enter the OSI model at the Physical layer using a MAC mode connection. Creating a MAC socket allows us to create send and receive Ethernet frames. However MAC or Ethernet mode is the outer wrapper that all the other protocols are wrapped in. It is concerned only with the physical sending of data between two MAC devices, not what the message is. So to demonstrate this messaging system in action we need a message that we can slot into the Ethernet frame and work with directly at the Ethernet level. To do this we will send an Address Resolution Protocol message (ARP).

ARP is as close as we can get to working with the Physical layer itself. ARP is a message that fits inside the Ethernet frame data section and can be acted upon directly. Unlike higher level protocols that may consist of messages within messages which require extracting and processing before being acted upon. The higher levels, such as IP and TCP use the Ethernet Physical layer as the outer physical wrapper in which their own datagram is sent. However the TCP/IP component handles this wrapping behind the scenes so we never actually see it in action when using those modes.

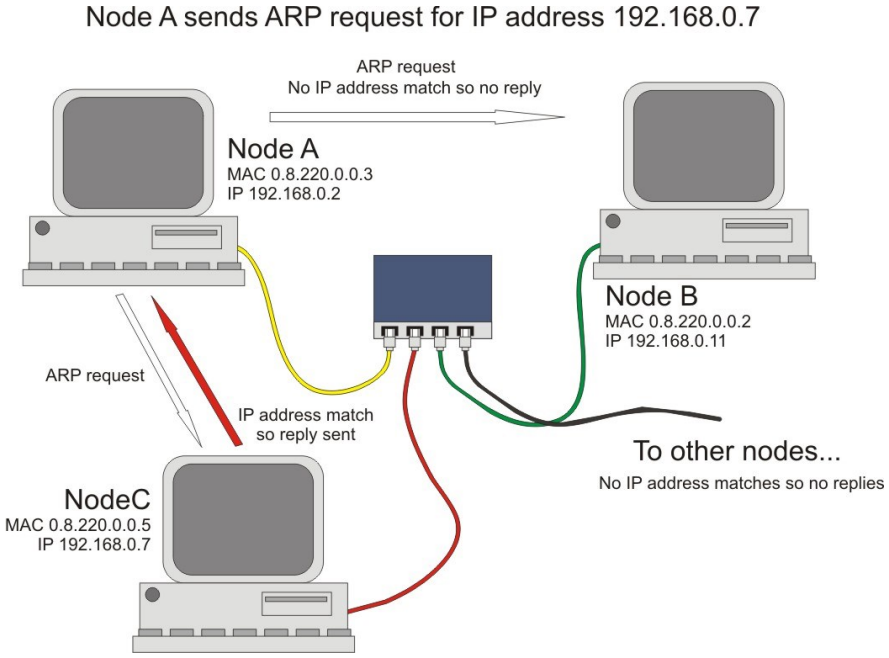
What's it used for?

- An ARP scanner program can be used as part of a larger TCP/IP system to verify MAC addresses before sending further messages i.e. as an initial point of contact error checking service.

The scanner can also be used to identify potential IP address conflicts as different nodes with identical IP addresses will have different MAC addresses.

ARP in action: finding the MAC address for a message

When a protocol sends a message it needs to send the message to a particular IP address. However the physical layer that does the sending communicates between MAC devices. So how does it get the MAC address it needs? This is where ARP comes in use. ARP can be broadcast to all the nodes in a network. If a node receives a message meant for its IP address it can respond. Otherwise it can simply ignore the message. So ARP can be used to ask for the IP address.



Example:

1. Node A needs to send a message to IP address 192.168.0.7 but does not know the MAC address to send to.

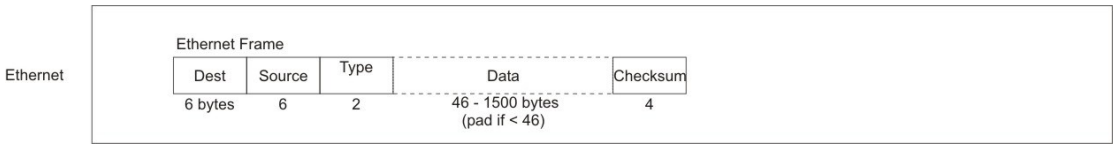
2. Node A broadcasts an ARP request for IP address 192.168.0.7 to all nodes on the network
3. Node B (IP address 192.168.0.11) receives the message but ignores it.
4. Node C (IP address 192.168.0.7) receives the message and responds, sending its MAC address with the reply.
5. Node D (IP address 192.168.0.34) receives the message but ignores it.
6. Node E (IP address 192.168.0.3) receives the message but ignores it.
7. And so on for all the other nodes in the network which will ignore the message.

8. Node A receives the reply from Node C and extracts the MAC address for IP address 192.168.0.7.
9. Node A is now ready to send the message.

In addition to being able to send the message the Node can also save the IP address/MAC address combination for future reference so that it does not need to clog up the network with broadcasts trying to find the same IP address later on. On a network with say 200 nodes this could represent a significant reduction in network traffic.

Ethernet frame: The outer wrapper

The ARP request is contained in the Ethernet datagram's data section.
So first you need to create the Ethernet header, and then add in the ARP datagram
The Ethernet header consists of the destination MAC address, the source MAC address and the Data – in this case the ARP datagram.



The following data is required:

Ethernet header

	Explanation	Bytes	Notes
Destination	The destination MAC address.	6	Using the MAC broadcast address 255.255.255.255.255.255 will pass the message to all MAC devices on the Local Area Network.
Source	The MAC address of the sending device.	6	Available in the TCP/IP component property page.
Type	The operation type	2	

Data

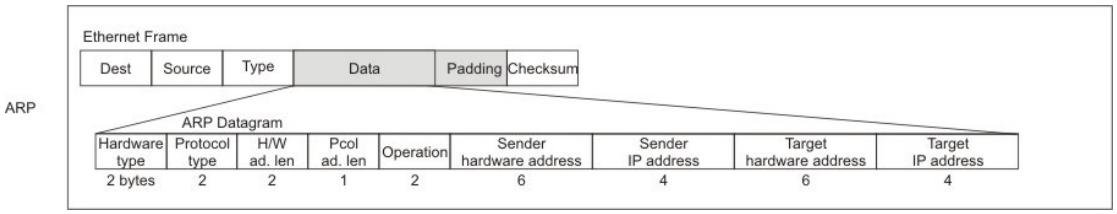
The Ethernet data section needs to be 46 or more bytes long. If the data length is less it needs to be padded out. Fortunately the TCP/IP component will handle this automatically.

Checksum

A checksum is needed at the end, but this too is handled automatically by the TCP/IP component.

The ARP datagram

The ARP data packet seems rather complex, but is mostly a matter of filling in the blanks with known information. Or using standard codes for items such as operation and protocol type.



ARP datagram

	Explanation	Bytes	Notes
Hardware type:	The type of hardware used. generally this will be an Ethernet connection, value 0x0001	2	
Protocol type	The type of protocol used. IP = 0x0008.	2	
H/W adjustment length	The Hardware length is the length of the hardware identifier, in this case 6 for the 6 byte MAC address.	1	
Protocol adjustment length	The Protocol length is the length of the protocol identifier, in this case 4 for the 4 byte IP address.	1	
Operation	The operations value that we will be using are 0,1 for an ARP request, and 0,2 for an ARP reply.	2	
Sender hardware address	The MAC address of the sending hardware device.	6	The default is 0.8.220.0.0.0
Sender IP	The IP address of the sender. Available on the component properties page.	4	The default is 192.168.0.2
Target hardware address	Needs to be filled in, but is ignored by the receiving device.	6	
Target IP address:	The IP address that we wish to check for.	4	

One important point not to overlook is that the ARP datagram is wrapped up in the data section of the Ethernet frame – a message in a message. When the Ethernet device at the other end receives the Ethernet message the ARP datagram will be split out and processed by the node. The job of the Ethernet frame is simply to ensure that the message arrives.

8 Implementing Ethernet mode in Flowcode

Using the TCP/IP component in Ethernet mode allows access to direct MAC to MAC communications at the physical level. This section discusses the icons, settings and macros that would be needed to implement an Ethernet mode project in Flowcode.

Note: This section covers the fundamentals of sending and receiving data, and should be read before moving on to the other protocols.

There are five basic elements to implementing Ethernet mode:

- Initializing
- Creating a MAC socket
- Sending data
- Receiving data

Matching data

Initializing

Before anything can be done the TCP/IP component requires initializing.
This initializing is required by all programs using the TCP/IP program.
Add an '*Initialize*' macro to the beginning of your program to initialize the TCP/IP component.

Creating a MAC socket

To send an Ethernet datagram you need to create a MAC socket to send it on. A Socket is simply a term for a connection to another system that you can communicate to, just like you need to plug a microphone into the socket for the sound for it to reach the mixing desk.
Add a *CreateMACSocket* macro to the program.
CreateMACSocket takes the parameters *promiscuous, broadcast* and *error,* which are detailed in the help file.

As *CreateMACSocket* returns a non-zero value if the connection is successful you can use this return value for error checking to inform the user that there has been a problem and end the program.

Sending data

The transmit macros take a channel parameter to set which channel to transmit on.
Ethernet packets need to be sent on channel 0, so make sure this is used for all of these macros.
(Channels are a feature of the Hardware TCP/IP stack IC that is used on the internet board not of the TCP/IP system in general.)

First thing to do is, of course, to calculate or collate the data to be sent.

Once you have the data ready the process of sending a message is as follows:

- *TxStart* (channel) Readies the buffer to begin accepting transmission data.
- *TxSendByte*(channel, byte) Sends the data byte to the buffer for the specified channel. Needs repeating until all the data has been sent to the buffer. Two other useful macros that send Property data are available as well:
  - ✦ *TxSendMyMAC* Sends the TCP/IP components MAC address (6 bytes)
  - ✦ *TxSendMyIP* Sends the TCP/IP components IP address (4 bytes)
- TxEnd (channel) Used to initiate the data transmission. The data sent to the buffer is then transmitted.

We have the following sections of raw data to send:

- Ethernet header data

Datagram data

Other items that are required, such as data padding and the checksum, are handled automatically by the TCP/IP component.

Reading in the data

RxDataAvailable (channel) is the key macro here. It returns a non-zero value if data is available.
You can then go through and read in the data.
The TCP/IP component has several macros for reading in the information.

- *RxReadHeader (channel, idx) RxReadHeader* allows you to read in the specific bytes of the header section. The size of the header and the data contained vary depending on which protocol is used. Refer to the macro's section above for details on the various header bytes.
- *RxReadByte (channel)* Reads the next byte in from the buffer. This is the basic data retrieval macro that you need to use.

RxSkipBytes(channel,count)

Once you have collected all the data you need from the message you will need to clear the data buffer to allow the next message to come in.

RxFlushData(Channel) will clear the buffer for that channel.
If you do not want to clear the data, but to reset it to read again e.g. after checking it through for errors, or in a firewall application, instead call the *RxDataAvailable* macro as this resets the reception buffer pointer to the beginning of the packet.

Matching data

When you analyze the data there will often be specific bits of data that you want to check for i.e. your MAC address, your IP address, or a specific set of bytes.
The TCP/IP component has a batch of *RxMatchXXXX* macros to help with this.
All these macros work in the same way. They check the required number of bytes in the buffer and return a non-zero value for a match, or 0 for no match.

- *RxMatch_2_Bytes*, *RxMatch_4_Bytes* and *RxMatch_6_Bytes* test the next 2, 4 or 6 bytes respectively to see if there is a match.
- *RxMatchMyMAC* and *RxMatchMyIP* check for the MAC address (6 bytes) and IP address (4 bytes) respectively.

Exercise 1: ARP scanner

Instructions
Create a basic ARP scanner that checks the available MAC address for the range of IP addresses 192.168.0.0 to 192.168.0.255.
If an ARP response is received – indicating a valid IP address, display the IP address and associated MAC address on the LCD display.

Exercise objective:

- To build a program that can scan a range of potential IP addresses on the network

Display the IP address and MAC address for any that are found on an LCD display.

Prerequisites:

- Familiarity with Flowcode.

Knowledge of the LCD display component.

Learning outcomes:

- MAC and IP address structure
- The structure of the Ethernet framework, which underpins all the other TCP/IP protocols.
- The datagram principle, where data inside a message may itself be a message.
- The process of sending, receiving and matching bytes.

10 Worked example: Implementing ARP in Flowcode

This section will take you through a worked example to demonstrate the basic macros involved in a bit more detail. We will use the example of a basic ARP scanner program set as Exercise 1to illustrate the various steps involved.

Exercise 1 objective:
To build a program that can scan a range of potential IP addresses on the network and display the IP address and corresponding MAC address for any that are found on an LCD display.

10.1 Network traffic analysis

Using a network traffic analyzer such as Wireshark is highly recommended as it allows you to examine the packets being sent. Wireshark is particularly useful for debugging errors in the packet formation.

10.2 The basic structure

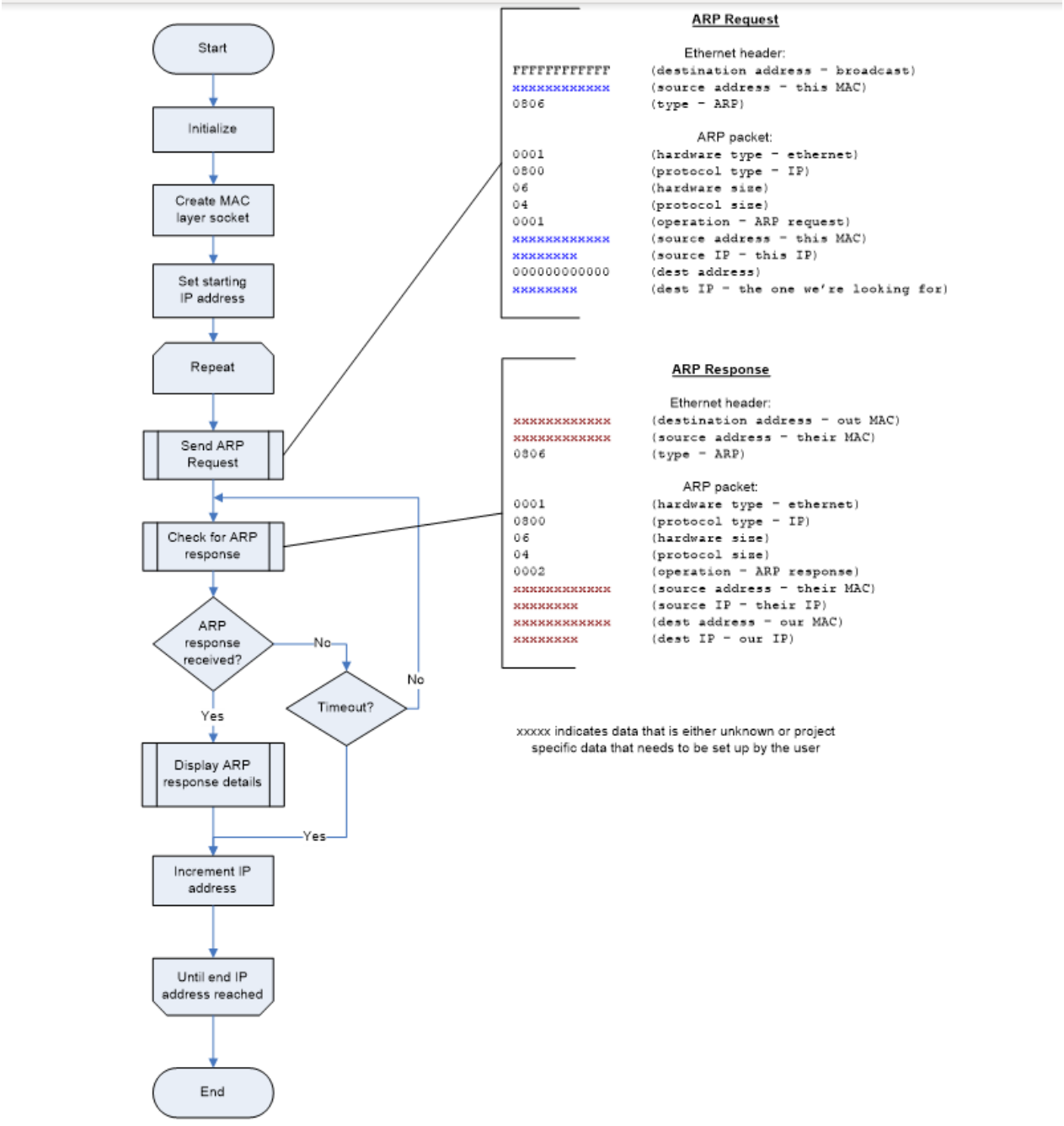
There are four basic elements to the ARP program:

1. Initialize and set up the TCP/IP component.
2. Loop through the range of IP addresses we wish to search and send a query to them.
3. Check if we got a response back from our query. If so retrieve the IP address and MAC address. If no response (timeout check) then go on to the next IP address in the loop.

Output the retrieved data to the LCD display.

The receiving and displaying parts of the program will be contained in the main send message loop. In order to simplify matters we will search only a certain range IP addresses 192.168.0.0 to 192.168.0.255. This means we only need worry about the last IP address byte which will simplify the program. Actually there is another trick here as well. This range of addresses is traditionally reserved for systems on the same local network or domain, which makes life much easier for us as we are much more likely to get a response. They are also non-internet addresses so we do not need to worry about accidentally trying to communicate to some far away machine.

We can represent the program design with the following flowchart:



10.3 Initializing

The first step is the standard initialization stage where you set up the components and variables.

- Initialize the TCP/IP component
- Initialize the LCD Display

Initialize any variables – e.g. IP address Min, Max and starting values

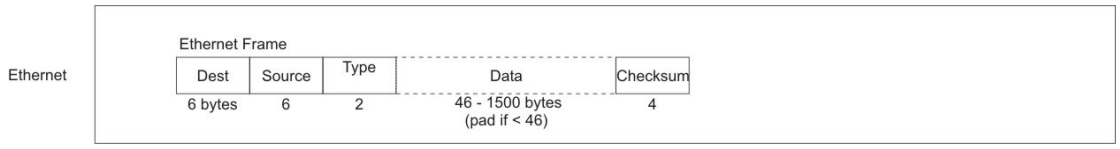
Add an '*Initialize*' macro to the beginning of your program to initialize the TCP/IP component, and a '*Start*' macro for the LCD as well.

To send an Ethernet datagram you need to create a MAC socket to send it on. Add a *CreateMACSocket* macro with the parameters set to *promiscuous = 0, broadcast =0, error = 0*. (You don't want to listen to any other traffic on the network, only messages sent to this node.)

Set up a return value such as *RETVAL* to check for a successful connection. If it fails you can then error trap it and inform the user that there has been a problem and end the program.

Sending the ARP request

The ARP request is contained in the Ethernet datagram's data section.
So first you need to create the Ethernet header, and then add in the ARP datagram
The Ethernet header consists of the destination MAC address, the source MAC address and the Data – in this case the ARP datagram.



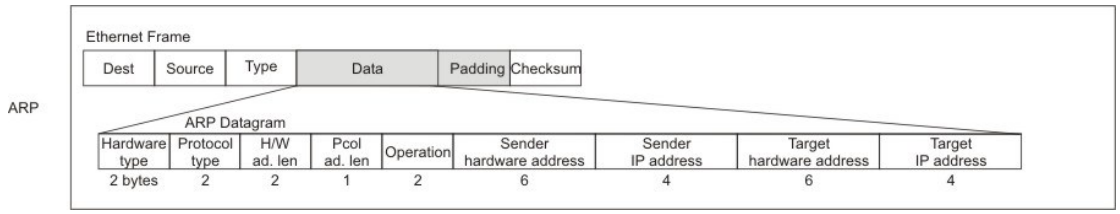
The following data is required:

Ethernet header

	Explanation	Bytes	Data for exercise
Destination	The destination MAC address. Using the MAC broadcast address 255.255.255.255.255.255 will pass the message to all MAC devices on the Local Area Network.	6	255.255.255.255.255.255
Source	The MAC address of the sending device. Available in the TCP/IP component property page.	6	Use TxSendMyMAC
Type	The operation type	2	ARP = 8, 6

ARP datagram

The Data section for this example is rather complex, but is mostly a matter of filling in the blanks with known information. Or using standard codes for items such as operation and protocol type.



ARP datagram

	Explanation	Bytes	Data for exercise
Hardware type:	The type of hardware used. generally this will be an Ethernet connection, value 0x0001	2	0,1
Protocol type	The type of protocol used. IP = 0x0008.	2	0, 8
H/W adjustment length	The Hardware length is the length of the hardware identifier, in this case 6 for the 6 byte MAC address.	1	6
Protocol adjustment length	The Protocol length is the length of the protocol identifier, in this case 4 for the 4 byte IP address.	1	4
Operation	The operations value that we will be using are 0,1 for an ARP request, and 0,2 for an ARP reply.	2	0, 1 for request

Sender hardware address	The MAC address of the sending hardware device.	6	Use TxSendMyMAC
Sender IP	The IP address of the sender. Available on the component properties page.	4	Use TxSendMyIP
Target hardware address	Needs to be filled in, but is ignored by the receiving device.	6	0.0.0.0.0.0
Target IP address:	The IP address that we wish to check for.	4	Either entered manually, or set as variables e.g. a 4 byte array if the values may change.

10.4.2 Sending the message

The transmit macros takes a channel parameter to set which channel to transmit on.
Ethernet packets need to be sent on channel 0, so make sure this is used for all of these macros.

Once we have the data figured out we can send the message. The process of sending a message is as follows:

- *TxStart* (channel)
- TxSendByte(channel, byte) Needs repeating until all the data is sent. Note that TxSendMyMAC and TxSendMyIP can be used to send the MAC address and IP address.

TxEnd (channel) Used to finish the data transmission.

The following sections need to be sent:

- Ethernet header data

ARP datagram data

Data padding and the checksum, are handled automatically by the TCP/IP component so can be ignored.

10.5 Getting a response

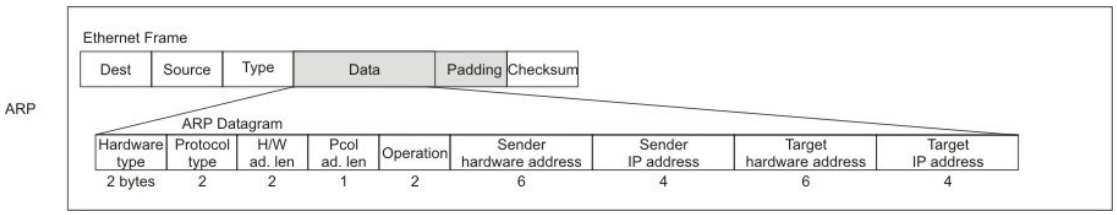
When a node receives the ARP request meant for it (matches the IP address) it will send a response back to the sender using the MAC address and IP address gathered from the message data. On a single connection, such as that used with the internet training solution, we should only get a single response from the PC, as there is only one IP address to send to. However on a full network we may receive multiple responses, one from each IP address present on the network that has a request sent to it.

Depending on issues such as network lag, or the speed at which a node process requests the responses may not arrive in the order that the requests were sent.

If the node is trying to send a message of its own it may also contact our node with an ARP request of its own. Something we will need to check for.

The first step is to check that for incoming data using the *RxDataAvailable* macro. Remember that all Ethernet communications are on channel 0.

The next step is to determine if it is an ARP message. We can do this by checking the Ethernet header to see if the type matches up as an ARP message type. We can skip over any preceding data until we get to the bytes that we wish to match. The macros *RxSkipBytes* and *RxMatch_2_Bytes*, *RxMatch_4_Bytes* and *RxMatch_6_Bytes* can be used to move to and match the required data.



Ethernet header

	Bytes	Skip read or match	Notes
Destination	6	skip	
Source	6	skip	
Type	2	Match for ARP – 8, 6	

Once we have determined it is an ARP message we can go through until we reach the operation bytes which we can test to see if the message is an ARP reply. If so we can go through the rest of the message and extract the MAC address and IP address. Once we have the data we need we can stop at that point. We don't need to read in the whole message.

ARP datagram

	Bytes	Skip read or match	Notes
Hardware type	2	Skip	
Protocol type	2	Skip	
H/W adjustment length	1	Skip	
Protocol adjust-ment length	1	Skip	
Operation	2	Match for ARP reply – 0, 2	
Sender hardware address	6	Read in and display	The MAC address we are looking for
Sender IP address	4	Read in and display	The IP address that sent the response
Target hardware address	6	Skip – Actually we can stop here as we don't need any more data.	
Target IP address	4	Skip	

Note that we get both the MAC address and the IP address. This is because we can't assume that the response is from any specific IP address, such as the last one we messaged. Due to network traffic, and different device response speeds it could potentially be any of the IP addresses we previously messaged. Or it could be an ARP request from another node on the network trying to find our MAC address. Also if two machines have the same IP address we could receive two responses from the IP address, but with different MAC addresses for the different nodes.

Once we have collected all the data we need from the message we will need to clear the data buffer to allow the next message to come in using *RxFlushData*.

Timeout

On a slow or busy network the program may be able to respond much faster than the network. So we need to be able to wait for a response.

Also our request may get ignored or lost, especially if the address we sent to doesn't exist. So we need to decide on a reasonable time span to wait before we decide that to move on and try the next address.

Displaying the response

Once a response has come in we will need to display the IP address.

This is a relatively simple task, which we shall leave up to you to implement.

Finishing off the program

Now that we have sorted out sending the ARP request and reading in the reply we just need to put the finishing touches to the program. The main loop needs to be set up for instance. We can control the

loop by setting a starting address and checking for when it has reached an end address. A handy range to check is from 198.162.0.0 to 198.162.0.255 as this particular IP range is reserved for nodes on the same network, which is what we mainly want to check.

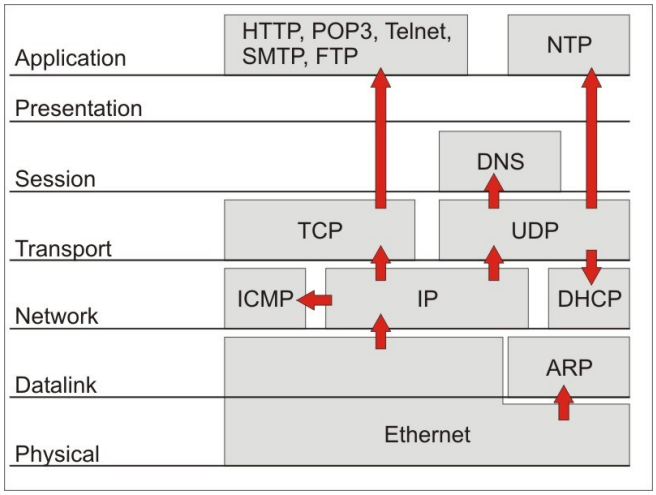
Example program

An example solution can be found in the Flowcode “TCP_IP\Examples” folder, or in the Examples section of the CD, for you to refer to, use for code, and use for demonstration purposes.

> Note that properties, macro parameters and variable values etc. were set up for use with our network and may require changing to match your own network or system configuration.

IP Layer

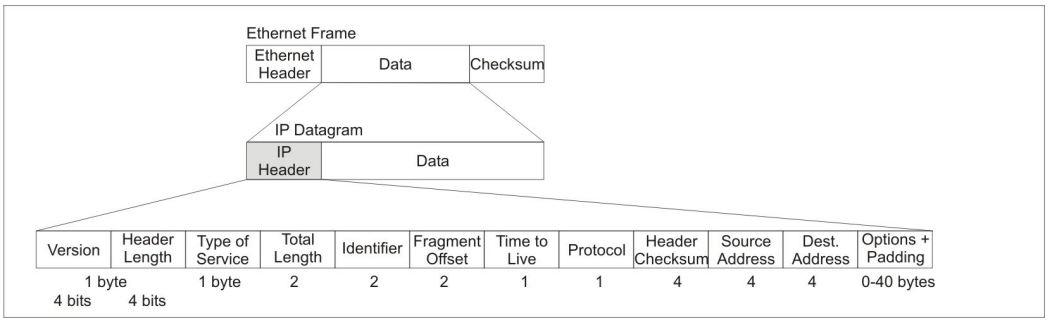
The IP layer is at the Network level of the OSI model where communications between nodes is the key, not communication between physical devices. A message may be required to be sent via a number of MAC devices in order to reach the final destination. The IP layer provides the stamped addressed envelope for network transport in the form of an IP address. The IP address is normally a four byte code such as 192.168.0.2. The IP address gives network layer software and hardware such as routers the information they need to send the message to the correct recipient node. Other information is passed as well that the receiving software can use to determine where the message originated from, and what kind of message it is.



The website addresses that we commonly used, such as www.matrixltd.com are actually IP addresses in disguise. TCP/IP protocols such as DNS are used to retrieve the numeric IP address from the non-numeric values we give them. This is due to the human mind being better able to remember words than numbers.

The IP datagram (derived from Telegram no doubt, adding to the mail system metaphors) is nestled inside an Ethernet frame to allow the physical sending of the message onto the network. The IP datagram contains a header file and a data packet. Just as the IP datagram (the envelope) is tucked inside an outer wrapper Ethernet frame, the data section is in itself a message (the letter itself). This message can be in a number of different formats e.g. TCP, UDP, and ICMP etc.

IP



IP header

	Description	bytes
Version and Header length	Two 4-bit values combined into one byte For this course we will be using IP version 4. The Header length is measured in 32 bit words. Normally this is 5 words. The normal value would thus be 0x45.	1
Type of Service	Sets the priority for the datagram. Can normally be ignored and set to 0.	1

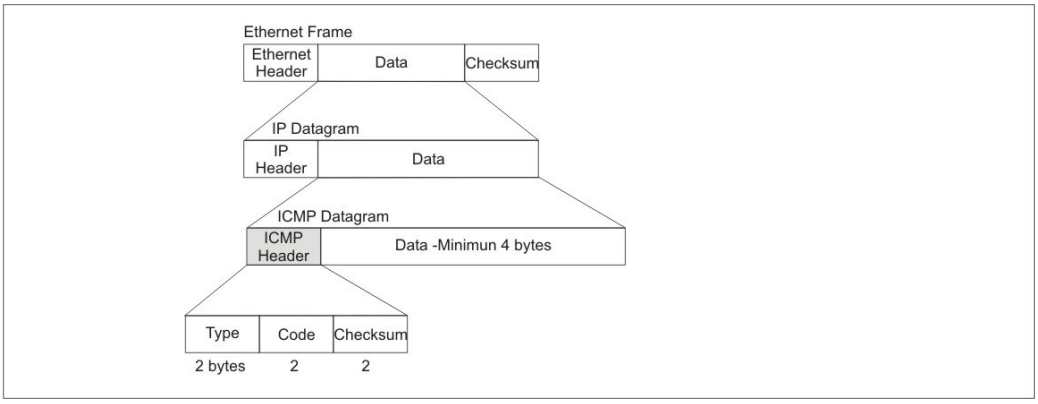
Total length	**Total length of the datagram including both header and data.**	**2**
Fragmentation Identifier	Used to aid fragmentation (i.e. breaking up messages too large for one datagram). We will not be covering fragmentation here, so can be set to 0.	2
Fragmentation offset	Used to aid fragmentation (i.e. breaking up messages too large for one datagram). We will not be covering fragmentation here, so can be set to 0.	2
Time to live (seconds)	Messages can only survive for a certain amount of time of the network. This value is decreased during transit by the routers until the message is finally discarded. A good default is 100 seconds.	1
Protocol	Identifies the protocol of the message contained in the data section. Possible values used to indicate which protocol is being used include: • ICMP = 1 • TCP = 6 • UDP = 17	1
Checksum	A checksum value for the IP header. Normally you will need to calculate this, as it all gets handled by the TCP/IP component, however should you need to generate the checksum manually you can use the following process. Byte swap the header Pad with 0 if an odd length Clear checksum value Sum the 16 bit header words Put one's complement into checksum	4
Source IP address	The originator's IP address.	4
Destination address	The IP address of the destination node.	4
Options	Various options are available for the IP header. We will not be using any options in this course, so this section can be safely ignored for now.	0-40

Using the IP layer: ICMP and Ping

You can't really have a standalone IP example, just like you can't really have a standalone Ethernet example. They are cargo carriers not the end data. Normally IP is used to transmit higher layer protocol messages such as TCP or UDP. What we can do is to put a simple message protocol into an IP datagram that we can use directly without having to add any further protocol layers on top. A nice and convenient one is ICMP – Internet Control Message Protocol.

ICMP is commonly used to check connections with other computers. This is known as 'pinging' as it works in the same way as the sonar 'ping' sent out by submarines. If it hits a target an echo will return which can be picked up and examined.

The ICMP datagram



The basics are that an ICMP message will receive an ICMP reply which can then be examined to see if there were any errors, and if so which ones.
For example sending an ICMP message to a node with an unreachable port would generate a respond with Type = 3 (Destination unreachable) and Code = 4 (Port unreachable). As you can guess getting this kind of information is invaluable for network debugging.

	Description	**bytes**
Type	Details the type of response or request being made. Uses the following types: • 0 Echo reply • 3 Destination unreachable (error code) 8 Echo request	1
Code	Gives details about any error encountered. Uses the following error codes: • 0 Network unreachable • 1 Host unreachable • 2 Protocol unreachable • 4 Port unreachable • 5 Fragmentation needed but not allowed • 6 Destination network unknown • 7 Destination host unknown	1
Checksum	The checksum used is the same as for the IP header • Byte swap the header • Pad with 0 if an odd length • Clear checksum value • Sum the 16 bit header words • Put one's complement into checksum	2
Data	Message length needs to be a minimum of 8 bytes (including the header), even if the final 4 bytes are unused. If the message is less than 8 bites pad the data until it reaches 8 bytes in length. The 4 data bits are a 2 byte Identifier number and a 2 byte Sequence number.	4+

ICMP Ping data

Standard ICMP Ping requests send the ASCII character a-z and then a-j as data. It is suggested that you do the same as receiving applications may be expecting that data.
To make life easier when implementing this we can loop through the ASCII values - from 97-120 for a-z and from 97-106 for the second a-j section.

IP mode allows you to enter the OSI model at the network level. At this level the TCP/IP component will handle the lower level Ethernet frames for you.

Where are the Ethernet and IP headers?

The most important change between Ethernet mode and IP mode is that the headers are handled for you by the TCP/IP component. In Ethernet mode you needed to add all the header information to the message as well as the data. But with IP you don't. Creating an IP socket takes care of all the headers for you, all you need to provide is the data for the IP datagram.

Note that if the datagram being sent has its own header, such as with ICMP, you will need to provide the header information yourself as that is part of the data being sent via IP mode.

Setting up the IP connection

The first part we need to look at is setting up the IP connection.

- Initialize the component
- Create an IP socket

Set the Destination address and port

First we need to initialize the TCP/IP component with an *Initialize* macro.

Next a *CreateIPSocket(Channel, Protocol, Broadcast)* macro is added.
For this example we will be sending an ICMP datagram (*protocol* value of 1) on *channel* 0, and wish to send and receive, so we will turn *broadcast* on (value set to1).

Add a *SetDestination(Channel, Dst_IP0, Dst_IP1, Dst_IP2, Dst_IP3, Dst_Port_Hi, Dst_Port_Lo)* macro.

- The channel can be set to 0-3.
- dst_ip0 to dst_ip3 are the four IP address bytes.

Values must be given for the Port numbers even though the IP socket does not use them as all parameters are required to be filled by Flowcode. They are needed for another mode which shares the same macro (UDP mode which we will cover later). Zero or a random value will suffice.

Sending the datagram

Apart from the actual data sent the procedure is the same as that outlined earlier for the Ethernet messages. We will need a *TxStart(Channel)*, a batch of *TxSendByte(Channel, Data)* macros to add the message data, and a *TxEnd(Channel)* macro to finish it off.

Receiving a response

Once again this follows the familiar pattern that we used with Ethernet packets.

- *RxDataAvailable(Channel)* lets us check if any data has arrived.
- The *RxReadHeader*, *RxReadByte*, *RxSkipBytes* and *RxMatchXXXX* macros are used to read in and check the data.

RxFlushData(Channel) is used to clear the buffer once we have finished reading.

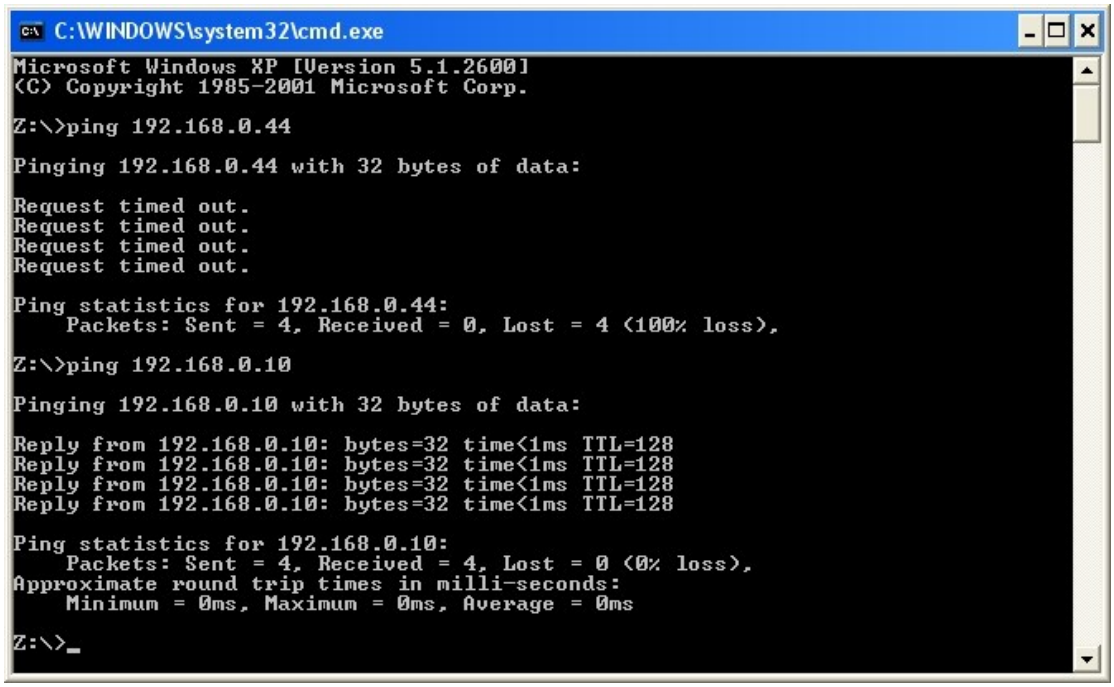
The Windows Ping program

Windows comes with its own Ping program that we can use to test that our Ping program can send responses. You may already have come across it, and used it to check connections and connection speeds.

We can ping the internet board from the PC to see if we can get a reply from it.

Ping is simple to use. Simply open up a Command prompt (Run CMD) and type Ping <IP address to ping> e.g. Ping 192.168.0.2

The address will be pinged and the resulting timing data displayed.



We can use Ping to test our program. Simply ping the IP address you set for the TCP/IP component (192.168.0.2 by default). If all goes well it should get a very fast response time.

Exercise 2: Ping program

Instructions
Create a Ping program that sends a ping - an ICMP request, to an IP address.
If a response is received – indicating a valid IP address, display the IP address and associated MAC address on the LCD display.

Exercise objective:
To ping another node to see if we get a response, hence can find it on the network.

Prerequisites:
Knowledge of the MAC/Ethernet layer

Learning outcomes:
- Structure of an IP Datagram.
- Sending IP messages.
- Receiving IP responses.
- Reading data from the IP message.

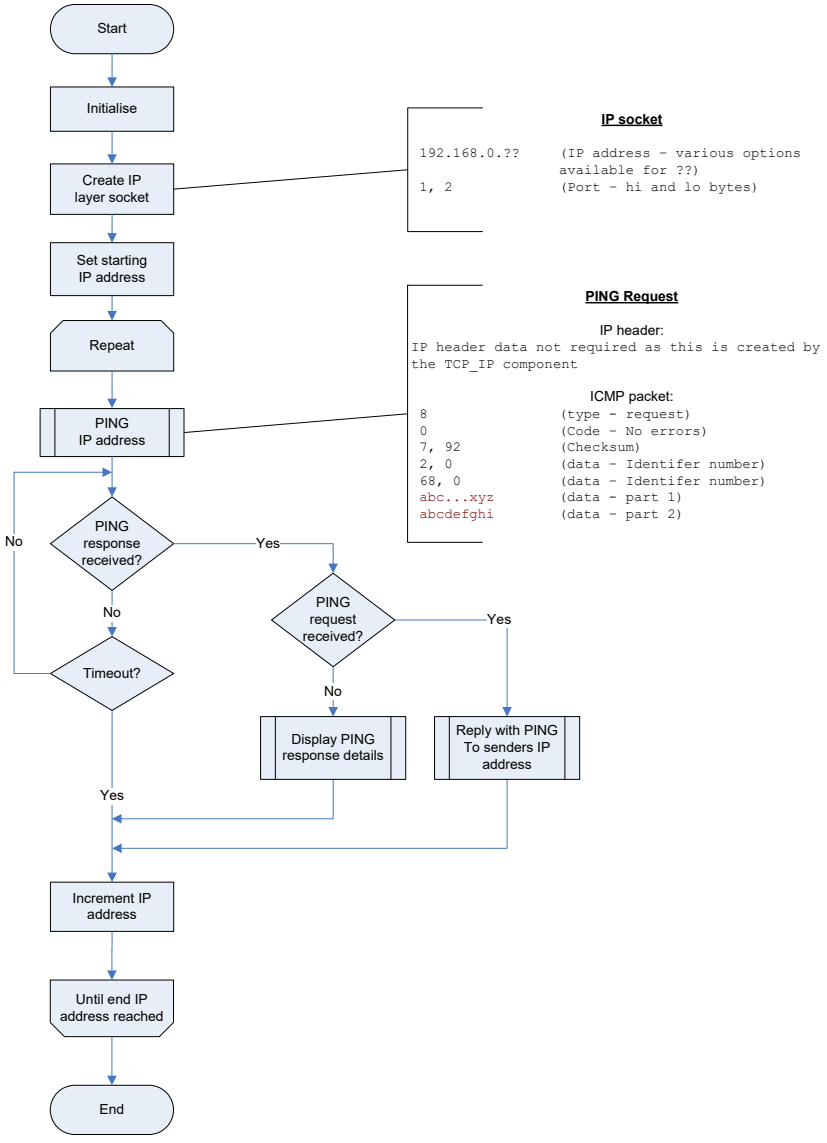
Worked example: Ping

This basic Ping program is the one set as Exercise 2, so this section can be used as the basis for a set of notes for use that exercise.

Exercise 2 objective:
To ping another node to see if we get a response, hence can find it on the network.

Ping overview

Ping sends out a signal to a specified IP address – i.e. a potential node on the network.
If the node exists and is functioning correctly then the node will reply with an echo of the data sent.
The IP address and MAC address can be stripped from the reply and displayed. Other nodes too can Ping this one, so we will need to respond to a Ping request with a reply of our own.



Setting up the IP connection

First we need to initialize the TCP/IP component, so an *Initialize* macro is added to the beginning of the program.

Next a *CreateIPSocket(Channel, Protocol, Broadcast)* macro is added.
For this example we will be sending an ICMP datagram (*protocol* value of 1) on *channel* 0, and wish to send and receive, so we will turn *broadcast* on (value set to1).

Add a *SetDestination(Channel, Dst_IP0, Dst_IP1, Dst_IP2, Dst_IP3, Dst_Port_Hi, Dst_Port_Lo)* macro and set up the parameters to be 0, 192, 168, 0, IP, 1, 2
.

The ICMP header values

We want to send a Ping so Type = 8, Echo Request. We don't have any errors to report, so code = 0.
The 4 data bits are a 2 byte Identifier number and a 2 byte Sequence number.
When replying you need to echo these back along with the data sent, so they are important for checking and responding. In our example we will set them to 2, 0 and 68, 0 respectively. All that remains for the header is to fill in the checksum.

ICMP datagram

	Description	**bytes**	**Value to use in exercise**
Type	Details the type of response or request being made. Uses the following types: • 0 Echo reply • 3 Destination unreachable (error code) 8 Echo request	1	8
Code	Gives details about any error encountered. Uses the following error codes: • 0 Network unreachable • 1 Host unreachable • 2 Protocol unreachable • 4 Port unreachable • 5 Fragmentation needed but not allowed • 6 Destination network unknown • 7 Destination host unknown	1	0
Checksum	The checksum used is the same as for the IP header • Byte swap the header • Pad with 0 if an odd length • Clear checksum value • Sum the 16 bit header words • Put one's complement into checksum	2	7, 92
Data	Message length needs to be a minimum of 8 bytes (including the header), even if the final 4 bytes are unused. If the message is less than 8 bites pad the data until it reaches 8 bytes in length. The 4 data bits are a 2 byte Identifier number and a 2 byte Sequence number.	4+	2 ,0, 68, 0 (Identifier = 2, 0 Sequence = 68, 0)

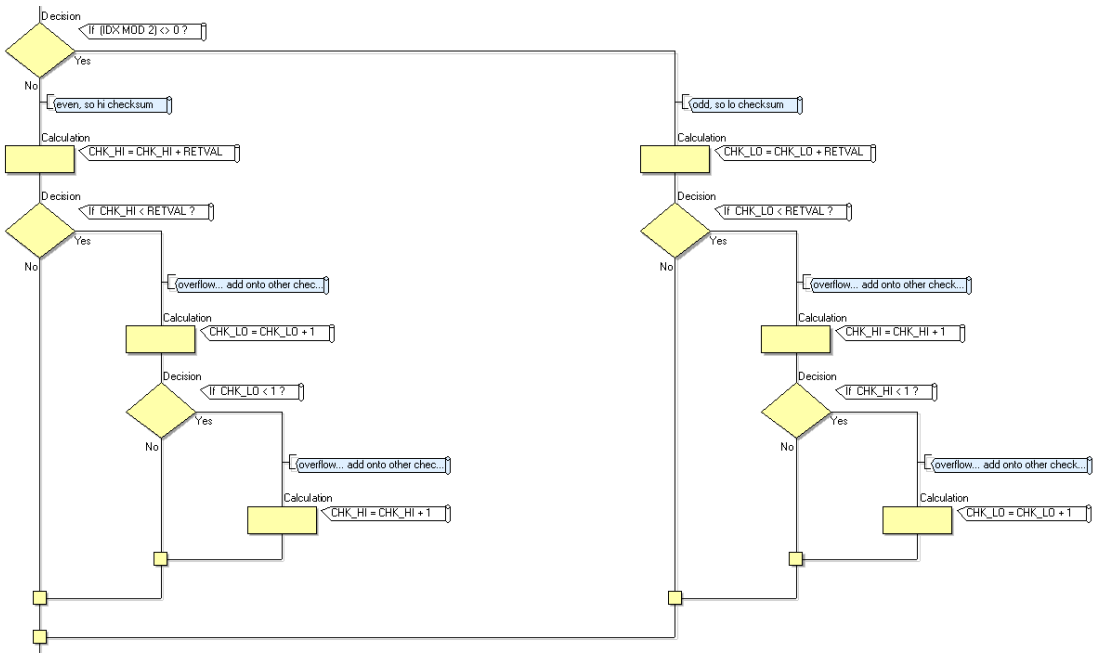
The checksum

We can either calculate the checksum values on the fly, or for set messages calculate them in advance and use those values.
In this case we have pre-calculated the checksum to be 7, 92, so we can add those two bytes in next.

However, when replying to a Ping request you will not be so lucky and will need to create checksums for your replies. The checksum code has been put into a convenient macro for you, which can be found in the "Flowcode\TCP_IP\Examples" folder. Otherwise you will need to create your own implementation of the checksum generation.

Byte swap the header
Pad with 0 if an odd length

Clear checksum value
Sum the 16 bit header words
Put one's complement into checksum



Part of a checksum routine.

The data

Once the header data is set up we can add the data.
We do this in two loops, one from 97-120 to add ASCII a-z and the second from 97-106 to add ASCII a-j.

Sending the ICMP datagram

Apart from the actual data sent the procedure is the same as that outlined earlier for the Ethernet messages. We will need a *TxStart(Channel)*, a batch of *TxSendByte(Channel, Data)* macros to add the message data, and a *TxEnd (Channel)* macro to finish it off.

Receiving a response

We are sending a ping request – i.e. a message for the other node to respond to us so we can time the response time. So we need to listen for the response and react accordingly.
For this we need to do one of several things.

Wait for so long then give up (timeout) if we haven't had a reply.
Process and display a result if we get a response.
If we get a ping request we need to retrieve the IP data and reply to the request.

Once again this follows the familiar pattern that we used with Ethernet packets.

- *RxDataAvailable(Channel)* lets us check if any data has arrived.
- The *RxReadHeader*, *RxReadByte*, *RxSkipBytes* and *RxMatchXXXX* macros are used to read in and check the data.

RxFlushData(Channel) is used to clear the buffer once we have finished reading.

The types of response we are looking for are a Ping reply (type = 0), or a Ping request (type = 8). The third option is Network unreachable (type = 3), in case we can then look at the error code value to see what kind of error was generated. Fortunately this is the very first byte in the buffer, so we can check it straight away.
Use RxReadByte to get the type value. You can also use a second RxReadByte to get the error code if needed.

Ping reply

If it is a Ping reply we can display the ping details on the LCD display.
The first 8 bytes of data are header information. The data starts after that. We can simply skip the next 7 bytes (we have already read in the 1st type byte) to get to the data section. We can then read the data in and print it out to the LCD display. As there may be quite a lot of data you may need to pick certain bits to print out, rather than trying to print it all.

Ping request

If it is a ping request we need to reply. It is customary for Command line programs like Ping to send up to four ping requests so that it can get timing data. So you may get more than one ping request.

We will be sending a Ping echo with no error code do the first two bits will be 0, 0.
The next two bytes will be the two checksum bytes, which we will need to calculate.

The rest of the message is an echo of the ping message that we received, including the Identifier and Sequence numbers and the data. We can reset the buffer, skip the first four bytes (type, code and the two checksum bytes) and go through reading bytes and sending them back. We need to know how long the data is to be able to do this. Fortunately the IP header stores the data length for us. We can use *RxReadHeader(0,1)* (channel 0, index 1) to retrieve the data length. Don't forget that we have already got the first four bytes though so the final check figure needs to be 4 less. (There is a second hi byte for the amount of data, but Ping data is generally quite small so we shouldn't need to worry about it.)

Expanding the Ping program

Even the basic program forms a useful diagnostic tool for networks, and can be easily expanded to provide additional error reporting capabilities.

With Ping you can verify IP addresses, useful for a multitude of tasks.
With a bit of timing code you can time or monitor connections – latency being a big issue in internet networking.
There are a number of error codes that can be returned with ICMP. These could form the basis of an expanded error checking project.

Example program

An example solution can be found in the Flowcode "Flowcode\TCP_IP\Examples" folder, or in the Examples section of the CD, for you to refer to, use for code, and use for demonstration purposes.

> Note that properties, macro parameters and variable values etc. were set up for use with our network and may require changing to match your own network or system configuration.

UDP

UDP (User Datagram Protocol) is a method of sending data direct to a specific socket (see below). UDP can be used as a direct communications protocol, where you send messages direct to a specific application on a specific system.

Sockets

A socket is a combination of an IP address and a port. A port is an electronic pigeonhole that the various applications on the system can scan for messages. When a message arrives at a port it is the responsibility of the applications to notice the arrival and to respond to the message. If no application is monitoring the port when a message arrives then that message will be ignored.

Applications can also monitor ports allowing you to communicate with them directly. Given a specific port number and a specific IP address you can communicate with a specific application on a specific node. This is useful for custom applications that need to communicate over the internet, such as online games, or off site control and monitoring programs.

In UDP we send a signal to a specific port and sit back and wait. If all went well and the application monitoring the port is configured to respond to incoming messages then the port will generate a response which we will receive, if not then we will receive nothing – no error message or anything. Unlike ARP and ICMP which generated responses UDP does not – unless you write an application that sends one. This has led to UDP being referred to as “send and pray” communications.

One thing UDP is useful for is to send custom data messages direct from one system to another. The data passed will be raw data so you will need to implement your own system for dealing with the data.

Predefined responses and reserved ports

A number of ports have predefined responses. E.g. port 7 echoes the data sent to it. Port 13 returns a date/time string etc. UDP can be used to trigger sockets that give a known response, such as returning a time string. If you need to check the time on another PC for synchronization purposes you could use UDP to send a message to Port 13, which will respond by sending a date/time string that you can then process.

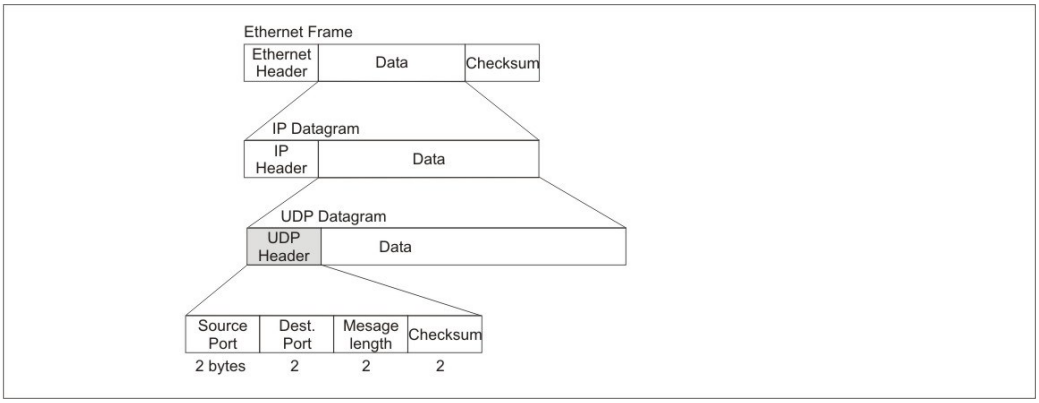
Some ports are normally reserved for specific protocols such as SMTP (port 25) or HTTP (port 80).
When dealing with known protocols such as SMTP this simplifies matters as we can connect to a known reserved port rather than having to find out from the system what port we need to connect to.
What ports are used can be changed but this would be unusual, and would be done for specific reasons such as security issues. In such cases you may need to speak to the Network Administrator to find out what ports are being used.

Ports and firewalls

In today’s age of viruses and Trojans an open port can quickly become infected. The first line of defense against such attacks is often a firewall application. Firewalls monitor communications and can block access for both incoming and outgoing messages on any of the ports. A standard tactic to minimize potential attacks is to block off ports that are not actively required. This has a major implication for UDP as firewalls may block access to a port, or prevent applications on that port from responding. Firewalls applications normally have properties or settings that can be configured to allow access to specific ports. Be careful when opening ports or disabling firewall protection as nodes that can be accessed from the internet will be vulnerable to attack.

UDP

The UDP datagram



UDP datagram

	Description	bytes
Source port	The senders port address.	2
Destination port	The port to be used at the destination. Note that if no application is listening to that port then the message may be lost.	1
Message length	The length in bytes of the data packet	2
Checksum	Checksum for the UDP packet. Handled automatically by the TCP/IP component in UDP mode.	4+
Data	The data for the UDP message. The data is sent as raw data. It is the responsibility of the sending and receiving programs to process this data.	0+

Note that like IP you will not normally need to create the header as it is handled automatically by the TCP/IP component in UDP mode.

Handling UDP data

UDP sends raw bytes of data. It is totally up to you to handle creating and interpreting the message data. This is both a blessing and a curse.

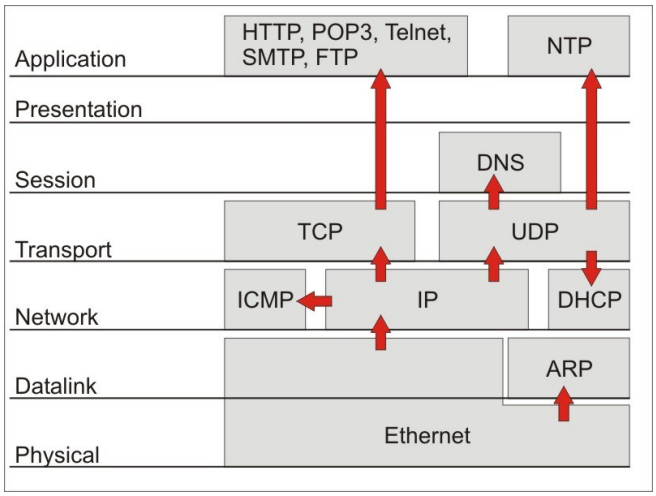
It is a blessing as you can decide what to send – ASCII text, integer values, data bytes for microcontroller output etc. etc. How much data and in what order is up to you as well. Basically you send what you want. If you wish to send custom messages UDP is the way to do it.

It is a curse in that you need to know what to expect when reading a message. Expecting and trying to read in 20 integer bytes when instead you got the 5 byte ASCII text “Hello” could cause errors.
UDP has no type information, or any other kind of in built error checking. Any error checking needs to be performed by the receiving application. As you are not in control of who sends a message to the socket you have chosen you need to be aware that a message received may not necessarily be in the expected format. It could for instance be from a completely different program with a completely different custom message structure.

Some predefined sockets, such as Port 13, respond with a predefined message – e.g. a date/time string but these are the exception not the rule, and even here the data format may vary from system to system

Implementing UDP mode in Flowcode

The Flowcode UDP brings you in at the Transport level of the OSI model. The lower protocols such as the IP transport layer and the Ethernet physical layer are handled by the TCP/IP component. In UDP mode you simply need to know which socket to monitor, and which socket to send data to.



Port bytes

There are a large number of available Ports, up to 65535 on some systems. However Flowcode is restricted to 8-bit variables i.e. 0-255. In order to increase the amount of ports we can address the port address is split into two byte, a high byte (src_port_hi) and a low byte (src_port_lo). Both bytes need to be supplied even for ports 0-255 which could be addressed with just one byte (simply use high byte '0' in this case).

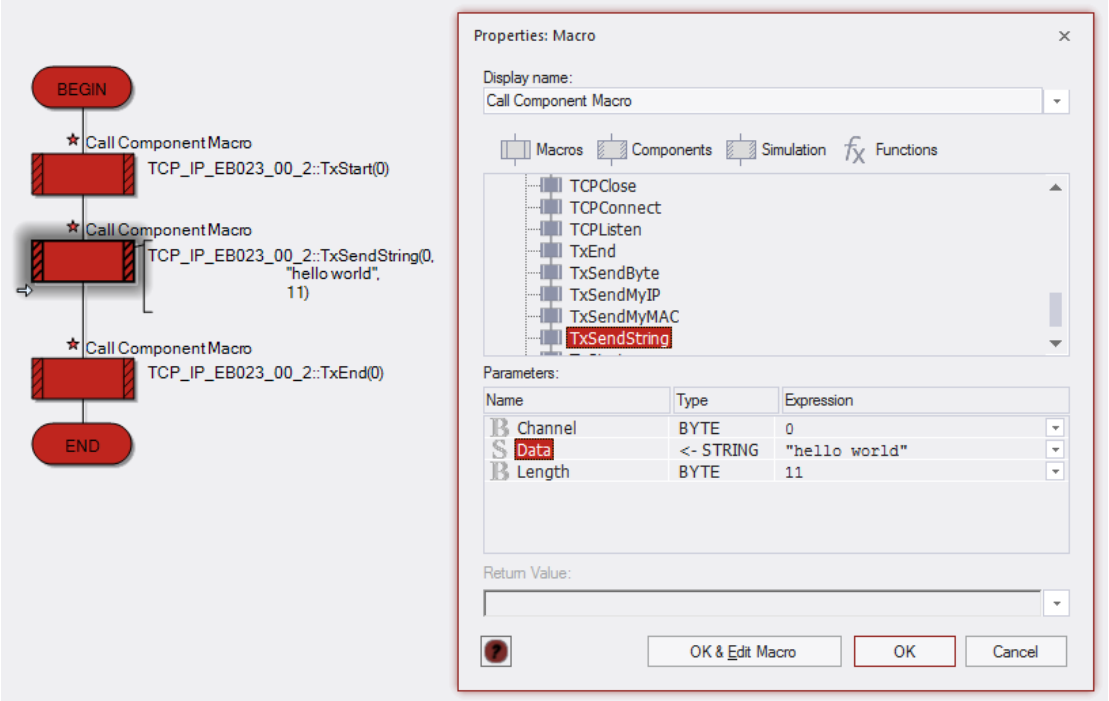
Setting up a UDP connection

Firstly add the *Initialize* macro to the beginning of the programs as usual.
UDP needs an IP address and a socket. The IP address is set up on the TCP/IP components properties page, so the only thing we need to set up is the socket port. The *CreateUDPSocket* macro takes the parameters *channel*, and the two port address bytes *src_port_hi* and *src_port_lo*. What port you choose is up to you. Once a socket has been set up you are able to monitor it using RxDataAvailable.
This allows you to receive messages even if you do not respond to them.

If you need to respond or send UDP messages you need to set the destination socket that you wish to send the message to. The *SetDestination* macro allows you to set the channel, IP address and port bytes of the target node. You can use the RxReadByte, RxSkipBytes and RxMatchXXXX macros to process.
Once you have set a destination the *TxSendByte* macro will send the data to that destination. To change the destination to a different node, use another *SetDestination* macro with the new target data.

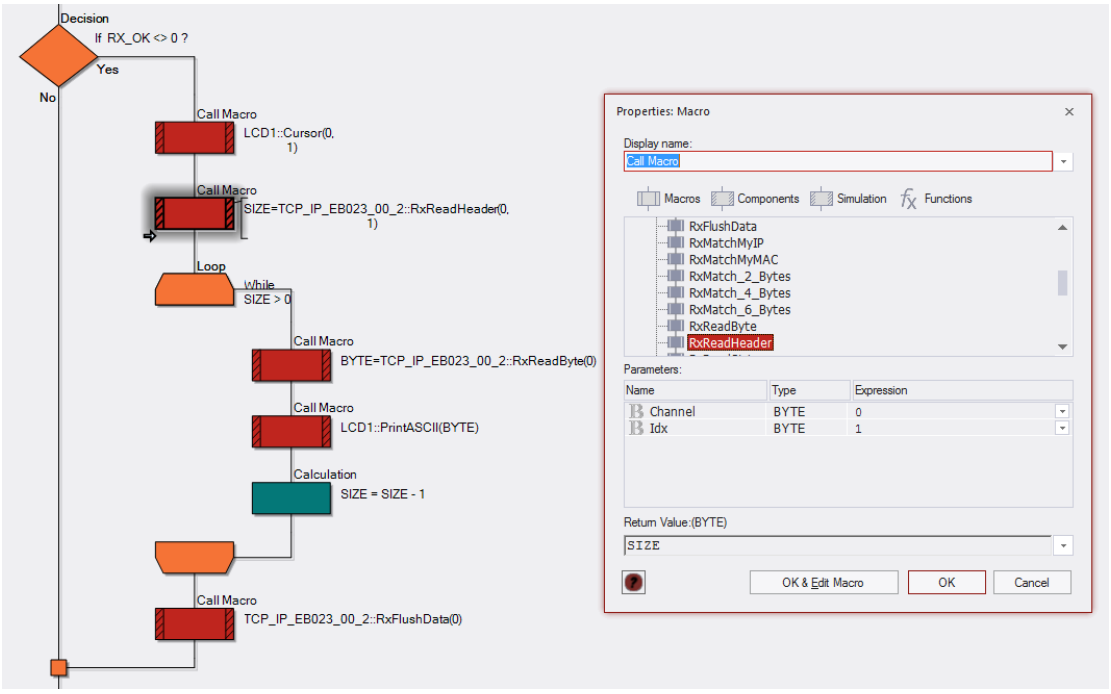
Sending data

Once we have connected the UDP socket we can simply send data straightaway.
We do not need to set up headers or any other wrapping information. We simply send the data using the TxStart, TxSendByte and TxEnd macros.



Receiving data

Receiving is relatively straightforward as well.
We can perform the usual *RxDataAvailable* test to look for fresh data, and read it in using *RxReadByte*.
There is however one thing we need to know. Firstly, how much data is there?
We can get the Message length header item using the *RxReadHeader* macro. The data from item index 0 of the UDP header gives the high byte of the message length, and item index 1 gives the low byte of the message length.
Using *RxReadHeader* we can put this into convenient variables such as SIZE_HI and SIZE_LO.
Once we have the message length we can loop through and *RxReadByte* it in ready to be dealt with.



Once we have the data we need to deal with it. This will of course depend on what the data is (e.g. sending ASCII text to the LCD as in the code snippet above). As UDP is custom data we will need to know what the data is supposed to be. And we may need to check to see if the data is what we expected and not an entirely different message.

Remember to flush the buffer once you have finished so that the next message can arrive.

Using UDP

The key point to grasp here is the fact that you could be responsible for dealing with everything, including processing the raw data. If you want error checking or responses than you will need to do that yourself in the program. However, for custom programs that is exactly what you want – freedom to pass data or messages as you see fit. The data is what you want, not a preconfigured datagram like in ARP or ICMP.

There are some protocols that use UDP as a base, such as DHCP and DNS. To communicate with and use these protocols you will need to look into details of how the data needs to be structured for them. You may also need to find out what replies are sent, and how those replies are formatted.

Although relatively simple, the UDP protocol allows direct communication between two systems that only need to be connected via a network, or linked to the internet. Physical distance then becomes irrelevant. This comes at a cost though. You need to know the correct IP address and port details to be able to send the message or it will simply disappear. Also there needs to be an application running on the other end to read and respond to your message. If there is not, then nothing will happen.

Whilst this is fine for custom programs, it is not robust enough for global communications standards such as SMTP email, or HTTP web pages. What we really need is something that is a bit more robust, able to check for errors, able to respond to prompts or requests, and able to handle larger messages automatically. This is where protocols such as TCP, with defined communication sequences, come in.

Exercise 3: Time and date using UDP mode

In this Exercise we will retrieve the Time and date string from Socket 13 on a target PC.
Socket 13 will respond to a UDP message by returning the Time and date as an ASCII string.
Note that you will need to set the Destination to the IP of the Target PC.

Program objective:

- Use UDP to retrieve the time and date string from a PC.

End if no response received after a 'timeout' period

Prerequisites:

- Knowledge of the MAC/Ethernet layer

Knowledge of the IP layer

Required Information
IP address of target PC

Learning objectives:

- Structure of a UDP Datagram
- Sending UDP messages
- Receiving UDP messages
- Preset socket responses

Notes on Exercise 3: Time and date using UDP mode

We will not be providing a worked example for Exercise 3. Instead we will be offering notes and advice on areas of potential difficulty, and providing items such as Program flowcharts that may be of use in collating handouts.

Firewall warning

This exercise requires access to Port 13 in order to work. However Port 13 may be blocked by firewall software, or may not automatically respond to messages for a variety of reasons. It is recommended that the port is checked for availability. If port 13 is not available you may need to do one of the following:

- Contact your system administrator to set up access to port 13
- Use another available port

Use another internet solution set up to respond to messages on port 13 to mimic the Time Date function.

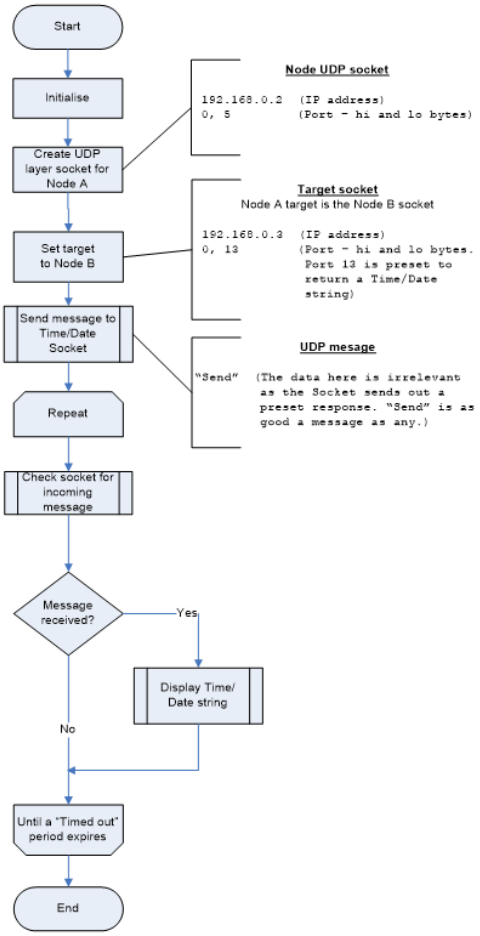
General notes

The target system will need to be a one that is capable of responding to a request on Socket 13, such as a standard PC. If in doubt as to whether a socket is producing a response Packet injector, such as Excalibur, and Network traffic analyzers, such as Wireshark, can be used to test the socket.

A socket is required for the UDP connection. This can be on any port, but you should be aware of and avoid using preset sockets such as Port 25, the SMTP port.

Program flowchart

The basic structure for this program is quite simple as shown on the flowchart below.



Example program

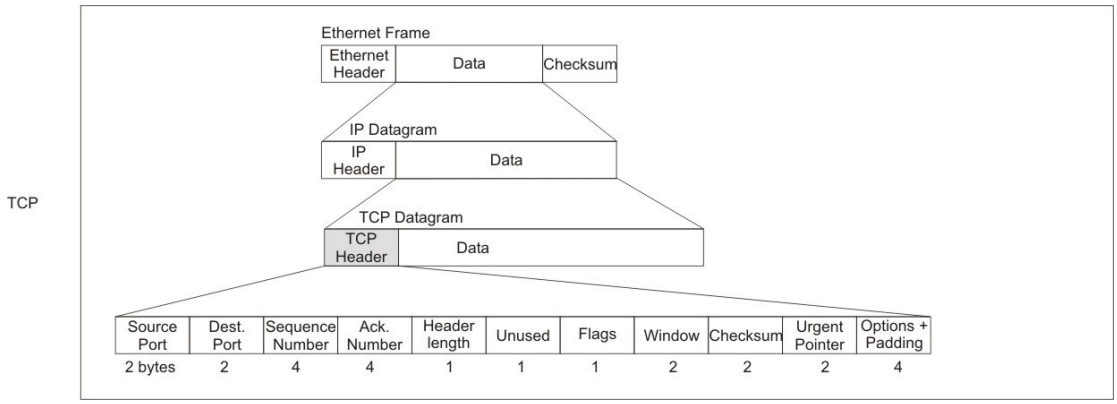
An example solution can be found in the Flowcode “TCP_IP\Examples” folder, or in the Examples section of the CD, for you to refer to, use for code, and use for demonstration purposes.

Note that properties, macro parameters and variable values etc. were set up for use with our network and may require changing to match your own network or system configuration.

TCP

To the general user TCP is what brings in the email message, or the HTML page. TCP is often thought of as the code that these programs use. Which in one respect it is (as that's the data that it carries), but in another it is not. TCP is the process of controlling the transmission of that data. To TCP the data is irrelevant; it's the sending that counts. It's the applications that pick up the data from TCP that care if it is email or HTML.

Preset sockets are used extensively in TCP. Email applications will listen in on one port, HTML programs on another. FTP and other programs will scan other ports. If you want to you can even build a custom program to scan a particular port and send custom data to it using TCP.



IP header

	Description	bytes	Notes
Source port	The port that the sending TCP socket is connected to.	2	
Destination port	The destination port to which the message will be sent.	2	
Sequence number	Used with fragmentation to aid reassembly of the datagram.	4	
Acknowledgement number	Used with fragmentation to inform the sender of successful reception of the datagram.	4	
Header length	Length of the header in bytes.	1	
Unused	Unused. Reserved for expansion.	1	
Flags	Used to store bit flags for FIN, ACK, SYN, Reset and Push.	1	
Window	The size of the buffer for incoming messages.	2	
Checksum	A checksum value for the TCP header.	2	
Urgent pointer	The IP address of the destination node.	2	
Options and padding	Various options are available for the TCP header. We will not be using any options in this course, so this section can be safely ignored for now.	4	

Connections

Connections are the key to understanding TCP. Using TCP revolves around controlling and coordinating the connection. It is a bi-directional process, with the two parties engaged in a dialogue rather than a send, reply, reply to the reply etc. system.

Acknowledgements

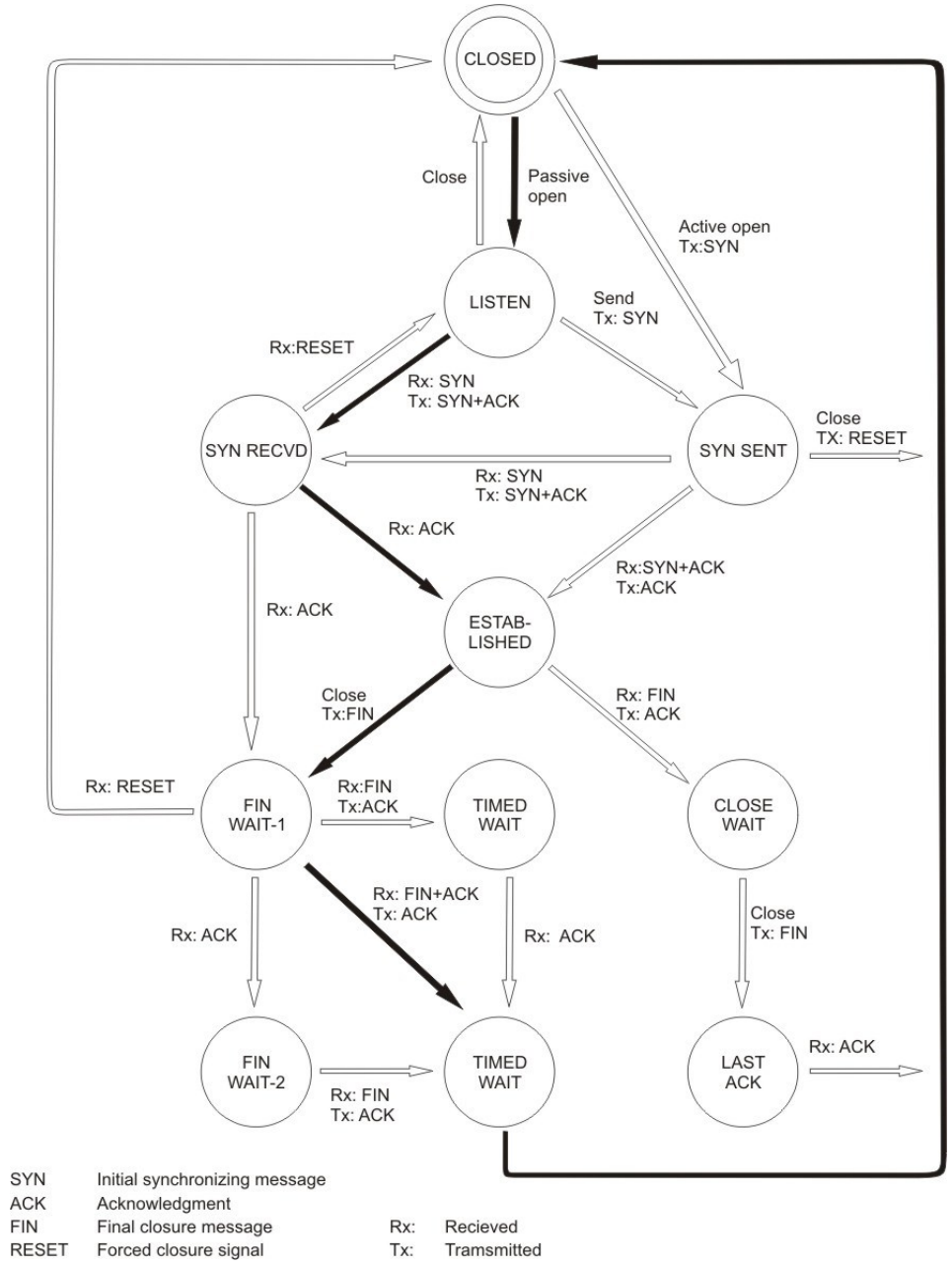
TCP is useful in that it will acknowledge the message as it arrives. In fact you can check, send, check, send etc. in TCP to get the data sent whilst checking it has been received at the same time.

Some systems such as SMTP requires an active dialogue with data sent in distinct steps with acknowledgments sent at specific stages, and various tags or markers sent to distinguish parts of the data being sent.

Fragmentation

Large amounts of data can be easily sent using TCP. TCP allows for messages to be split into fragments that can be sent as IP datagram's. TCP can then send the IP datagram's one by one (lockstep method), in numbered blocks (block sequencing method), or by byte count (byte sequencing) where both sides keep track of the amount of bytes sent and received.

The fragments are marked with a sequence number to aid reconstructing the message at the destination. This allows fragments to arrive out of sequence without damaging the message data. It also aids in checking for and resending lost fragments.



The aim for an application is to move from CLOSED to an ESTABLISHED connection. The route there depends on whether the connection is opened passively, or actively. Once a connection is established the applications can communicate between each other until a FIN final closure message is sent by one of the parties. Whereupon a closure sequence starts as the applications seek to move back to the CLOSED state. At all stages there is the possibility of communications failure leading to a RESET back to CLOSED.

To establish a connection a SYN message needs to be sent by the party initiating communications. Once a SYN and an ACK have been received an ACK is sent and communications established.

Closing a connection is similar. A FIN is sent by the party wishing to end the connection. A FIN and an ACK are expected in response, and a final ACK finishes it all off.

Whilst the state diagram seems complex at first following through the flow shows that the different paths are just a matter of who initiates contact, and who terminates it and in which order they send the required signal. The following sections demonstrate some of the main pathways through the state diagram in action.

Passive open

Applications can opt to move to the Listen state where they can listen for SYN requests or if they wish send a SYN request of their own to go active moving them to the SYN SENT state as for active open.

When an incoming SYN requests is received they send a SYN+ACK to indicate that they are ready to communicate and move to the SYN RECVD state waiting for a final ACK before moving to ESTABLISHED.

Active open

In active open the application initiates communication with a SYN message moving it from CLOSED to SYN SENT. If no response is received a RESET is sent and the application goes back to closed. If a SYN+ACK is received i.e. the other application is ready and waiting then a connection has been made and the process moves to ESTABLISHED. A third possibility is that the application receives a SYN message, but no ACK, in which case the application needs to transmit a SYN+ACK message and move to SYN RECVD and wait for an ACK before it can move to ESTABLISHED.

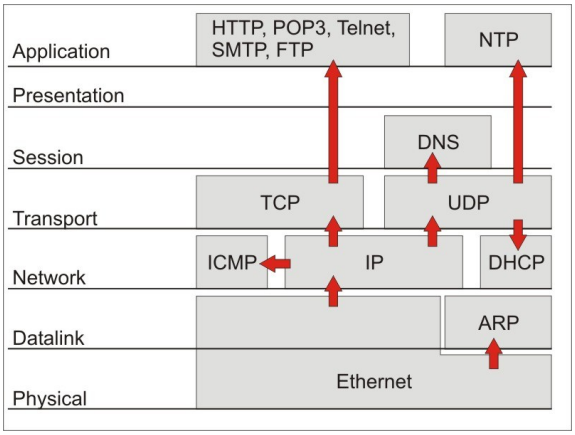
Closing the connection

To close a connection a FIN needs to be sent to inform the other application that you wish to end the connection. Once a FIN has been sent the application needs to wait for a FIN and an ACK to be received to indicate that the other application is ready to close the connection. The FIN and ACK can come is any order so there are several possible routes though the various states until the application can move to CLOSED. If no messages are received the application can also RESET as the chances are by that stage communications have broken down.

Responding to a close request

When an application receives a FIN message it needs to respond with an ACK and a FIN message. Once these have been sent, moving the application through CLOSE WAIT to LAST ACK, the application waits for this last ACK before moving to CLOSED.

TCP bring you into the OSI model at the transport level. Here we are concerned with getting the data to the applications, and with error checking that process.



The basics of TCP mode include:

- Setting up a socket Connects to a Socket that other nodes can communicate with the program on.
- Actively connecting or passively listening

o If not actively initiating communications you can passively listen for a message and reply when spoken to.

o In active mode you will need to set which socket you wish to send to.

o Sending data The *TxStart*, *TxSendByte* and TxEnd macros are used to send data transmission in the same way as with previous modes.

Checking for and responding to messages handled in the same way as all the previous modes with *RxDataAvailable*, *RxReadByte*, *RxMatchXXXX* and *RxFlushData* being used to check for and process the incoming data.

> Note that like IP and UDP you will not normally need to create the header as it is handled automatically by the TCP/IP component.

Initializing a TCP connection

You can create a TCP connection with the *CreateTCPSocket* macro, specifying which channel to use, and the port number bytes. TCP can use up to 4 simultaneous channels, channel 0-3.

Active connection with TCPConnect

If you wish to initiate communications you will need to connect to the appropriate socket on the other node. To connect for active transmission you need to call the *TCPConnect* macro with the channel, IP address bytes and port number bytes. *TCPConnect* returns 1 if a connection can be established and 0 if the connection could not be established. *TCPConnect* handles the various SYN and ACK messages required to establish connection.

Passive connections with TCPListen

TCPListen allows you to monitor a port for messages. You can remain silent and passive whilst monitoring this port. Once a message has been received a connection is automatically established by the TCP/IP component and you can reply to the sending socket.

You can use the *GetSocketStatus* macro monitor the socket status to see if any external application is trying to connect. The help file section on *GetSocketStatus* lists the various states and their return values. Note that many of the variables relate to the various states in the state diagram (the others mostly being concerned with forms of data transfer). The three most important states for monitoring are SOCK_CLOSED – value 0, SOCK_ESTABLISHED – value 6 and SOCK_CLOSE_WAIT – value 7.

Sending data

The *TxStart*, *TxSendByte* and TxEnd macros can be used to perform the actual data transmission once a connection has been established. Note that many TCP based applications expect data to be sent in a specific sequence.

Checking for and receiving data

As with the previous modes with RxDataAvailable is used to check for incoming data. RxReadByte, RxMatchXXXX and RxReadHeader can be used to extract and test the data. RxFlushData is used to clear out current data ready for the next message.

Closing a connection

TCPClose handles the closure sequence both actively or passively depending on whether you or the receiving application has initiated closure.

Once you have finished with the TCP connection you need to actively close it down. Call the *TCPClose* macro to close the connection. TCPClose will handle the various FIN and ACK messages that are required to close the connection

If the connection needs to be closed by the receiving application it will transmit a FIN message which will put your application's socket into the SOCK_CLOSE_WAIT state. You can monitor the socket status using *GetSocketStatus*. Once SOCKET_CLOSE_WAIT is detected you should call the *TCPClose* macro to close the connection.

<CRLF>, quotes and other problem characters

Whilst most of characters can be typed into the *TxSendByte* macro there are some extra ones we need to allow for. We cannot send the Return/Enter key to get to the next line as it doesn't go into the parameters list. This is because the Return key is a non-printable control character. However all characters, including control character have an associated ASCII value that we can use instead. The Return key is a bit unusual as is actually added as two character codes rather than just one – the Carriage return character (character code 13) and the Line feed character (character code 13). This is often abbreviated to CRLF. All we need to do is send the ASCII character codes instead, in this case ASCII codes 13 and 10 at the end of each line of email code. (This is all a hang up from the early days of computing when outputting was on done electronic printers where you actually moved the printer head around.).

The SMTP response messages tell you to end the email data with the <CRLF>.<CRLF> message tag i.e. a full stop on a line on its own. This would need to be sent as characters 13, 10, '.', 13, 10.

Another problem character is the " quote character. The quote character is used by Flowcode to distinguish between strings e.g. "Hello world" and variables e.g. COUNT. So we can't send the " character as it will confuse Flowcode. Instead we need to send the ASCII code for ", which is ASCII character code 34.

Communication sequences

TCP communications are generally sequence based with a message requiring a reply, which could be either dealt with, or responded to. Another reply would then be needed and so on until the sequence is complete.

To communicate successfully with TCP applications it is necessary to know what these sequences are, what responses to expect and what error codes will be sent. Some, such as an HTTP GET request can be a simple response. Others such as SMTP may require a number of steps in the sequence with specific responses being required or being sent at various stages.

It is the understanding of communications sequences that will unlock TCP applications. And it is the sequences that we will need to look at in the TCP exercises and examples.

Exercise 4: Sending a HTML page using HTTP

Instructions

Create a basic web page that can be displayed on the PC in a web browser.
We will be using just a single page that is returned for all GET requests for this exercise so we do not need to check what page is requested.

Program objective:
Send HTML data when a HTPP GET request is received.

Prerequisites:

- Knowledge of the MAC/Ethernet layer
- Knowledge of the IP layer
- You will need PC with an internet browser connected to the network in order to view the HTML page.

A network traffic analyzer will also be useful for this particular task to aid in debugging the data being sent and received.

Required information
HTML code to be sent. The following is a sample fragment of HTML that you can send:

```
HTTP/1.0 200 OK
Content-type: text/html

<html>
<head>
<title>TCP/IP Comp Web page</title>
</head>
<body>
<b>Hello World</b>
<p>How are you today?</p>
</body>
</html>
```

Learning outcomes:

- Structure of a TCP Datagram.
- TCP state diagram
- Create a passive listening TCP connection
- Sending TCP messages.
- Receiving TCP responses.
- Responding to a message.
- The structure of the GET request
- Basic HTML code

Notes on Exercise 4: Sending a HTML page using HTTP

This section contains notes and advice on areas of potential difficulty, and provides items such as Program flowcharts that may be of use in collating handouts.

Prerequisites

You will need PC with an internet browser connected to the network in order to view the HTML page. To view the HTML you will need to point your browser at the page on the node. Open up the browser and go to the IP address that you set up for the TCP/IP component (for example **http://192.168.0.2**). The default page will be sent for all requests so we do not need to worry about the page name.

Useful tools

A network traffic analyzer will also be useful for this particular task to aid in debugging the data being sent and received. The network traffic analyzer will also allow you to monitor the TCP state and see the various SYN, ACK and FIN messages sent.

Sample HTML

The following is a sample fragment of HTML along with the header section for it.

```
HTTP/1.0 200 OK
Content-type: text/html

<html>
<head>
<title>TCP/IP Comp Web page</title>
</head>
<body>
<b>Hello World</b>
<p>How are you today?</p>
</body>
</html>
```

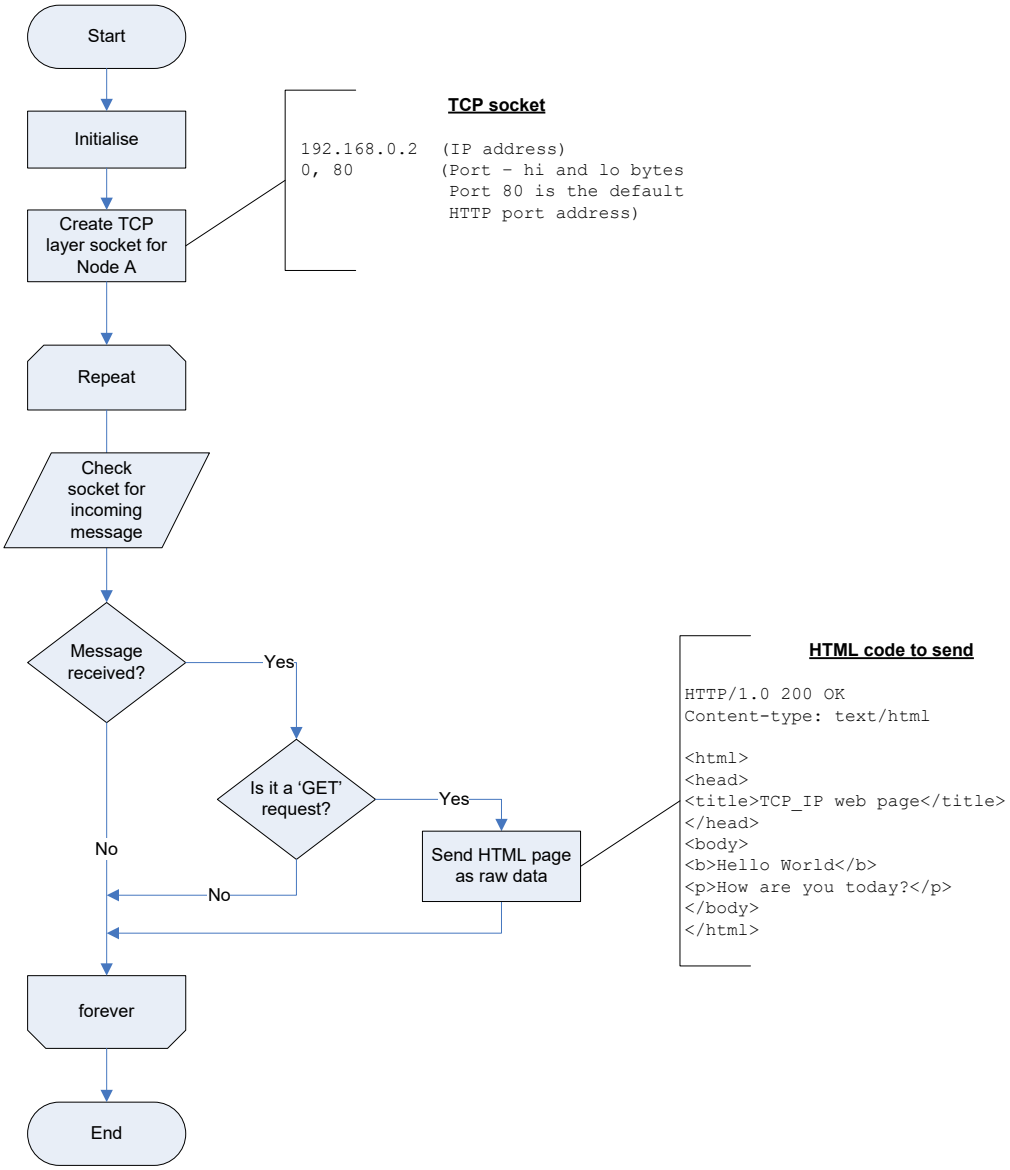
The first two lines are a HTTP header to inform the browser application of the format and type of message being sent. The rest of the data is the HTML code for the web page. (HTTP header information and HTML code tutorials can be found on the internet.)

We need to break up the HTML code to put it into the TCP datagram.
We will need to send the character codes 13 and 10 at the end of each line for the CRLF return key ASCII values. (For your own HTML code other ASCII character codes, such as ASCII 34 for quotation marks, may also be required depending on your HTML code.)

Notes on Exercise 4: Sending a HTML page using HTTP

Program flowchart

The overall structural elements of the program are as follows.



Creating the connection

HTML is normally sent to Port 80, so we will need to set this up in the *TCPConnect* macro (*dst_port_hi* = 0, *dst_port_lo* = 80).

We are not initiating communications so we can use passive mode with *TCPListen*. The socket can then be checked using *RxDataAvailable* to see if a request has arrived that needs responding to.

Checking for the web page request

A HTML transfer is initiated with a GET command being sent to the node containing the web page. E.g.:
GET FOLDER/PAGE.HTM HTTP 1.0

RxMatch_4_Bytes can be used to check if this is indeed a “GET” message. (Note that we need to check for the space as well to make the four characters needed for the macro. A small thing, but one so easily overlooked.)

Notes on Exercise 4: Sending a HTML page using HTTP

As we are sending the same page for all requests we can ignore the page request part of the GET request for this program.

Sending the data

The actual sending is easy, just the same *TxSendByte* commands we have used previously. Don't forget the *TxStart* and *TxEnd* macros though.

Closing the connection

Once the data has been sent we need to close the TCP connection to end the transmission process with the *TCPClose* macro. If the web page is a one off we can simply close the connection and end the program. However if the web page is meant to be accessed multiple times we need to re-establish the connection ready for the next request. To do this we can call the *CreateTCPSocket*, and *TCPListen* macros again.

Spicing up the page

The example HTML code is a bit bland, but can be spiced up easily.
Change <body> to <body bgcolor=0000ff text=ffff00> for example and the page will become a bit more colorful. Note that the objective is to send HTML data, not to create an artistic page so layout is secondary to getting the program to work.

Suggestions for further work

Variable markers

Browsers frequently store received HTML in a cache which can be used instead when the page is unobtainable. This can cause problems in testing, especially if we have changed the HTML code. Pressing refresh forces the browser to get a fresh copy of the page and usually fixes the problem. However having a marker of some kind on the page that alters each time it is sent is useful to check that the page is being retrieved correctly and not just brought from the cache. For our example adding a variable which is updated each time the page is accessed.

Multiple pages

The basic program outlined above sends a single page in response to a GET request.
An interesting project could be to expand the system to handle requests for a number of pages.

Handling errors with multiple pages

If you have a system with multiple web pages, and the code to check which page to send you should ideally create a default 'File not found' page to send if a non-existing page is requested. This is the infamous 404 page error regular web users will be very familiar with.
For example:

HTTP/1.0 404 Not found
Content-type: text/plain

File not found.

Images and other files

We haven't covered images or other permissible file types as the small size of microcontroller memory would make storing image data difficult. But the principle is the same. Send the header with the file type, and then the data.
For example:

HTTP/1.0 200 OK
Content-type: image/gif

FF006735BBCC234788…….. etc.

Notes on Exercise 4: Sending a HTML page using HTTP

Example program

An example solution can be found in the Flowcode “Flowcode\TCP_IP\Examples” folder, or in the “Examples” section of the CD, for you to refer to, use for code, and use for demonstration purposes.

> Note that properties, macro parameters and variable values etc. were set up for use with our network and may require changing to match your own network or system configuration.

Exercise 5: Receiving HTML

Instructions

Request a page from a web server and display a portion of the HTML code returned on the LCD display.

Program objective:

- Retrieve HTML data with a HTPP GET request.

Display the incoming data on the LCD display

Prerequisites:

- Knowledge of the MAC/Ethernet layer

Knowledge of the IP layer

Required information

IP address of the server for requesting the page

Learning objectives:

- Structure of a TCP Datagram.
- TCP state diagram
- Creating a TCP connection
- Sending TCP messages.
- Receiving TCP responses.
- Responding to a message.
- The structure of the GET request

This section contains notes and advice on areas of potential difficulty, and provides items such as Program flowcharts that may be of use in collating handouts.

Prerequisites

You will need a Server capable of sending the page that is requested. You may wish to test the page is available in an internet browser. A simple page is preferred as a large complex page with multiple parts and images may be too much for the basic program to handle.

A Sever is simply a system that is capable of 'serving up an HTML web page' when requested. The previous exercise – Sending HTML using HTTP is an example of a rather basic server. Although generally Servers are large sophisticated computer systems that perform a myriad of other network tasks as well.

Useful tools

A network traffic analyzer will also be useful for this particular task to aid in debugging the data being sent and received. The network traffic analyzer will also allow you to monitor the TCP state and see the various SYN, ACK and FIN messages sent.

Receiving the HTML

Receiving the HTML is fairly straightforward; however the microcontroller does not have much memory, or much in the way of graphics abilities (even with an LCD display) which will limit how you can deal with the resultant data.

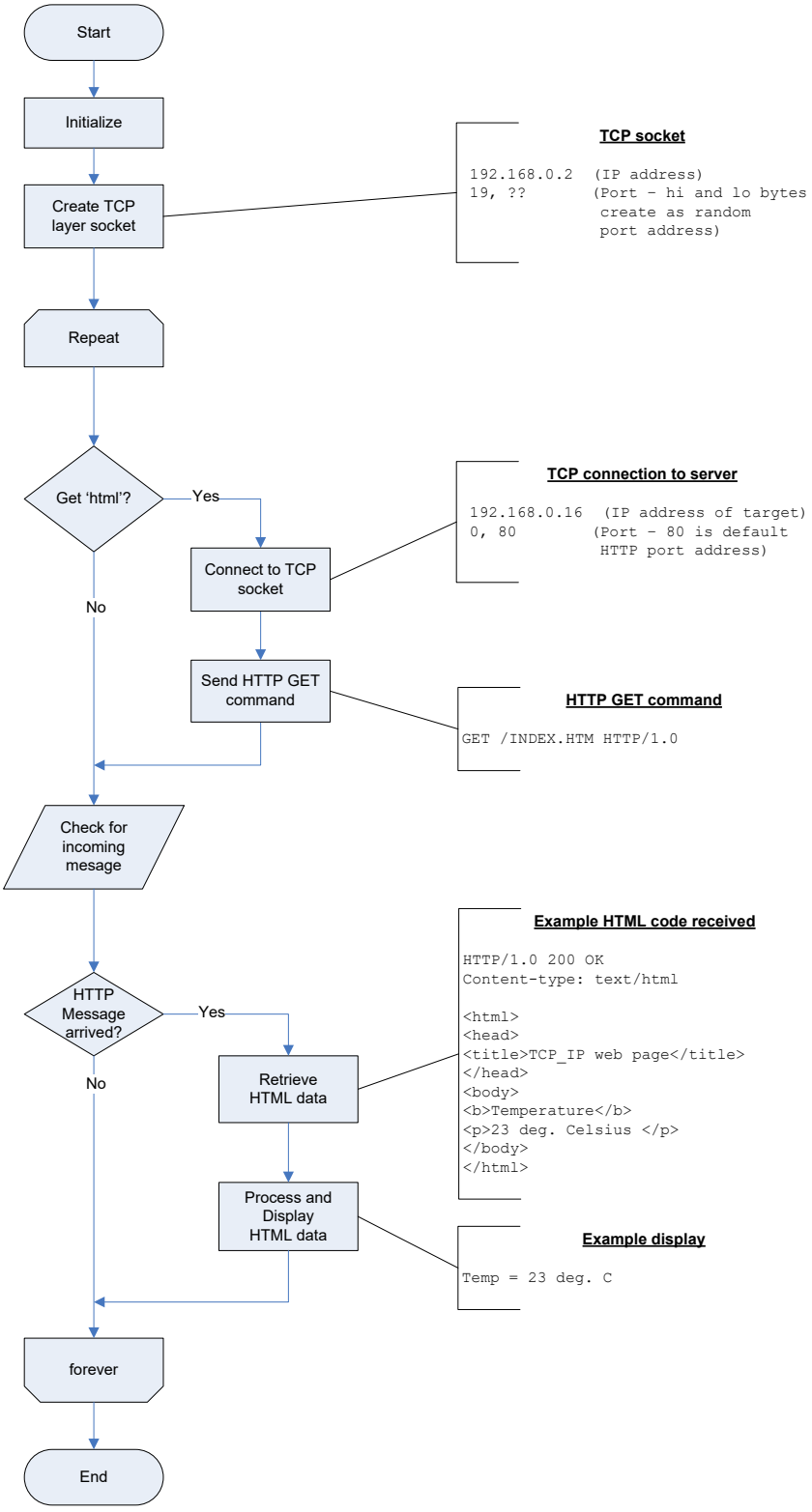
Displaying the data is needed to show that we have indeed received the web page. However what data to display is open for choice. For example:

- The data can be displayed as received allowing it to overflow the LCD display.
- A specific amount of data can be displayed e.g. enough to fill the LCD.
- A specific tag or piece of text can be searched for and a specific piece of HTML text extracted from the code.

Program overview

The basic process can be summed up in the following flowchart:

Notes on Exercise 5: Receiving HTML



Requesting the HTML page

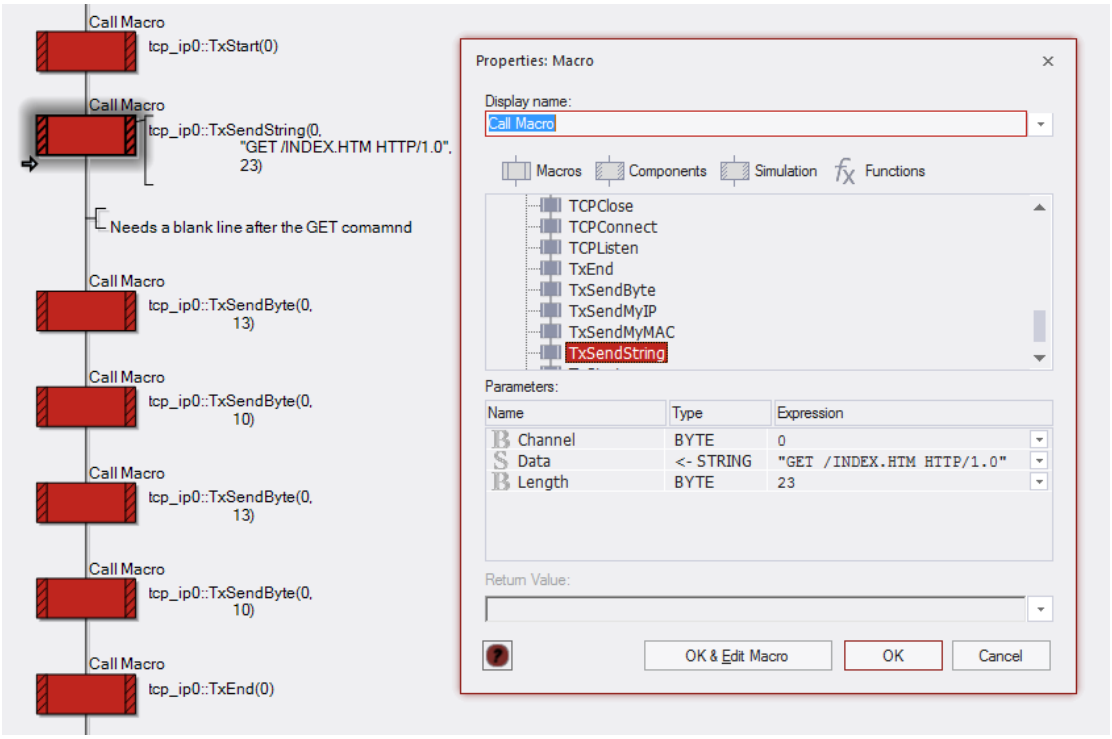
To get a web page you need to initialize the TCP/IP component and then connect it using the *TCPConnect* macro to the IP address of the server that has the web page, and to the server's HTML port – usually Port 80 E.g. Channel 0, IP 192.168.0.17 Port 0, 80.

To receive a page of HTML you need to request the page that you want from the server with a GET request with page name, and the HTTP version (we can use HTTP/1.0 for this). The GET request is

followed by a blank line, so we will need to CRLF to the next line and CRLF again to produce the blank one.

GET FOLDER/PAGE.HTM HTTP 1.0

The whole process can be seen in the image below.



Once the Server receives the GET command it will start sending the data.
At this point we need to start checking for incoming data and read it in when it arrives. We may not be able to store all the data due to the relatively small size of the microcontroller's memory, so we will need to work through it and either retrieve the data there and then and display it, or do whatever processes we need to do with it, before we move onto the next batch of incoming data. If the incoming HTML data is particularly large it may arrive in more than one message.

Further work

Now that we can get HTML data in, we can process it. This opens up all kind of possibilities. If we know that a set piece of data – e.g. a temperature reading, will be in the data in a tag or with a set wording that we can search for, retrieve and display that specific information using the *RxMatchXXXX* macros. For example if the HTML sent on a page contained the information "Temperature = ?? deg C". We could expand the program to search for the temperature value and to display the value on the LCD.

Example program

An example solution can be found in the Flowcode "Flowcode\TCP_IP\Examples" folder, or in the "Examples" section of the CD, for you to refer to, use for code, and use for demonstration purposes.

Note that properties, macro parameters and variable values etc. were set up for use with our network and may require changing to match your own network or system configuration.

Instructions

Create a basic email program that sends a predetermined email to a preset email address. (We wish to test the communication side of SMTP so a simple preset message will suffice.)

Program objective:

- Create a TCP connection.
- Open a structured dialogue with the mail server.

Error checking dialogue responses.

Prerequisites:

- Knowledge of the MAC/Ethernet layer
- Knowledge of the IP layer

Knowledge of TCP (as gathered from the previous TCP examples)

Required Information

- IP address of SMTP email Server

Useable email address on the network that can be used to check the message.

Learning outcomes:

- TCP connections
- Performing structured TCP dialogues
- Error checking responses
- SMTP data formatting

Notes on Exercise 6: Sending an SMTP email message

The notes here detail the sequences involved in sending an SMTP email. This section is quite extensive compared to the notes for the previous TCP implementations as the sequence involves a number of steps and requires specific relies at specific stages. However it is a very worthwhile task to complete as many applications that use TCP communications require similar complex dialogues.

SMTP email

Email is something we are all familiar with. Modern email systems can be quite fancy with graphics, formatted text, and attachments, but at their heart is the original simple text based system. The flashy bits are just different ways of formatting the data. At its most basic email is just a bit of text with some preliminary information to tell you where the message is going, who is sending it, type of message, options etc. In fact it is pretty much like the Header/Data split we have been looking at in the TCP/IP datagram's.
An email is sent as a TCP datagram to the SMTP socket on the server.
The big difference between sending a TCP datagram to the SMTP email socket and sending a UDP datagram is that the SMTP socket expects there to be a dialogue. You say Hello, it responds to let you know it's ready for you. You tell it the 'To' and 'From' information and it will ask for the email message etc. It will even say goodbye once you have finished the message.
The core of this section will be about explaining the system, not about explaining the code. You should already be at a level where you can create, test and debug the code itself.

Key transmission messages

Sending the message consists of sending key messages, waiting for the correct acknowledgement, and proceeding to the next step. The following are certain key messages that alert the SMTP socket about what stage the message transmission is at.

- HELO [System name] – an introduction message to get the message sequence started.

Note: This is correct it is HELO not HELLO. All the sequence dialogues start with a four character word (except for '.').

- MAIL FROM:<me@myaddress.com> - the address the email is being sent from.
- RCPT TO:<me@myaddress.com> – the recipients email address.
- DATA – a marker to indicate that you are ready to send the data. This will be followed by the message text itself.
- . – a single full stop on a line of its own. Used to indicate that the message data has been sent.

QUIT – a message to terminate the process.

SMTP Acknowledgment codes

At various times during the TCP communication the client will respond with an acknowledgement. These can be various messages such as an introduction, an echo of the data sent, or a data ok message. The messages are prefixed by a three digit code that we can validate to check that the process is working ok. The code number and the acknowledgement data contains a lot of useful information for error checking and troubleshooting. Here we will just concentrate on the responses we are expecting. Further details on SMTP response codes can be found in SMTP guides and specification, which can be found on the web.
The ones we will use in sending the email data are:

- 220 [Server details] – the server introduces itself.
- 250 [Message data repeated] – the server acknowledges the data sent and returns it so it can be verified if need be.
- 354 Ok send data ending with <CRLF>.<CRLF> - the server is now ready to receive the message data.
- 221 [Server name] closing connection – a QUIT message has been successfully received.

Notes on Exercise 6: Sending an SMTP email message

Sending an Email message step by step

The following walkthrough will take you through sending a simple email message.
The data send, and the client acknowledgements for a typical will be shown in sequence.

- TCP connection started.
- Response: 220 [Server Introduction and details message]
- HELO MY_NAME
- Response: 250 [Server name]
- MAIL FROM: <me@my_address.com>
- Response: 250 Sender <me@my_address.com>
- RCPT TO: <you@your_address.com>
- Response: 250 Recipient <you@your _address.com>
- DATA

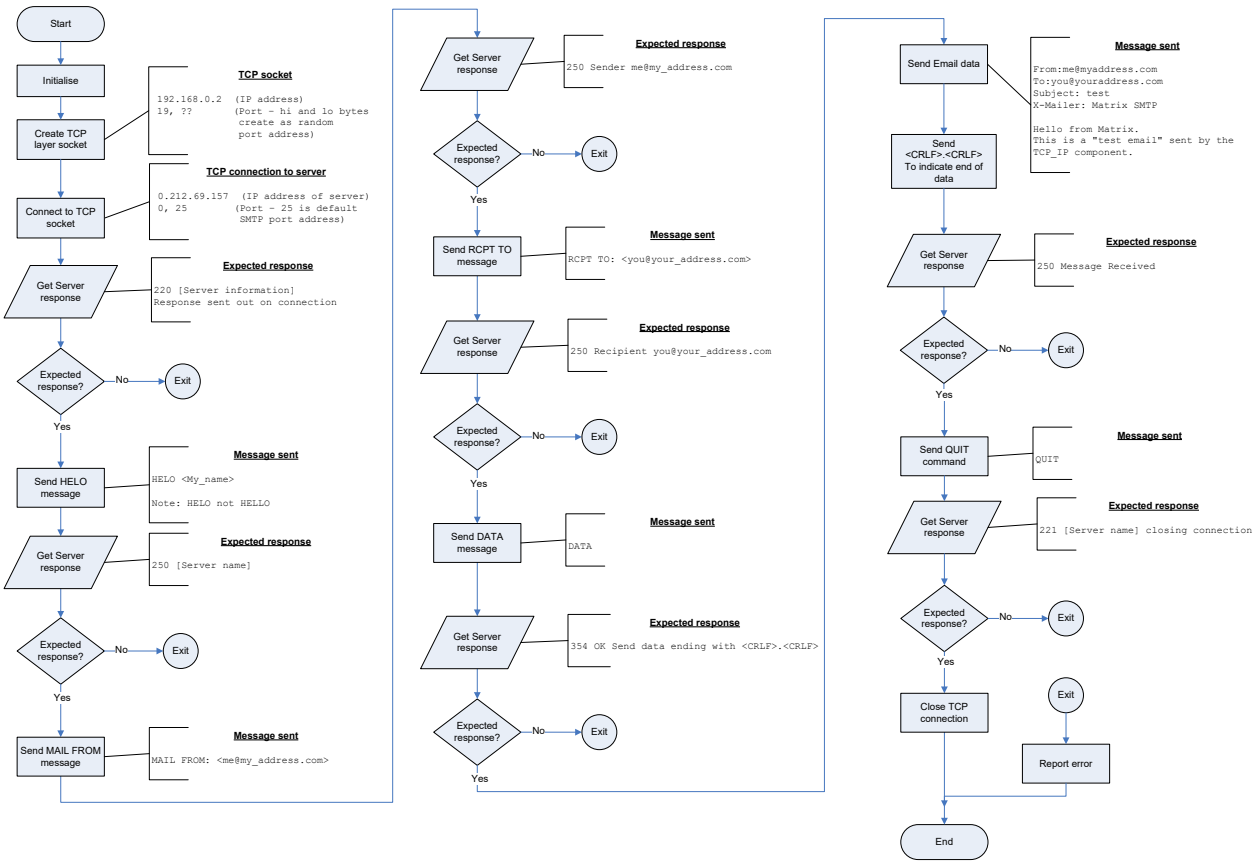
Response: 354 Ok send data ending with <CRLF>.<CRLF>

…. The email message data is sent

- . (this is the <CRLF>.<CRLF> end tag mentioned above)
- Response: 250 Message Received …..
- QUIT

Response: 221 [Server name] server closing connection

Below is a flowchart showing an example of this process in action:



Notes on Exercise 6: Sending an SMTP email message

Sample email message to send:

Emails, like HTML can contain a variety of features and options. However for our purposes (i.e. TCP communications) a basic plain email will suffice.

From:me@myaddress.com
To:you@youraddress.com
Subject: test
X-Mailer: Matrix SMTP

Hello from Matrix.
This is a “test email” sent by the TCP/IP component.

Notice how the ‘*From’* and ‘*To’* addresses are added in even though they were sent to the server as part of the connection and acknowledgement process. There are a lot more header items that can be added as well, but the above should suffice for basic messages. For more details on other header items you can check for SMTP protocol details on the web.

Implementing the SMTP program in Flowcode

Hardware considerations

First thing to consider is where the email server is located on your system.
On our test system the email server was at IP address 212.69.254.157 but you will need to find out what IP address to use for your network. Ask the Network administrator if you are not sure how to find it.

SMTP messages are usually sent to Port 25 so you will need to set this up in the *TCPConnect* macro (*dst_port_hi* = 0, *dst_port_lo* = 25)

Next you need two valid email addresses: a ‘From’ email address and a ‘To’ email address. The ‘To’ email address definitely needs to be a valid email address that you have access to otherwise you will not be able to receive the emails to test and debug the program.
The From email address can be a spoof or invalid test email address, however items such as firewalls may perform checks and reject such emails. It is easier to set up a testing email address if this is possible.

The only issue at this stage is getting the sequence right. By now you will have the knowledge and experience to create the code required for the actual sending and receiving of messages.
With SMTP we are no longer concerned with sending just a message; we are now having dialogues and negotiating communications.

<CRLF>, quotes and other problem characters

Whilst most of characters can be typed into the *TxSendByte* macro there are some extra ones we need to allow for. We cannot send the Return/Enter key to get to the next line as it doesn’t go into the parameters list. This is because the Return key is a non-printable control character. However all characters, including control character have an associated ASCII value that we can use instead. The Return key is a bit unusual as it is actually added as two character codes rather than just one – the Carriage return character (character code 13) and the Line feed character (character code 13). This is often abbreviated to CRLF. All we need to do is send the ASCII character codes instead, in this case ASCII codes 13 and 10 at the end of each line of email code. (This is all a hang up from the early days of computing when outputting was on done electronic printers where you actually moved the printer head around.).

The SMTP response messages tell you to end the email data with the <CRLF>.<CRLF> message tag i.e. a full stop on a line on its own. This would need to be sent as characters 13, 10, ‘.’, 13, 10.

Another problem character is the “ quote character. The quote character is used by Flowcode to distinguish between strings e.g. “Hello world” and variables e.g. COUNT. So we can’t send the “ character as it will confuse Flowcode. Instead we need to send the ASCII code for “, which is ASCII character code 34.

Notes on Exercise 6: Sending an SMTP email message

Example program

An example solution can be found in the Flowcode “Flowcode\TCP_IP\Examples” folder, or in the “Examples” section of the CD, for you to refer to, use for code, and use for demonstration purposes.

> Note that properties, macro parameters and variable values etc. were set up for use with our network and may require changing to match your own network or system configuration.

Advanced Exercise 1: Custom messaging using UDP

This is an advanced exercise as it requires two fully set up internet board sets and either a hub, or two available sockets on a network.

Program objective:

- Create two Nodes which will respond to communication by sending the communicator a reply message.
- Node A
  - Can initiate when ‘Send button’ is pressed
  - Will respond to a signal by sending a response
  - 192.168.0.2
  - Port 5
  - Message: “Hello world”
  - Node B
  - Can block contact when ‘Off button’ is pressed
  - Will respond to a signal by sending a response
  - 192.168.0.3
  - Port 10

Message “Hi globe”

Prerequisites:

- Knowledge of the MAC/Ethernet layer

Knowledge of the IP layer

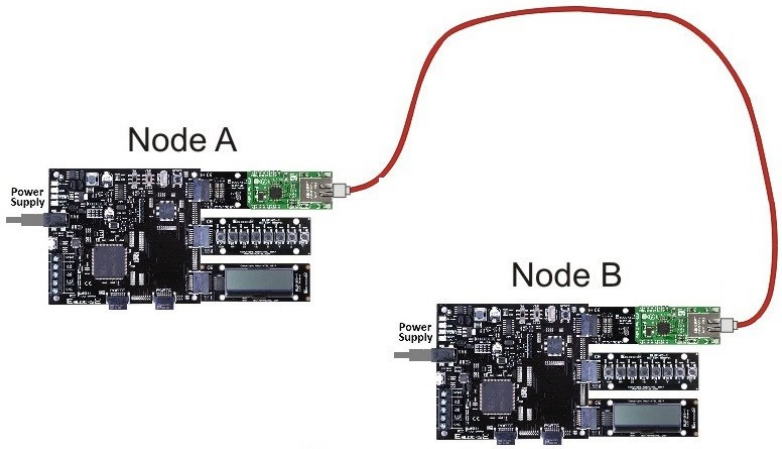
Learning objectives:

- Structure of a UDP Datagram
- Sending UDP messages
- Receiving UDP messages
- Responding to a message

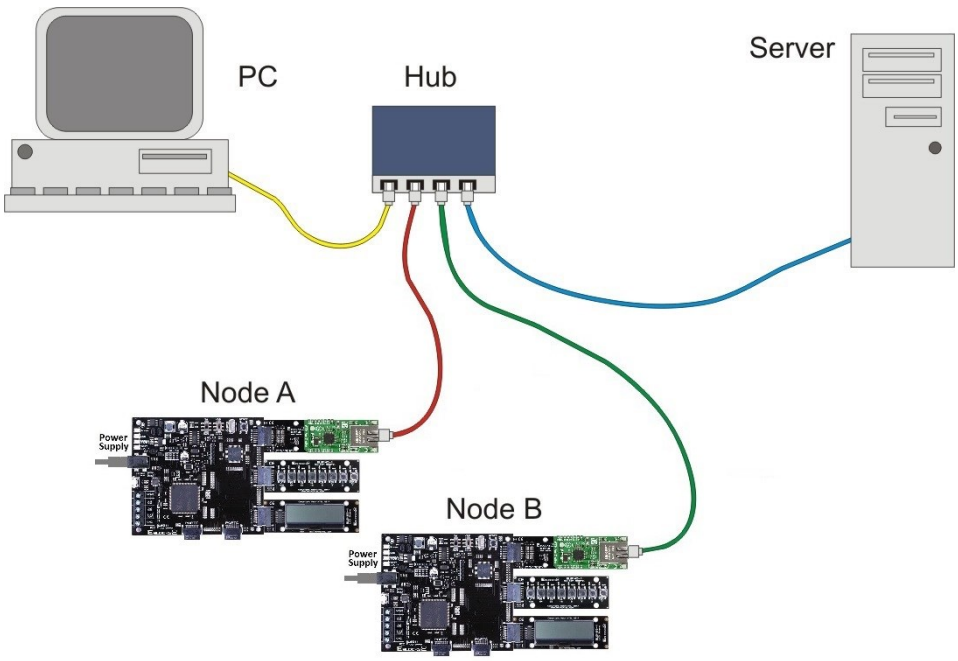
Notes on Advanced Exercise 1: Custom messaging using UDP

Hardware setup

You will need two internet board sets as each program is self-standing. You will also need to enable the two boards to communicate between to each other. The two boards can be connected directly to each other via a cross-over cable.



If you wish to use Network traffic monitoring software you will need either a hub or two connection points onto the network.



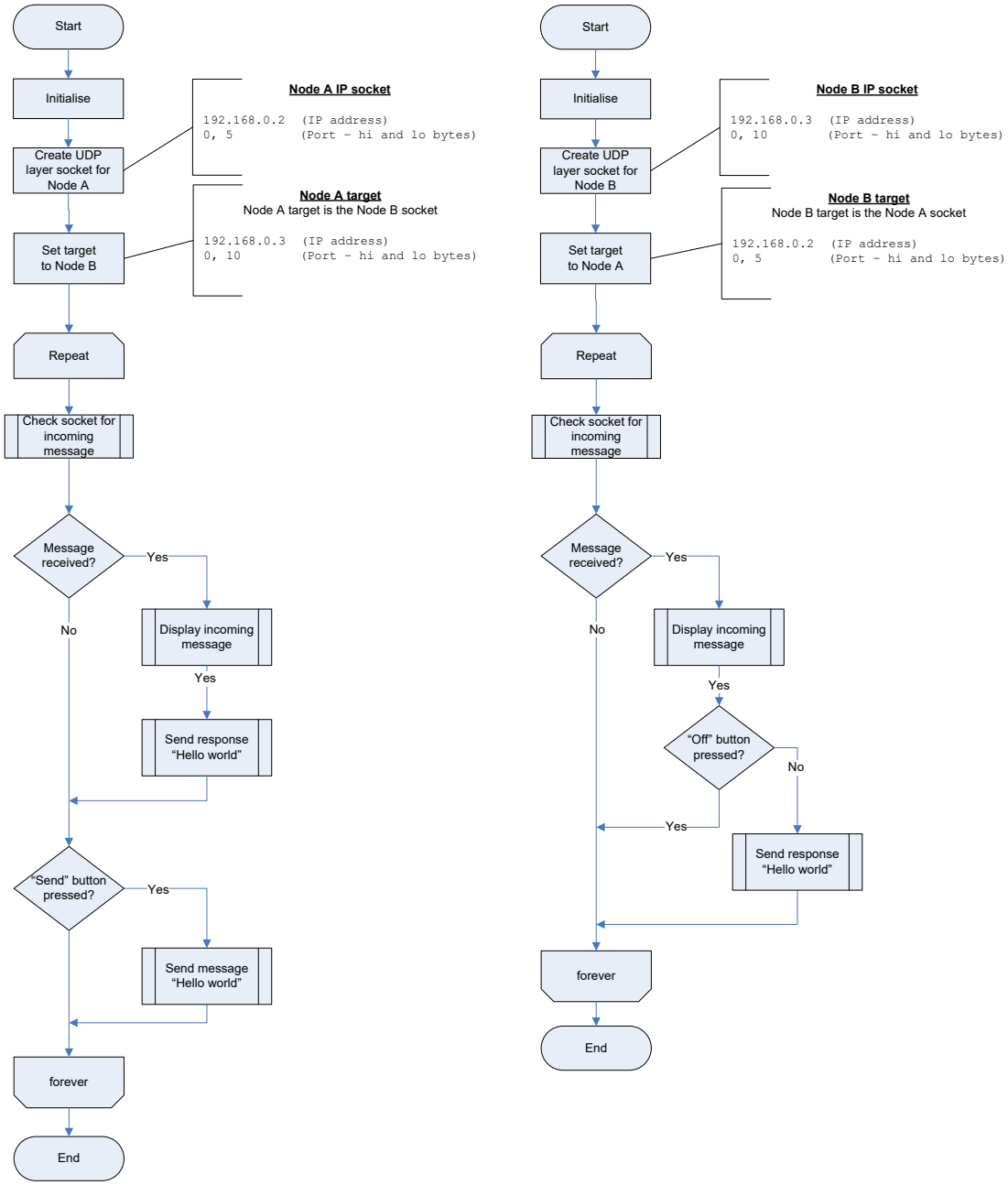
Notes on the programs

You will need two programs, one for Node A and one for Node B. The main difference between the two programs is that a signal can be used to initiate a send on Node A to get the ball rolling, and a signal is used with Node B to prevent a response to halt the process.
On a practical note it would be wise to add a delay before responding to a message to prevent a flood of messages jamming up the network.

Notes on Advanced Exercise 1: Custom messaging using UDP

Program overview

The basic programs are quite simple and are outlined on the flowchart below.



Further work

The messages received were accepted and responded to. What would we need to do to introduce an element of error checking into the messaging system?

- How would we cope with a completely empty message?
- How would we know it was empty?

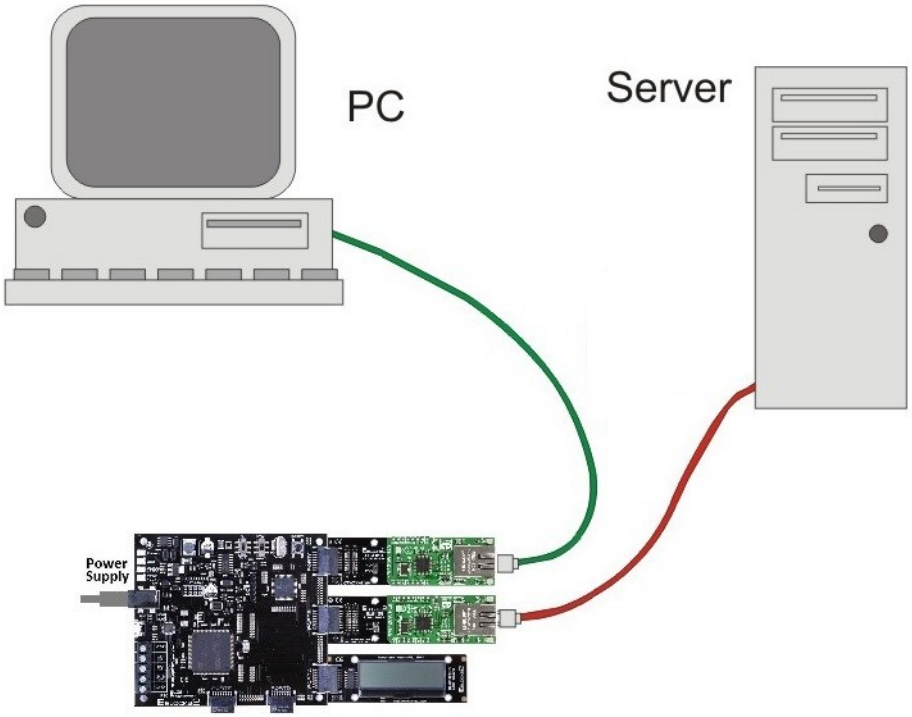
Advanced Exercise 2: Firewall application design

A firewall sits in front of an application or applications and monitors any communication going to that application. The firewall is designed to detect and block any unauthorised access to the application.
Due to the spread of viruses and hacker programs firewalls have become an important part of the TCP/IP armoury against unwanted intrusion.

Our project is to design and implement a firewall application. You will need to create a program that acts as an intermediary between two Ethernet boards, deciding if a message should be allowed to pass through or not.

Hardware considerations

The simplest solution is to connect two internet boards to the same programmer board.



One board connected to the network, and the other connected to the protected PC. You can then create an application that monitors both and decides whether to pass the data on to the other board or not.

Security Criteria

What would you like to block?
What does the end user need?
What can you block?

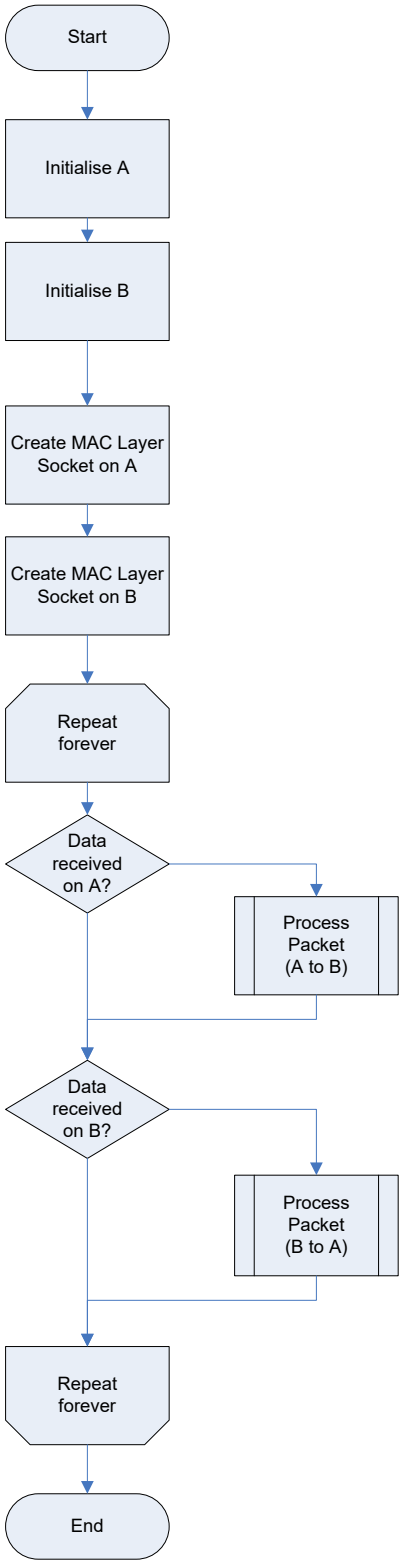
We will leave this up to you to decide what to block – specific IP addresses, ARP requests, Emails etc.
Bear in mind that the microcontroller has limited memory available so it may be best to keep it simple at first, and maybe expand the application later once it is working.

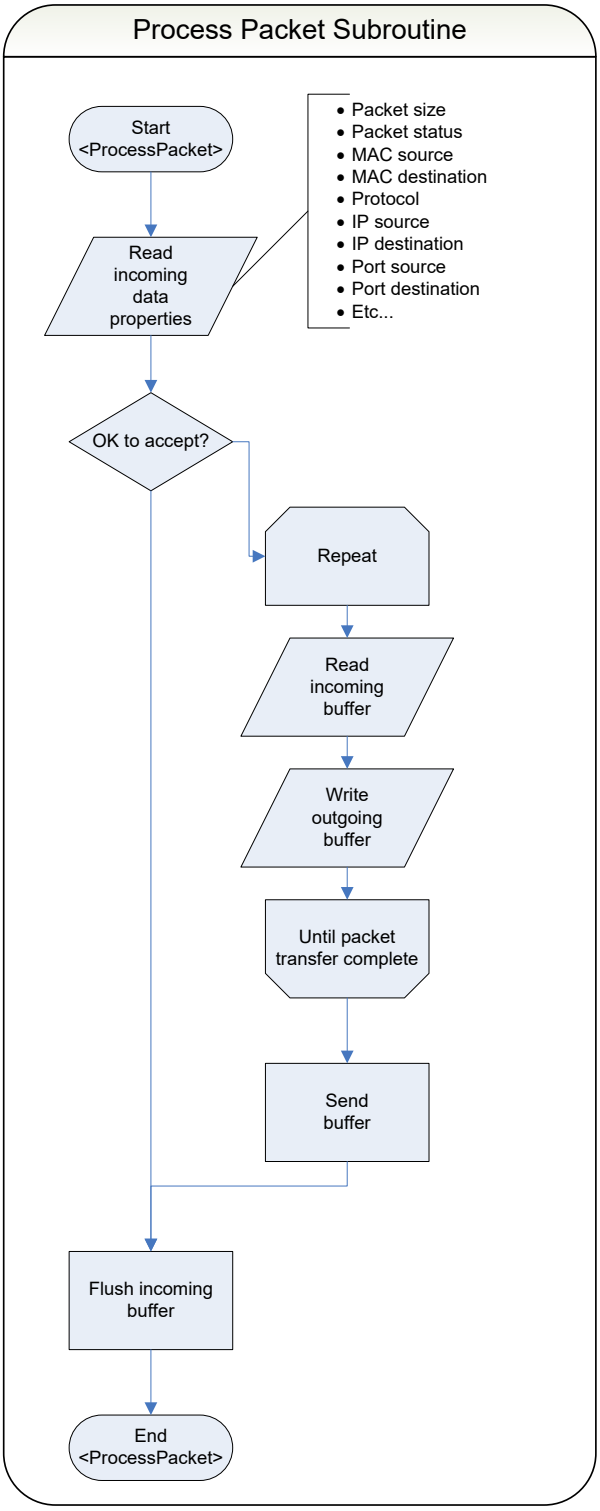
Program design

This too is up to you. By now you will have the experience with TCP/IP needed to be able to send receive and examine the various datagram's and Ethernet frames used in TCP/IP.

To help you get started we have included an example flowchart for a basic firewall system that passes the Ethernet messages between two systems.

Advanced Exercise 2: Firewall application design





There is a lot more to learn about TCP/IP, both about the process of sending and receiving messages, and about the data messages that can be sent.

Fragmentation

There are issues of fragmentation, where messages require breaking into smaller parts, or fragments, and also require reassembly at the other end. This is largely irrelevant for the microcontroller as the relatively small memory capacity will most likely be a more immediate factor. Basic fragmentation issues are also handled automatically by the internet board. However using memory chips, or using the microcontroller as a transfer conduit, may create files sufficiently large for fragmentation to be an issue.

30.2 Header options

Other areas for further work include header options and prioritizing. There are a number of header options and settings that we have only briefly covered in this course. Advanced users may want to look into these to see what they are, and how they can be used to aid communications.

30.3 Error checking

There are various TCP/IP procedures for error checking and error reporting. From simply checking that the Ethernet datagram contains the expected message type values, through to the two-way dialogue error codes of SMTP.

30.4 Message data

On the data side there is much to learn about formatting TCP messages such as HTML and SMTP E-mail. There is a lot of data available on these message formats on the web.

Should you wish to look into other TCP message formats, such as POP3, FTP or TELNET you will need to look for details on how to structure the data, and what request/response dialogues are required.

30.5 Other protocols

There are a number of protocols that we have not looked at here including SLIP, DHCP, and NTP. Details of these protocols can be found in various books, and on the web.

31 Web links and resources

SMTP document
http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol

HTTP document
http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Port assignment numbers
http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml

32.1 Acronyms

ARP – Address Resolution Protocol
DHCP – Dynamic Host Configuration Protocol
DNS – Domain Name Servers
FTP – File Transfer Protocol
HTML – HyperText Markup Language
HTTP – HyperText Transfer Protocol
ICMP – Internet Control Message Protocol
IP – Internet Protocol
MAC – Media Access Control
POP3 – Post Office Protocol 3
SMTP – Simple Mail Transfer Protocol
TCP – Transmission Control Protocol
UDP – User Datagram Protocol
URL – Universal Resource Locator

32.2 Glossary

Datagram – a data packet (data-telegram) contained inside another message.
Datagram's can also be nested with one datagram tucked inside another datagram.

Firewall – a security monitoring system that's helps prevent unauthorized messages getting through to the node. Normally software based but can also include hardware based firewalls.

Hub – A network connection point. Messages received are sent to all other connections on the hub. Hubs are non-intelligent and simply pass incoming messages on with no filtering.

Node – A system on the network with an IP address and a MAC address, be it a PC, a Unix server, or an E-blocks internet solution etc.

Router – A device that can accept TCP/IP messages and pass them on to the next device in the path between the sender and the recipient nodes. Routers are capable of intelligently assessing network problems and blockages and re-routing messages depending on the network conditions.

Switch – Similar to a hub but with intelligent filtering which stops messages being sent to connections that the message is not for. Not recommended for use with Network traffic analyzers as they prevent general network traffic messages from getting through to the analyzing node.