Connecting an M68HC08 Family Microcontroller to an Internet Service Provider (ISP) Using the Point-to-Point Protocol (PPP)

By Rene Trenado
SPS Latin America
Tijuana, Baja California, Mexico

Introduction

This application note is based on an M68HC08 Family microcontroller (MCU) and implements one of the most popular and accepted Internet protocols: the point-to-point protocol (PPP) to exchange UDP/IP (user datagram protocol/Internet protocol) data with other hosts on the Internet. The source code is written entirely in C, showing much of the benefits of the M68HC08 CPU features to support this high-level language (HLL) programming and enables it to be easily ported to other MCUs. The program code occupies less than 6K of memory.

Today the Internet is an integral part of our daily lives. Millions of people all over the world are familiar with the mediums to obtain and manage information over the World Wide Web. Those same people feed the exponential growth of the Internet, enabling new consumer products in the electronic industry.

The Internet is entering a new era where it impacts our lives at work and at home, regardless of distance. It is clear that this tendency will effect the next evolution of the information super highway.
The benefits are endless. Imagine the ability to add new product features remotely, perform device management and remote diagnostics, integrate an interactive and intuitive browser interface to the electronic device, and using that interface all over the world. As these consumer requirements evolve, the integration of the Internet-enabling technology into new and existing electronic devices will become more of a reality.

Unfortunately, for most electronic devices, implementing the technology to achieve this networking connectivity based on open Internet standards isn’t easy. For instance, most household appliances are based on very low-cost 8-bit microcontroller technology, and chances are that the host MCU includes neither a network port nor the hardware resources to support TCP/IP (transmission control protocol/Internet protocol) and other Internet protocols without disrupting their primary function.

Implementing an entire Internet communications stack requires significant memory and processing resources from the microcontroller-based system. In most cases, adding those resources to the system surpasses the cost and viability of the main reason why the system was conceived.

However, different techniques to Web-enable devices have come to life recently: from implementations of limited TCP/IP functionality in resource constrained systems, to single-chip stacks, to device object servers. Each method has its own advantages and disadvantages.

The intention of this application note is to show that a small, resource-constrained microcontroller can be connected to the Internet when the appropriate resources and well-suited CPU (central processor unit) architecture, such as the one of the M68HC08 Family of MCUs, are put in place.
The Internet can be seen as a network of several internetworks (or networks of networks) operating over a mechanism used to connect them together. This mechanism relies on the Internet protocol, often referred as the IP protocol.

To understand how this Internet platform operates, first consider how a local area network or LAN works. A LAN is basically a group of electronic devices (or hosts) in relative physical proximity connected to each other over a shared medium. A host is essentially anything on the network that is capable of receiving and transmitting information packets on the network. Regardless of the technology used for networking, all hosts share a common physical medium. On top of this medium, a commonly accepted protocol allows all hosts in the LAN to send and receive information to each other.
A LAN works well in practice, especially when a relatively small number of hosts conforms to it. The larger the number of hosts connected to the LAN the larger the traffic of data the shared conduit will experience.

Consider this scenario. A company runs a common LAN for its departments. Human resources (HR) is working on the weekly payroll while production is programming the manufacturing plan of the day and engineering is testing the next fancy product the company will launch to market. It does not make sense for HR to experience the high latency on the network caused by the manufacturing process or even engineering testing using the same channel of communication. At the end of the day, nobody will get paid because HR did not finish the payroll processing.

One solution to this scenario would be to split the company LAN in different sections, one for each department. Then instead of having just one network, the company would have three, and data traffic would be reduced to a specific department only. Although the problem is now solved, all three LANs still need to be connected together so they can share specific information. To interconnect two or more networks, we need a computer or host that is attached to both networks and that can forward packets from one network to the other; such a machine is called a router. Figure 2 shows how a router interconnects two networks.
A router listens to data traffic in network A and network B at the same time. It will detect any transmissions intended for one network to the other and will forward such data over the appropriate network. According to the figure, it is assumed that the router is a machine connected to both networks at the same time. This approach works well in an office environment where hosts are physically close to each other. However, when the physical distance becomes an issue, this scheme changes a bit to define a wide area network or WAN.

In a WAN, connections are typically point-to-point. This means that only a single computer is connected to another in a remote location. In this scheme, a conduit is shared between two hosts rather than being shared by many computers. Consider the diagram in Figure 3.
The Internet is not very different from this scenario. As a matter of fact, the Internet is a collection of LANs or WANs connected to each other by routers operating on a worldwide basis. It is mainly composed of two different kinds of machines: hosts and routers running standard protocols.

According to Figure 2, assume the fact that network B can be connected to a network C and in turn be connected to another network in Asia called network D and so on. Such networks interconnected by routers form an internet. When different internets are connected together on a worldwide basis, they form the Internet.

What makes it possible for different computer systems (and in turn different network platforms) to operate together is a complete suite of standards and networking protocols commonly referred as Internet protocols.

Like most networking software, Internet protocols are modeled in layers. A layered model of a software is often referred to as a stack. The Internet protocols can be modeled in five layers as shown in Figure 4.
In the Internet protocol stack, every layer adds a header and/or a trailer to data moving down the stack. For instance, if an application using the HyperText Transfer Protocol (HTTP), such as a Web browser, wants to send an HTTP command to a remote host on the Internet, the TCP layer will add a header intended for its peer TCP at the remote location. The TCP will move the HTTP command down the stack to the IP layer. In turn, this layer will add another header to the TCP encapsulated HTTP packet with information intended to the peer IP layer and so on, as shown in Figure 5.

Figure 4. OSI Reference Model and Internet Networking Stack

Figure 5. Header and Trailer Data Added to an HTTP Message Traveling Down the TCP/IP Stack
Of all the internet family of protocols, the most fundamental is the Internet protocol (IP). Being the best place to start in the quest of understanding the Internet, a brief description of the Internet protocol is included in the next section.

Internet Protocol (IP)

The Internet protocol (IP) is the protocol that makes possible the transmission of blocks of data, called datagrams, from one host to another over the Internet.

The primary functions of the IP are:

1. Finding a route for each datagram and getting it to its destination in an internetwork
2. Fragmenting and reassembling of IP packets
3. Removing old IP packets from the network

The IP protocol defines datagrams or blocks of data plus a header added that conforms the fundamental units of internet communication. The header contains the numerical address of both the source and destination devices connected to the Internet. These types of addresses are often referred to as IP addresses. IP addresses uniquely identify each host on the Internet and are used by routers to direct the datagrams to their destinations. Often, routers do not care about the payload inside the datagram, since their job is to route the datagram to its destination as fast as possible.

Routers are machines that are primarily concerned with the Internet protocol. From the network standpoint, a router is just another host; from the user standpoint, routers are invisible. The user and the upper layer of the stack only see one large internetwork. These are the benefits of the IP protocol.

Fragmentation is another task performed by the IP. Fragmentation is needed when a packet is too large to fit the network interface below the IP layer. If a large datagram arrives at the IP layer, IP divides the datagram into smaller fragments before sending them. When a datagram fragment arrives, IP must reassemble the entire packet before passing it to the next upper layer.
A complete IP implementation should include features to support fragmentation and reassembly. Implementation of such features requires more CPU bandwidth and more memory resources in RAM and ROM, not to mention the complexity it adds to the software implementation. For this reason, this application note does not implement fragmentation or reassembly. If for any reason the remote computer sends a fragmented packet to the M68HC08, the PPP implementation will reject it and ignore it.

The IP protocol implements a mechanism to remove old datagrams from the network. On each header of an IP packet, an 8-bit long time-to-live field indicates the maximum number of routers that this packet must travel on to reach its destination before it is discarded. This is due to the fact that unroutable packets could be bouncing all over the Internet, forever eating valuable bandwidth.

The best way to get a better understanding of the IP protocol is to take a look at the format of an IP packet. See Figure 6.

![Figure 6. Internet Protocol Datagram Layout](attachment:image.png)
A brief description of each of the fields found in an IP packet is given in Table 1.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Indicates the format of the Internet header. Two values are valid for this field: Four (current IP standard) and six for the future IPv6.</td>
</tr>
<tr>
<td>IHL (IP header length)</td>
<td>The length of the Internet header measured in 32-bit words, usually 5.</td>
</tr>
<tr>
<td>Type of service</td>
<td>Specify reliability, precedence, delay and throughput parameters.</td>
</tr>
<tr>
<td>Total Length</td>
<td>Total length of the datagram (header and data) measured in bytes.</td>
</tr>
<tr>
<td>Identification</td>
<td>An ID assigned by the sender to aid in assembling fragmented datagrams.</td>
</tr>
<tr>
<td>Flags (3 bits)</td>
<td>One bit indicates fragmentation; another is the &quot;Don't fragment&quot; bit, specifying whether the fragment may be fragmented. The last bit is reserved.</td>
</tr>
<tr>
<td>Fragment offset</td>
<td>Indicates a fragment portion.</td>
</tr>
<tr>
<td>Time to live</td>
<td>Indicates the maximum time the datagram is allowed to remain in the Internet.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Indicates the next layer protocol which is to receive the payload of the datagram.</td>
</tr>
<tr>
<td>Header checksum</td>
<td>A checksum of the header only.</td>
</tr>
<tr>
<td>Source address</td>
<td>The sender IP address.</td>
</tr>
<tr>
<td>Destination address</td>
<td>The destination IP address.</td>
</tr>
<tr>
<td>Options</td>
<td>The option field is variable in length and is optional. There may be zero or more options. This application note does not support options.</td>
</tr>
<tr>
<td>Padding</td>
<td>If options are present, padding ensures the IP header ends on a 32-bit boundary.</td>
</tr>
<tr>
<td>Data</td>
<td>Payload of the datagram.</td>
</tr>
</tbody>
</table>

An example of an IP datagram is shown in Figure 7. Notice how the IP packet carries ICMP data of a ping request from 192.168.55.2 to 192.168.55.1.

```
45 00 00 1C 00 F4 00 00 80 01
A4 99 C0 A8 37 02 C0 A8 37 01
08 00 F6 51 01 00 00 AE
```

Figure 7. Example of an IP Datagram with ICMP Payload
The IP implementation used by this application note does not use most of the fields in the IP header. For every incoming datagram, the implementation checks the version and header length to avoid IP headers longer than 20 bytes. IP checksums are not checked since a more robust frame check sequence (FCS) over the entire IP datagram are computed at the PPP level.

The IP protocol does not provide a mechanism to detect if a datagram has successfully reached its destination. It does not care if a packet sent is lost, duplicated, or corrupted. It relies on higher level protocols to ensure a reliable transmission. That's precisely the job of the next layer up the stack, the transport layer, which in the case of TCP/IP includes UDP and TCP.

UDP Protocol

UDP stands for user datagram protocol, a standard protocol with assigned number 17 as described by RFC 790 (request for comments). Its status is recommended, but almost every TCP/IP stack implementation that is in use in commercial products includes UDP. Think of UDP as an application interface to IP since applications never use IP directly. The UDP layer can be regarded as extremely thin with eight bytes of header, and, consequently, it has low overhead. But it requires the application layer to take full responsibility for error recovery, packet retransmissions, and so on.

Figure 8. UDP as an Application Interface to IP
UDP provides no means for flow control or error recovery like his peer TCP, thus making it an unreliable protocol. Unreliable means that UDP does not use acknowledgments when a datagram arrives at its destination, it does not order incoming messages arriving out of sequence, and it does not provide feedback to control the rate at which incoming information flows between hosts. Thus UDP messages can be lost, duplicated, or arrive out of order. This means that it is up to the application using UDP to make the transfer reliable.

UDP is mainly used for transmitting live audio and video, for which some lost or out of sequence data is not a big issue and the advantage of having a transport protocol with low overhead is evident.

The UDP header reflects the simplicity of the protocol in Figure 9.

<table>
<thead>
<tr>
<th>16-BIT SOURCE PORT</th>
<th>16-BIT DESTINATION PORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-BIT LENGTH (UDP HEADER + DATA)</td>
<td>16-BIT CHECKSUM (UDP HEADER + DATA)</td>
</tr>
<tr>
<td>PAYLOAD DATA</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. UDP Packet Format

UDP simply serves as a multiplexer/demultiplexer for sending and receiving datagrams using ports to direct them to different services at both ends of the Internet conversation. Notice how the UDP format specifies two ports; one is the source port and the other is the destination port.

A port is a 16-bit number, used by the host-to-host protocol to identify to which higher level protocol or application program it must deliver incoming messages. In a TCP connection, for instance, a well-known port is port 80. HTTP servers expect an incoming request from clients through this port.

Standard applications using UDP include Trivial File Transfer Protocol (TFTP), Domain Name System (DNS) name server, Remote Procedure Call (RPC) used by the Network File System (NFS), Simple Network Management Protocol (SNMP), and Lightweight Directory Access Protocol (LDAP).
A UDP/IP packet containing a "Hello World!" message is shown in Figure 10. The packet is being sent from a host with IP address 192.168.55.2 to 192.168.55.1. The source port is 1020 while the destination port is 11222.

```
45 00 00 28 00 F0 00 00 80 11 97 34 C0 A8 37 02
C0 A8 37 01 03 FC 2B D6 00 14 DB 63 48 65 6C 6F 20 57 6F 72 6C 64 21
6F 20 57 6F 72 6C 64 21
```

**Figure 10. UDP Packet Carrying "Hello World!" Message**

### Internet Control Message Protocol (ICMP)

The Internet control message protocol or ICMP is used to provide feedback about problems in the communication environment used by the IP as stated in RFC 792 which describes this protocol. ICMP provides mechanisms to tell whether the part of the Internet we are sending datagrams to or want to access is active.

ICMP is always carried by the IP or encapsulated within the IP data packets. ICMP datagrams will always have a protocol number of 1 inside the IP header, indicating ICMP. The IP Data field will contain the actual ICMP message in the layout shown next in Figure 11.

**Figure 11. ICMP Message Layout**
The ICMP message layout is very simple. Implementations of this protocol should check the type and code fields to determine the nature of the message. For instance, a type field set to 8 requires an echo reply from the destination IP. The originator of this ICMP message can then determine if the host is reachable or not. This is perhaps the most popular ICMP application used today and is called ping (described next). After the Code field, the checksum follows and is calculated over the entire ICMP packet without taking the IP header into account.

This application note implements ICMP support to send and receive ping messages. The format of a ping message (officially called echo request) is shown in Figure 12.

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
<th>Checksum</th>
<th>Identification</th>
<th>Sequence</th>
<th>Optional Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>-</td>
<td></td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Type**
- 8 - ECHO REQUEST
- 0 - ECHO REPLY

**Code**
- Always 0 for ECHO messages

**Identification and Sequence**
- Two 16-bit fields to aid in matching echoes and replies

**Data**
- This data is optional for the originator; however, for an upcoming ping request, the data must be returned in the reply message.

Once the sender sets the type field to 8 (echo request) and the code to 0, it must initialize the identifier and sequence number prior to a ping execution. Those fields are used when multiple echo requests are sent. If desired, the ping originator can add optional data to the ICMP packet. The maximum amount of data should be no more than 64 Kbytes long. Since this amount also applies to incoming requests, this application note silently discards such big packets.
Dialing an Internet Service Provider (ISP)

Once connected to the Internet, a system can send packets of information to other hosts who are on-line regardless of the physical location of the destination host. That’s the main job of the IP protocol and the internetwork infrastructure of routers and gateways that form the Internet. Each time a system wants to be connected to the Internet it must have the physical interface to do so. One of the most popular ways to establish an Internet connection is by using a modem attached to a phone line with the help of an Internet service provider or ISP. An ISP is a company that provides access to the Internet and other related services such as Web site building and hosting. An ISP has the equipment and the telecommunication line access required to have an access point to the Internet with a unique IP address.

Figure 13. Modem Connection to an Internet Service Provider (ISP)
A host first dials to the phone number of the ISP. After the user is logged in and the password authentication process is done, the ISP assigns a unique IP address to the dialing host. This unique IP address is often referred to as point of presence or POP. Since the dialing host now owns a POP, it is part of the ISP network by means of the ISP router. At that time, the dialing host is now connected to the Internet.

When the host sends an IP packet to the Internet, the host does not know where the destination device is; it simply knows its IP address. When the IP packet reaches the ISP router, the router will try to resolve the IP address on the ISP local network. This step will be executed by each router the IP packet travels on.

### Point-to-Point Protocol (PPP)

The point-to-point protocol or PPP is the predominant connection type used today for serial links. PPP is a complete suite of standard protocols widely adopted by the industry that allows two hosts to interoperate in a multi-vendor network using a serial link such as RS232.

Accordingly to RFC 1662, PPP uses a HDLC-like framing providing address and control fields; for PPP these fields are constants 0xFF and 0x03. For RS232 interfaces, PPP can be seen as a byte-oriented asynchronous link with one stop bit, no parity, and with no special requirements for the transmission rate.

The only absolute requirement imposed by PPP is the provision of a full-duplex circuit not requiring the use of control signals such as RTS or CTS. Because signaling is not required, the physical layer can be decoupled from the data link layer hiding much of the details of the physical transport.
The format of a PPP packet is shown in Figure 14.

<table>
<thead>
<tr>
<th>Start Flag</th>
<th>Address</th>
<th>Control</th>
<th>Protocol (2 Bytes)</th>
<th>Code (1 Byte)</th>
<th>ID (1 Byte)</th>
<th>Length (2 Bytes)</th>
<th>Payload (Variable)</th>
<th>Checksum (2 Bytes)</th>
<th>End Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7F</td>
<td>0xFF</td>
<td>0x03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x7F</td>
</tr>
</tbody>
</table>

**Figure 14. PPP Packet Format and Protocol Identifiers**

### PPP Framing

Every frame starts and ends with the 0x7F flag. Since this is a special flag, no other instances should be placed inside the packet. To avoid confusion with the link status, this character and other control characters of the ASCII set inside the frame must be escaped. The control escape sequence is defined as 0x7D followed by the result of an XOR operation of the control character with 0x20. This also applies to the 0x7D escape indicator. The escape sequence must be applied to all bytes in a PPP frame but the start and stop indicators. After the start flag, two HDLC constants follow: 0xFF and 0x03. The protocol field is always two bytes long, indicating what type of protocol is contained in the payload and how it should be treated. For practical purposes, this application note will treat the code, ID, and length fields as separate fields from the payload, but, officially, they are part of it.

The code is the type of negotiation packet for LCP, PAP, IPCP, and CHAP frames. For IP datagrams it is usually 0x45 (when the header does not include options which is true most of the time). The ID should be unique for each frame to be negotiated and responses should use that same ID to tie them up together. An exception to this rule is when a PPP frame encapsulates an IP datagram. In such a case and for practical purposes, the ID usually will be the type of service. The payload is variable and depends on the negotiation options of a request or a response. In the case of a IP datagram, the size is compatible with the size field of the PPP frame.
The payload contains the negotiation options or the rest of the IP packet. Finally, a 2-byte checksum or frame check sequence (FCS) which is computed over the entire unescaped packet with the help of a lookup table defined in RFC 1662.

In a PPP session, both peers have no distinction of who is the server and who is the client. Both end-points can carry up a negotiation equally. However, for practical purposes, this application note defines a PPP server as the end-point located and handled by the ISP and a PPP client as the end-point that initiates the connection. Another way to define a PPP server is the end of the link that requires password authentication, that is the authenticator.

Usually, PPP sessions are started by a client dialing up an ISP. To start a session, the PPP client must establish, maintain, and terminate a physical connection with the ISP.

The overall process is illustrated in **Figure 15**.

---

**Figure 15. Creating a PPP Link with an ISP**
A more in-depth look of the dial up sequence for PPP will show that the sequence involves the following three steps:

1. LCP negotiations — Establish and configure link and framing parameters such as maximum frame size

2. Negotiate authentication protocols — The authentication protocols defined for PPP are the challenge authentication protocol (CHAP) and the password authentication protocol (PAP). The security level of these protocols ranges from encrypted authentication (CHAP) to clear text password authentication (PAP). This application note only supports PAP.

3. Negotiate network control protocols (NCP) — NCPs are used to establish and configure different network protocol parameters, such as IP. This includes negotiating protocol header compression or IP address assignation.

Before a link is considered ready for use by network-layer protocols, a specific sequence of events must happen. The LCP provides a method of establishing, configuring, maintaining, and terminating the connection.

LCP goes through four phases:

1. Link establishment and configuration negotiation (LCP phase) — In this phase, link control packets are exchanged and link configuration options are negotiated. Once options are agreed upon, the link is open, but not necessarily ready for network-layer protocols to be started.

2. Authentication (PAP or CHAP phase) — This phase is optional. Each end of the link authenticate itself with the remote end using authentication methods agreed to during phase 1.

3. Network-layer protocol configuration negotiation (IPCP phase) — Once LCP has finished the previous phase, network-layer protocols may be separately configured by the appropriate NCP.

4. Link termination — LCP may terminate the link at any time. This usually will be done at the request of a human user, but may happen because of a physical event.
LCP Negotiations

The link control protocol (LCP) is used to establish the connection through an exchange of configure packets. LCP negotiations are the first to take place during the PPP session.

The mechanism for PPP negotiations relies on the packet codes described in Table 2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Packet Type</th>
<th>Defined In</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vendor specific</td>
<td>RFC2153</td>
<td>Proprietary vendor extensions</td>
</tr>
<tr>
<td>1</td>
<td>Configure-request</td>
<td>RFC1661</td>
<td>Configuration options the sender desires to negotiate</td>
</tr>
<tr>
<td>2</td>
<td>Configure-ack</td>
<td>RFC1661</td>
<td>Configuration options the sender is acknowledging</td>
</tr>
<tr>
<td>3</td>
<td>Configure-nak</td>
<td>RFC1661</td>
<td>Unacceptable configuration options from the configure-request packet; acceptable values are included</td>
</tr>
<tr>
<td>4</td>
<td>Configure-reject</td>
<td>RFC1661</td>
<td>Configuration options are not recognizable or are not acceptable for negotiations</td>
</tr>
<tr>
<td>5</td>
<td>Terminate-request</td>
<td>RFC1661</td>
<td>Terminate this link</td>
</tr>
<tr>
<td>6</td>
<td>Terminate-ack</td>
<td>RFC1661</td>
<td>Terminate acknowledge</td>
</tr>
<tr>
<td>7</td>
<td>Code-reject</td>
<td>RFC1661</td>
<td>Reception of an LCP packet with an unknown code</td>
</tr>
<tr>
<td>8</td>
<td>Protocol-reject</td>
<td>RFC1661</td>
<td>Reception of a PPP packet with an unknown protocol field</td>
</tr>
<tr>
<td>9</td>
<td>Echo-request</td>
<td>RFC1661</td>
<td>Initiation of a loopback mechanism</td>
</tr>
<tr>
<td>10</td>
<td>Echo-reply</td>
<td>RFC1661</td>
<td>Response to an echo-request</td>
</tr>
<tr>
<td>11</td>
<td>Discard-request</td>
<td>RFC1661</td>
<td>Discard this packet for testing and debugging purposes</td>
</tr>
</tbody>
</table>

Figure 16 shows an example of the first LCP packet transmitted by an ISP.

<table>
<thead>
<tr>
<th>LCP Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000: 7F FF 03 21 01 71 00 2B 01 04 06 40 05 06 3A 5D B4 02 06 00</td>
</tr>
<tr>
<td>0016: 00 00 00 11 04 06 40 17 04 00 64 00 02 03 04 C0 23 13 09 03 08 00</td>
</tr>
<tr>
<td>002C: 03 0A 2C 2C 95 7F 7F</td>
</tr>
</tbody>
</table>

NOTE: The figure shows a packet without applying the escape sequence.

Figure 16. First LCP Packet Transmitted by an ISP
A description of the LCP data is given in Table 3.

### Table 3. LCP Data Description

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Hexadecimal Value(s)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framing</td>
<td>7F</td>
<td>Start of packet</td>
</tr>
<tr>
<td></td>
<td>FF 03</td>
<td>Framing</td>
</tr>
<tr>
<td>Protocol</td>
<td>C0 21</td>
<td>LCP protocol</td>
</tr>
<tr>
<td>Negotiation code</td>
<td>01</td>
<td>REQ - Request options</td>
</tr>
<tr>
<td>ID</td>
<td>71</td>
<td>ID for this packet</td>
</tr>
<tr>
<td>Size of packet</td>
<td>00 2B</td>
<td>Size of payload starting from negotiation code</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>Option 1, Maximum-Receive-Unit</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Size of option 1, 4 Bytes</td>
</tr>
<tr>
<td></td>
<td>06 40</td>
<td>Option value requested, MRU = 1600</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>Option 5, Magic number</td>
</tr>
<tr>
<td></td>
<td>06</td>
<td>Size of option 5, 6 Bytes</td>
</tr>
<tr>
<td></td>
<td>3A 5D 8B B4</td>
<td>Value of magic number</td>
</tr>
<tr>
<td>Options</td>
<td>02</td>
<td>Option 2, Async-Control-Character-Map</td>
</tr>
<tr>
<td></td>
<td>06</td>
<td>Size of option 2</td>
</tr>
<tr>
<td></td>
<td>00 00 00 00</td>
<td>Escape no characters</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Option 17, Multilink-MRRU</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Size of option 11</td>
</tr>
<tr>
<td></td>
<td>06 40</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Option 23, Link Discriminator for BACP</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Size of option 17</td>
</tr>
<tr>
<td></td>
<td>00 64</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>Option 0, Vendor Specific</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>Size of option 0</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>Option 3, Authentication-Protocol</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Size of option 3</td>
</tr>
<tr>
<td></td>
<td>C0 23</td>
<td>Value set to PAP</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>Option 19, Multilink-Endpoint-Discriminator</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>Size of option 13</td>
</tr>
<tr>
<td></td>
<td>03 08 00 03 0A 2C 2C</td>
<td>Value of option 13</td>
</tr>
<tr>
<td>Checksum</td>
<td>95 7F</td>
<td>Checksum of this packet</td>
</tr>
<tr>
<td>Framing</td>
<td>7F</td>
<td>End of packet</td>
</tr>
</tbody>
</table>
The most common LCP negotiations happening during initial connection are maximum-receive-unit, protocol-field-compression, magic-number, authentication-protocol and async-control-character-map, all described in RFC1661 and RFC1662. This application note tries to force negotiations to go our way. It first tries to use the default settings provided by the ISP and goes from there. However, different implementations should modify the state machine inside HandleLCPOptions () routine to handle LCP options differently.

Password Authentication Protocol (PAP)

The password authentication protocol is defined in RFC 1334. PAP is intended primarily for use by hosts and routers that connect to a PPP network server commonly via dial-up lines, but it might be applied to dedicated point-to-point links as well. The server can use the identification of the connecting host or router in the selection of options for network layer negotiations. The authenticate-request packet format is shown in Figure 17.

Figure 17. PAP Packet Layout and Sample — User ID = "", Password = "rene"
Internet Protocol Control Protocol

After the PPP host has been authenticated, the next phase is the network-layer protocol. The Internet protocol control protocol (IPCP) is used to configure the Internet protocol environment to be used in a PPP link. Options such as IP address, IP compression, primary DNS server, etc., are negotiated using IPCP.

The format of an IPCP frame is similar to that of LCP: a 1 byte negotiation code followed by ID, length, and options. Once the IP protocol has been configured, datagrams from each network can be sent in both direction over the link. Further details of IPCP are covered in RFC 1332.

PPP Negotiations

All LCP negotiations are performed in a state machine implemented inside the PPP.C module. When the first LCP packet arrives from the ISP, the state machine responds with a NAK packet with the same options the ISP sent us before. This will force the ISP to reply with a request for the authentication protocol to be used. The negotiation flow is shown in Figure 18.
A hexadecimal dump of the LCP, PAP, and IPCP negotiation sequence is shown in Figure 19. This dump is a recorded PPP session between a real ISP and the M68HC08-based application. First, LCP negotiations are shown in Figure 19.
Figure 19. LCP negotiations with an ISP

After ISP agrees to use PAP during the LCP negotiation phase, the M68HC08 must send the user ID and password to log into the ISP network. This process is illustrated in Figure 20.

Figure 20. PAP Sequence

For More Information On This Product,  
Go to: www.freescale.com
Now that the M68HC08 has been authenticated, the next step is to configure the network protocols to be used inside the ISP network. Since we are negotiating with an Internet service provider, IPCP will be used for sure to negotiate IP. IPCP negotiations follow PAP authentication as illustrated in Figure 21.

---

(11) ISP send REQ for IPCP negotiations
FF 03 80 21 01 01 00 10 02 06 00 2D 0F 01 03 06 C0 A8 37 01 C2 81

(12) HC08 reply with a NAK for all options but option 3 - IP address
FF 03 80 21 04 01 00 0A 02 06 00 2D 0F 01 6C 65

(13) ISP sends a reply because of the previous NAK sent, this time with IP address only
FF 03 80 21 01 02 00 0A 03 06 C8 26 16 02 A4 17

(14) HC08 now as an IP address assigned by the ISP
FF 03 80 21 02 02 00 0A 03 06 C8 26 16 02 A4 17

(15) HC08 REQ an IP address to complete three way hand shake
FF 03 80 21 01 03 00 0A 03 06 00 00 00 CD 63

(16) ISP reply with a NAK containing the pre-assigned IP address
FF 03 80 21 03 03 00 0A 03 06 C8 38 6F 42 41 F2

(17) HC08 is now On-Line with IP Address: 200.56.111.66
FF 03 80 21 02 03 00 0A 03 06 C8 38 6F 42 66 DE

---

**Figure 21. IPCP Negotiations between an ISP and the MC68HC08GP32**

---

**Serial Line Internet Protocol (SLIP)**

This application note also implements the serial line Internet protocol (SLIP) to communicate directly with hosts acting as routers or gateways. The SLIP specifies a way to encapsulate raw IP datagrams over a regular serial communication line. It is a de facto standard not an Internet standard. However, given its popularity, SLIP is described in RFC 1055. Because of its simplicity, SLIP is very easy to implement in comparison with other point-to-point protocols. However, since SLIP specifies only a way to frame an IP packet, it is far less reliable than PPP since it does not provide mechanisms for IP addressing or support for multiple protocols running on top of it. Addressing is a big issue since both ends of the point-to-point link need to know each other’s IP addresses for routing purposes.
SLIP defines the following escape codes to signal frame boundaries: END (hexadecimal 0xC0) and ESC (hexadecimal 0xDB).

To send an IP datagram packet, the SLIP host commonly sends an END character, signaling the start of a frame. If any instance of the END code exists within the IP datagram, a 2-byte sequence of ESC and 0xDC are sent instead. After the last byte of the datagram has been sent, an END character is then transmitted as shown in Figure 22.

Since the ESC code is also a special character, a SLIP implementation should escape this code as well but with this 2-byte sequence: ESC and 0xDD.

![Figure 22. SLIP Frame Layout](image)

One major disadvantage of SLIP is that it requires a dial-up script to negotiate the user ID and authentication with an Internet service provider. Different ISPs would require different scripts, and any changes on the script in the ISP side would require appropriate changes on the client side, thus making it more difficult to implement in a small MCU. Because of the limitations and lack of features, the SLIP protocol is expected to be replaced by the point-to-point protocol (PPP).
UDP/IP Application

This application note shows how a small and inexpensive microcontroller such as the MC68HC908GP32 can be connected to the Internet and still save resources on chip to perform basic operations like remote monitoring and/or control.

The application is very simple: a small system based on the MC68HC908GP32 that monitors an external variable by using the 8-bit analog-to-digital (A/D) builds on chip via a module channel.

In case the A/D reading or some other event is triggered (a pre-fixed A/D threshold has been reached for example), the MC68HC908GP32-based system will send a UDP/IP asynchronous notification to a pre-compiled IP address. This destination IP could be a proxy gateway on the Web, or a custom UDP/IP terminal working as a standalone application, or in the form of a Java applet, or an ActiveX control embedded in a Web page.

Application Framework Block Diagram

The application framework is shown in Figure 23. The MC68HC908GP32 acts as a message initiator. It waits until program-defined conditions are meet. A predefine condition could be a security system signaling that it has been triggered, air conditioner has reached a pre-defined threshold, door bell, etc. The system will first dial an ISP to establish a PPP link (1). The ISP will authenticate the system and will assign a unique IP address. After that, the MC68HC908GP32 will now be ready to send a notification to the Internet via PPP/UDP/IP (2).

Figure 23. Application Framework
Once on the Internet, a message could travel to virtually everywhere in the planet. With little effort, the UDP datagram could be publicized by a program running at the destination host.

**Software Operation**

The software implementation has been divided in a series of C modules. Code reuse/borrowing and expandability are the main intention for such modularity, so M68HC08 programmers can borrow and/or modify the source code to meet specific application needs for other members of the M68HC08 Family of MCUs. Or they can build a set of libraries and/or features to be integrated in future applications in the form of object code to be linked together during the development process.

These modules are defined by this application note:

- **Main C modules**
  - Main.C
  - CommDrv.C
  - ModemDrv.C
  - PPP.C
  - SLIP.C
  - IP.C
  - UDP.C
  - ICMP.C

- **Miscellaneous C module**
  - Delay.C

The software consists of a main routine (the standard C main() function) that is divided in two in-line portions of code. The first portion initializes the communications port and all the other software modules of the system. The second portion is an infinite loop which calls `ModemEntry()` and `PPPEntry()` functions. This is needed to perform modem handshake and PPP negotiations, respectively. (SLIP could be used instead of PPP by calling `SLIPEntry()` from the main loop.)
The first module we need to inspect is the CommDrv.C. This module is responsible for the appropriate operation of the serial communications of the system. It implements a pseudo-standard method of accessing the serial port hardware. To the application, the serial port can be seen as a set of "API like" routines that perform straight and logical operations (OpenComm(), CloseComm(), WriteComm() etc).

The intention of such implementation is to pursue a fixed level of abstraction to the application code. Abstraction can bring us a lot of benefits. For instance, code reuse and code maintainability are, among others some of the strongest justifications of using it. When hardware changes, the abstraction changes in one portion of the code; changes are almost transparent to the application or portions of the source code. For example, changing the baud rate of the serial port or more often changing the address of the registers (and even the registers) in the initialization sequence of the serial port would require a change in the definitions in the header file of the module and/or the source code of the OpenComm() routine. The benefit, if not obvious, will become evident after linking. Different methods for abstracting hardware exists today, but the implementation is well beyond the scope of this application note.

**NOTE:** The MC68HC908GP32 defines the interrupt vector table in the upper section of the FLASH ROM at address 0xFFDC to 0xFFFF as illustrated in Table 4. In that space, we need to store each of the FLASH ROM locations of every interrupt service routines (ISRs) used by the microcontroller.
The CommDrv.C module defines the ISR code for the interrupt generated each time the SCI receives a byte character. However, this ISR is compiled by the compiler to generate the object code that the linker will realize and place it in FLASH ROM. That means that the source code of the ISR is installed at link time (or design time, if you will) not at run time. Since the serial port of the MCU in this specific implementation will be shared between different modules to perform different tasks at run time, a way must be found to share that ISR with different modules. For instance, the MCU must dial to an ISP by using a modem; after the ISP answers, SLIP scripts or PPP negotiations need to be executed. Modem.C and PPP.C must rely on the CommDrv.C ISR.

One way to achieve the flexibility needed is to forward the ISR to a location in RAM that points to the ultimate interrupt service handler: in other words, a pointer to an ISR that turns out to be a pointer to a function. By using this approach, the programmer has total control of the incoming flow of characters through the serial port.

Table 4. MC68HC908GP32 Interrupt Vector Table

<table>
<thead>
<tr>
<th>Vector</th>
<th>Address</th>
<th>Vector Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0xFFDC</td>
<td>Timebase module vector</td>
</tr>
<tr>
<td>16</td>
<td>0xFFDE</td>
<td>Analog-to-digital conversion complete</td>
</tr>
<tr>
<td>15</td>
<td>0xFFE0</td>
<td>Keyboard scan vector</td>
</tr>
<tr>
<td>14</td>
<td>0xFFE2</td>
<td>Serial communications transmit vector</td>
</tr>
<tr>
<td>13</td>
<td>0xFFE4</td>
<td>Serial communications receive vector</td>
</tr>
<tr>
<td>12</td>
<td>0xFFE6</td>
<td>Serial communications error vector</td>
</tr>
<tr>
<td>11</td>
<td>0xFFE8</td>
<td>SPI transmit vector</td>
</tr>
<tr>
<td>10</td>
<td>0xFFEA</td>
<td>SPI receive vector</td>
</tr>
<tr>
<td>9</td>
<td>0xFFEC</td>
<td>Timer interface module 2 over ow vector</td>
</tr>
<tr>
<td>8</td>
<td>0xFFEE</td>
<td>Timer interface module 2 channel 1 vector</td>
</tr>
<tr>
<td>7</td>
<td>0xFFF0</td>
<td>Timer interface module 2 channel 0 vector</td>
</tr>
<tr>
<td>6</td>
<td>0xFFF2</td>
<td>Timer interface module 1 over ow vector</td>
</tr>
<tr>
<td>5</td>
<td>0xFFF4</td>
<td>Timer interface module 1 channel 1 vector</td>
</tr>
<tr>
<td>4</td>
<td>0xFFF6</td>
<td>Timer interface module 1 channel 0 vector</td>
</tr>
<tr>
<td>3</td>
<td>0xFFF8</td>
<td>PLL vector</td>
</tr>
<tr>
<td>2</td>
<td>0xFFF9</td>
<td>IRQ vector</td>
</tr>
<tr>
<td>1</td>
<td>0xFFFCC</td>
<td>Software interrupt vector</td>
</tr>
<tr>
<td>-</td>
<td>0xFFFEE</td>
<td>Reset</td>
</tr>
</tbody>
</table>
Actually, the body of the ISR of the CommDrv.C is simple and is shown Figure 24.

Listing 1

The M68HC08 CPU has very powerful addressing modes in comparison with other 8-bit MCUs’ architectures in the market. The ISR definition in CommDrv uses a powerful indexed addressing mode provided by the M68HC08 CPU. The JSR instruction can jump to a subroutine pointed to by the index register H:X, which allows the program counter to jump to an effective address with 16-bit resolution.

But for every value-added feature, we must pay a price, and, in this case, we lose valuable CPU bandwidth. The minimum assembly code needed to represent the code in Listing 1 is represented in Figure 25.

If we can force the compiler to place *EvtProcedure (register BYTE value) pointer in the zero page section of RAM, we can get similar results from a compiler, but this will depend mainly on the compiler itself and the context of the development environment used at design time.

The *EvtProcedure pointer becomes initialized at design time by this construct.
static void (* EvtProcedure) (register BYTE value) = CommDrvDefaultProc;

CommDrvDefaultProc() is a private function defined in CommDrv which does nothing but initialize *EvtProcedure pointer and is defined as follows.

static void CommDrvDefaultProc (register BYTE value) { (void) value; }

By using the CommEventProc() function, an application can "mutate" the behavior of the SCI ISR, as shown in this application note.

Overview of the Modem Interface

This application note was built around a “Hayes-compatible” external modem. In the past, when a high-speed modem was considered to be a 9600-baud unit, a company called Hayes Microcomputer Products Inc. made a modem that was widely accepted by microcomputer users. The implementation features and the serial commands used by these modems became a de facto standard in the industry. Given its popularity and for compatibility reasons, nowadays most modems are “Hayes-compatible.”

Operation of a Hayes-Compatible Model

A Hayes modem is always in two states:

- Command mode
- On-line state

When in command mode, instructions can be given to it from the serial port. For example, we can instruct the modem to dial a number or to ignore incoming calls by means of simple commands. These commands are diverted to the modem and are never transmitted.
In the on-line state, once a connection has been established with a modem of a remote system (for instance, an ISP), the local modem enters the on-line state and no longer attempts to interpret the data being sent to it. In other words, every data sent while on-line state is transmitted to the remote modem regardless of its nature. If the remote system hangs up or for any other reason the carrier signal is lost while in on-line state, the modem will revert to local command mode.

When the modem receives a command (in command mode), it returns a result code. This code can be in the form of either a text string or a numeric code. A numeric code is more appropriate for embedded systems, but if we want to control the modem by using a terminal and a keyboard, a verbose mode or text messages are more preferable. We can set the type of result code by using a command message.

Table 5 shows the result codes of a Hayes-compatible modem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Verbose Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
<td>Command executed</td>
</tr>
<tr>
<td>1</td>
<td>CONNECT</td>
<td>Connection established</td>
</tr>
<tr>
<td>2</td>
<td>RING</td>
<td>Ring signal detected</td>
</tr>
<tr>
<td>3</td>
<td>NO CARRIER</td>
<td>Carrier signal lost or not detected</td>
</tr>
<tr>
<td>4</td>
<td>ERROR</td>
<td>Invalid command, checksum, error in command line, or command line too long</td>
</tr>
<tr>
<td>5</td>
<td>CONNECT 1200</td>
<td>Connection established at 1200 bps</td>
</tr>
<tr>
<td>6</td>
<td>NO DIALTONE</td>
<td>No dial tone detected</td>
</tr>
<tr>
<td>7</td>
<td>BUSY</td>
<td>Busy signal detected</td>
</tr>
<tr>
<td>8</td>
<td>NO ANSWER</td>
<td>No response when dialing a system</td>
</tr>
<tr>
<td>9</td>
<td>CONNECT 2400</td>
<td>Connection established at 2400 bps</td>
</tr>
</tbody>
</table>
All command messages start with AT, unless otherwise specified. Several commands can be given in one command line. The Hayes command set provides comprehensive messages to configure the modem, dial phone numbers, and answer incoming calls. This application note implements a way to initiate calls only, but making the software answer incoming calls should not be that difficult if the appropriate commands are listened to and issued to the modem.

Although the term “Hayes compatible” is often used in this document, there is no absolute standard defined. Not all Hayes modems work the same way. Always refer to the modem documentation provided by the modem manufacturer.

The software in this application note assumes the configuration and behavior from the modem listed in Table 6.

### Table 6. Default Configuration of Modem Used in This Application Note

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Hayes Command Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character echo in command state disabled</td>
<td>ATE0</td>
</tr>
<tr>
<td>Modem returns result codes</td>
<td>ATQ0</td>
</tr>
<tr>
<td>Display result codes in verbose form</td>
<td>ATV1</td>
</tr>
<tr>
<td>Long space disconnect disabled</td>
<td>ATY0</td>
</tr>
<tr>
<td>Track the presence of data carrier</td>
<td>AT&amp;C1</td>
</tr>
<tr>
<td>Hang up and assume command state when an on-to-off transition of DTR occurs</td>
<td>AT&amp;D2</td>
</tr>
</tbody>
</table>

As far as the M68HC08-based system is concerned, the external Hayes-compatible modem is just a serial device connected to the SCI. From a software standpoint, the modem implementation runs on top of the serial port driver; in other words, it relies on services provided by the CommDrv module. The wire connections made from the modem to the M68HC08 system include signal ground, transmitter, and receiver pins.

The modem provides several standard hardware signals for modem handshaking. Only two have been hardwired to the system, carrier detect (CD) and data terminal ready (DTR), making a total of five pins to drive the modem as shown in Table 7.
Notice that the SCI on chip drives the transmitter and receiver signals "directly" from the modem (after an RS232 to CMOS converter) while two extra GPIO (general-purpose input/output) pins provide the DTR (data terminal ready) and CD (carrier detect) signals for modem handshaking. DTR is required to hang up the phone while in on-line state and return to command mode when an on-to-off transition occurs. A CD signal can be pooled from the application to know if the modem is in command mode (CD = 1) or in the on-line state (CD = 0).

The modem driver runs on top of the serial communications routines and relies on them. Because of this, the modem implementation provides its own service routine for incoming characters through the serial port, thus avoiding problems while decoding modem response messages. Once the modem goes on line, the modem service routine is removed from the SCI ISR. This allows installation of the appropriate handler for the point-to-point link (SLIP or PPP) at run time.

The modem service routine simply enqueues (puts into queue) incoming characters from the serial port. By default the maximum number of characters that can be stored in the modem queue is 32. This queue performs as a FIFO (first in, first out) buffer and most of the modem functions rely on it. A common FIFO like the one used in this application note has two pointers; one is used to add data to the FIFO while the other removes queue data. This operation is described in Figure 26.

### Table 7. DB9 Connector Interface to the MC68HC908GP32

<table>
<thead>
<tr>
<th>DB9 Pin No.</th>
<th>Pin Name</th>
<th>Description</th>
<th>M68HC08 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CD</td>
<td>Carrier detect</td>
<td>PORTD 1</td>
</tr>
<tr>
<td>2</td>
<td>RxD</td>
<td>Receiver data</td>
<td>SCI receiver</td>
</tr>
<tr>
<td>3</td>
<td>TxD</td>
<td>Transmitter data</td>
<td>SCI transmitter</td>
</tr>
<tr>
<td>4</td>
<td>DTR</td>
<td>Data terminal ready</td>
<td>PORTD 0</td>
</tr>
<tr>
<td>5</td>
<td>GND</td>
<td>Signal ground</td>
<td>System ground</td>
</tr>
</tbody>
</table>
Figure 26. Implementation of a FIFO for the Modem Interface

**Figure 26** shows the two internal pointers that make up a FIFO. At initialization time, both pointers are equal to zero, thus indicating that the FIFO is empty. Once a character is received from the SCI, it is stored at the location pointed to by *emptySlot* before it becomes incremented by one. The figure also shows an ATE0 reply from the modem stored in the FIFO following the process just described. Notice how *emptySlot* now points to the next free location of the FIFO. The *dataSlot* pointer has a similar behavior. To read a character from the FIFO, the application calls the `ModemGetch()` function to retrieve the letter A pointed to by *dataSlot*, then it is incremented by one. At this point, *dataSlot* now points to the letter T. This process is repeated for every character added to the FIFO by the modem input routine.

The code to enqueue character in the FIFO is simple and is illustrated in the next piece of code in **Figure 27**.

```c
#define MODEM_BUFFER_SIZE 32     // Default size of modem buffer

volatile BYTE mDataSlot = 0;     // Points to the next available character
volatile BYTE mEmptySlot = 0; // Points to next available slot of the FIFO

static BYTE *ModemBuffer;                // Pointer to Modem buffer

void ProcModemReceive (BYTE c) {
    ModemBuffer [mEmptySlot++] = c;           // enqueue the character
    if (mEmptySlot > MODEM_BUFFER_SIZE) {     // Check for FIFO overflow
        mEmptySlot = 0;                      // the FIFO is circular
    }
}
```

**Figure 27. Code to Enqueue a Character in the Modem FIFO**
The listing shows the modem service routine that must be called from the ISR of the SCI driver.

The method just described allows great flexibility while handling the FIFO. For instance, to retrieve the number of characters stored in the FIFO, the software only needs to subtract `dataSlot` from `emptySlot`. Another example is the operation to flush the contents of the FIFO will simply require the statement `dataSlot = emptySlot`.

The code to dequeue (pull out of queue) a character from the FIFO is shown in Figure 28.

```c
BYTE ModemGetch (void) {
    BYTE c = 0;
    if (mDataSlot != mEmptySlot) {
        c = ModemBuffer [mDataSlot];
        mDataSlot++;
        if (mDataSlot > MODEM_BUFFER_SIZE) mDataSlot = 0;
        return(c);
    } else {
        return (BYTE)0x00;
    }
}
```

**Figure 28. Code to Dequeue a Character from the Modem FIFO**

Two important functions are also defined inside the modem module: the `Transmit()` and `Waitfor()` functions. The first transmits data to the modem while the second waits for any particular character or a string of characters before it times out. When used together, both functions provide support for complex scripts required for SLIP sessions. Obviously, those scripts will be built in the ROM code, making it difficult to maintain in some applications.
PPP Module

The PPP implementation runs on top of the hardware interface software. It provides the appropriate mechanism required for LCP, PAP, and IPCP negotiations. These negotiations are performed in a fixed state machine called by the PPPEntry() function. This machine is responsive; it builds response packets based on the contents of the received ones. This helps the user to force negotiations to go the desired way.

The PPP module defines two buffers in RAM: the InBuffer[] and OutBuffer[]. By default, each buffer is 88 bytes long. The InBuffer stores all incoming packets either from the PPP or SLIP while the OutBuffer stores the packets for output.

These buffers are defined inside the PPP module because of the exhaustive use they are exposed to at the PPP level. The buffers are global since they are used by all the other modules of the stack. Each module must define a structure describing the data arrangement they expect. Consider the situation in Figure 29.

![Figure 29. How InBuffer is Shared Between Different Protocol Modules — A Ping Response Using PPP, IP, and ICMP](image-url)
**Figure 29** shows an ECHO reply message type as received by the `PPPEntry()` function. This function then executes the IP handler which in turn passes the `ip_in` pointer to the ICMP handler. Inside this handler the ICMP data can be accessed using the *Payload* field by casting a `ICMPDatagram` struct defined in `Icmp.h`.

To fill the `InBuffer`, each time a character arrives through the serial port, the SCI ISR should pass the character to the `ProcPPPReceive()` in the case of PPP or `ProcSLIPReceive()` for SLIP. Both functions decode an entire frame once completed and validated.

The diagram in **Figure 30** illustrates this procedure.

![Diagram of PPP Module Frames Incoming Packet and Stores It in InBuffer](image)

**ProcPPPReceive()** acts as the ISR for each incoming character. Since the only way for an ISR to communicate with the main thread of execution is by means of a global variable, the PPP module defines a global status byte called `PPPStatus`. When a complete PPP frame is ready for processing, `ProcPPPReceive` sets the `IsFrame` flag. This flag is pooled by `PPPEntry()` in the application main loop.

**Listing 4. Body of PPPEntry Function** shows the body of the `PPPEntry` function. Note that this also applies to the SLIP interface module.
After a PPP packet is detected, `PPPEntry()` retrieves the protocol field from the packet and then calls the appropriate handler. If new protocols are to be implemented, handlers should be placed inside the switch statement.

Notice how the `IsFrame` flag is cleared at the end of the packet processing. This is needed to avoid frame overlapping (when a new frame is being received before the processing of the previous one occurs). Clearing the `IsFrame` flag tells the `ProcPPPReceive` routine that it can wait for another PPP packet. To do so, it must check the first occurrence of a 0x7F character (the start of a PPP packet). That is why the `ReSync` flag must be set to True. The `ReSync` flag commands the PPP framer to wait for the start of the next incoming packet.

```c
void PPPEntry (void) {
    if (PPPStatus & IsFrame) { /* Is a PPP packet available for processing? */
        switch (*(WORD *)&InBuffer [2])) {   /* Process specific protocol */
            case 0xC021:/* LCP Handler */
                HandleLCPOptions ();
            break;
            case 0xC023:/* PAP Handler */
                HandlePAPPackets ();
            break;
            case 0x8021: /* IPCP Handler */
                HandleIPCPOptions ();
            break;
            case 0x0021:/* IP Data Handler */
                IPHandler ((IPDatagram *)&InBuffer [4]);
            break;
            default:
                break;
        }
        PPPStatus &= ~IsFrame; /* Reset IsFrame Flag */
        PPPStatus |= ReSync; /* Resynchronize PPP framer */
    }
}
```

**Figure 31. Body of PPPEntry Function**
Internet Protocol Implementation

IP datagrams are handled by a switch statement inside the interface entry function `PPPEntry()` or `SLIPEntry()`. Not much happens at the IP level: Only the destination IP address is checked to see if the datagram has been intended to the M68HC08 IP address.

```c
void IPHandler (IPDatagram *ip) {
    /* Compare IP address with datagram received */
    if (!IPCompare ((BYTE *)ip->SourceAddress[0]) {  /* Misrouted datagram or broadcast message received */
        return;
    } else
    switch (ip->Protocol) {
        case UDP: /* Call UDP Handler */
            UDP_Handler ((UDPDatagram *)&ip->SourceAddress[0]);
            break;
        case TCP: /* Handle TCP segment */
            break;
        case ICMP: /* Handle ICMP commands */
            IcmpHandler ((IPDatagram *)ip);
            break;
        default: /* Transport protocol unsupported */
            break;
    }
}
```

**Figure 32. Handler of IP Packets**

At reset the `IPInit()` function must be called to initialize the IP datagram pointers to the input and output buffers, respectively. The `ip_in` and `ip_out` pointers are global, so other modules can rely on them to build and send datagrams from scratch. For instance, some ICMP messages would require access to the TTL field in an IP datagram or, in the case of UDP and TCP, calculating the pseudo-header involves the source and destination addresses from the IP header. This is why the UDP implementation defines a `UDPDatagram` structure containing the source and destination IP addresses from the IP header.
The IP implementation checks the protocol field located in the IP header to call the appropriate protocol handler. Since this application note describes UDP and some ICMP functionality, only those protocols are presented with a handler.

In case an ICMP message is received, this code is executed.

```c
switch (ip->Payload[0]) {
    case ECHO:
        Move ((BYTE *)ip, (BYTE *)ip_out, ip->Length); /* Move ping datagram to output buffer */
        /* Swap source and destination IP addresses on Output Buffer */
        ip_out->DestAddress[0] = ip->SourceAddress[0];
        ip_out->DestAddress[1] = ip->SourceAddress[1];
        ip_out->DestAddress[2] = ip->SourceAddress[2];
        ip_out->DestAddress[3] = ip->SourceAddress[3];
        ip_out->SourceAddress[0] = ip->DestAddress[0];
        ip_out->SourceAddress[1] = ip->DestAddress[1];
        ip_out->SourceAddress[2] = ip->DestAddress[2];
        ip_out->SourceAddress[3] = ip->DestAddress[3];
        ip_out->Payload[0] = ECHO_REPLY; /* This will be the echo reply */
        ip_out->Payload[1] = 0; /* Set ICMP Code to 0 */
        ip_out->Payload[2] = 0; /* Set ICMP checksum field to 0 */
        ip_out->Payload[3] = 0; /* during checksum generation */
        /* Calculate ICMP checksum */
        Value = IPCheckSum ((BYTE *)&ip_out->Payload[0], (ip->Length - 20) >> 1);
        ip_out->Payload[2] = (Value >> 8); /* Set ICMP checksum */
        ip_out->Payload[3] = (Value & 0xFF);
        IPNetSend (ip_out); /* Send ICMP packet over IP */
        break;
    case ECHO_REPLY:
        // Code to handle ping responses
        // goes here
        break;
    case TRACEROUTE:
        break;
    default:
        break;
}
```

**Figure 33. Handler of ICMP Packets**
An ECHO type message is commonly referred to as a ping request from a remote host. The handler simply swaps the source and destination IP addresses and changes the message type to ECHO_REPLY. Before the packet is sent back through the IP interface (using the IPNetSend function), a new checksum for the ICMP message must be recalculated.

The UDP implementation is not that different from the ICMP. However, since almost all UDP processing is done at the application level, the UDP module supports the use of a CALLBACK for processing incoming UDP data.

Each time an incoming IP packet containing UDP data is received by the PPP or SLIP interface, the CALLBACK function specified by UDPSetCallbackProc() is called from within the UDP handler. The UDP implementation specifies a default callback procedure in case it is not specified outside this module. The callback function has this format.

```c
void UDPReceive (BYTE *udp_data, BYTE size_of_data, WORD udp_port) {
    // Do something
}
```

Because no buffered mechanism is used in the software, the data pointer passed to the callback function points to the UDP data physically located inside the section of RAM allocated for InBuffer[]. For this reason, this data must be processed on the fly. Also there is no risk of recursivity while executing the callback function because the InBuffer and the PPP framer have been blocked by the PPPEntry() function.

## Summary

The M68HC08 has a powerful instruction set and addressing modes. With some effort, the source code presented in this application note can be highly optimized in both speed and size using the M68HC08 CPU features for the C language (not to mention the optimizations that can be achieved using assembly language).

Imagine the possibilities, and keep in mind that the MC68HC908GP32 has plenty of hardware resources to use in an Internet-enabled application: an SPI, two 2-channel timers, A/D channels, a timebase module, a keyboard interface module, and more than half the RAM and FLASH ROM of the total available.
Internet programming can be difficult sometimes, especially when the programmer has little or no experience with the inner aspects of the TCP/IP protocol suite. This document serves as a good introduction to such exciting technology. Remember that when the appropriate tools and utilities are in place, getting the knowledge to create Internet applications can be achieved easily through experimentation.

The software presented in this document can be used as a reference for more professional and serious applications. Improving the software should be easy. Here is a hint: Because buffering is used, adding more hardware interfaces should be easy. Just code the appropriate framer for input and output, define a global flag to signal events to the application main loop, share the InBuffer with the ip_in pointer, and you are finished.

Perhaps the reader can argue that the buffer approach is slow and inappropriate for a small MCU, but it has been proven that the M68HC08 supports it easily. Besides, there is no reason to avoid a byte-by-byte processing technique. The CPU can process and validate incoming packets on the fly without storing headers or trailers reducing the amount of RAM required to store a packet.

The same applies to outgoing packets when there is enough information on memory to reproduce them. Perhaps this would be the job of a SOCKET structure running on top of the PPP implementation. It is just a matter of sitting down, coding, and experimenting with the M68HC08. A creative programmer with an Internet-ready M68HC08 can be a powerful combination.
Figure 34. MC68HC908GP32 UDP/IP Implementation
The source code of this application note is described in Table 8.

### Table 8. Code Statistics

<table>
<thead>
<tr>
<th>Segment</th>
<th>Location Org</th>
<th>Location End</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-initialized data in zero page RAM</td>
<td>0x0040</td>
<td>0x0044</td>
<td>4</td>
</tr>
<tr>
<td>Non-initialized data in RAM</td>
<td>0x0067</td>
<td>0x0128</td>
<td>193</td>
</tr>
<tr>
<td>Program code</td>
<td>0xB000</td>
<td>0xC513</td>
<td>5395</td>
</tr>
<tr>
<td>Program initialized RAM</td>
<td>0x0045</td>
<td>0x0066</td>
<td>33</td>
</tr>
<tr>
<td>Text string and constants</td>
<td>0xC53E</td>
<td>0xC7C8</td>
<td>650</td>
</tr>
<tr>
<td>Vector table</td>
<td>0xFFDC</td>
<td>0xFFFF</td>
<td>35</td>
</tr>
<tr>
<td>Total RAM</td>
<td></td>
<td></td>
<td>230</td>
</tr>
<tr>
<td>Total ROM</td>
<td></td>
<td></td>
<td>6080</td>
</tr>
</tbody>
</table>
Main.C
Application Main Function

#include <iogp20.h>
#include "CommDrv.h"
#include "ModemDrv.h"
#include "ppp.h"
#include "UDP.h"
#include "IP.h"
#include "SLIP.h"

// define USE_SLIP
// Uncomment this line if SLIP is to be used


const char * ModemCommand[] = {  // Array of modem initialization commands
 "ATZ\r",  // Reset Command
 "ATE0\r",  // Disable Echo
 "AT&C1\r",  // Track presence of data carrier
 "AT&D3\r"  // Reset modem when an on-to-off transition of DTR occurs
};

/****************************************************************************************************
Function : ModemHandler
Parameters : Code - Numeric response code from a Modem dial command
Date : January 2001
Desc : This function handles the numeric responses from a dial command issued to the modem
***************************************************************************************************/
void ModemHandler (BYTE Code) {
    switch (Code) {
        case '0': // OK
            break;
        case '1': // CONNECT
            #ifdef USE_SLIP
                CommEventProc (ProcSLIPReceive); // Install SLIP Service
            #else
                ModemBuffFlush (); // Flush contents of Modem Buffer
                if (ModemGetch () != 0x7F) { // Test for PPP packets
                    Waitfor (":", 100); // Wait for "Username:" of ISP script
                    PPPSendVoidLCP (); // ForcePPPtransactionsinsteadof scripts
                }
            #endif
    }
}
CommEventProc (ProcPPPReceive); // Install PPP service routine
break;

case '2': // RING
break;

case '3': // NO CARRIER
break;

case '4': // ERROR
break;

case '6': // NO DIAL TONE
break;

case '7': // BUSY
break;

case '8': // NO ANSWER
break;

case '9': // CONNECT 2400
break;

default: // TIME OUT, NO RESPONSE FROM MODEM RECEIVED!
break;

}/***********************************************************************
Function : UDPReceive
Parameters : Data of UDP packet,
size - size of data in bytes
RemoteIP - sender IP address
port - UDP port number
Date : January 2001
Desc : This function is executed each time a UDP packet is received
and validated.
***********************************************************************/
void UDPReceive (BYTE *data, BYTE size, DWORD RemoteIP, WORD port) {
  switch (port) {
    case 1080: // If port number equals 1080 then reply
      ADSCR &= 0x00; // Get an A/D lecture
      while (!(0x80 & ADSCR));
      udp_out->Payload [0] = ADR; // Format UDP payload
      UDPSendData ((BYTE *)&RemoteIP, 11222, 0, 1); // Send UDP reply
      break;
  }
}
case 1081:    // Port = 1081, reply with ADC ch1
    ADSCR &= 0x01;
    while (!(0x80 & ADSCR));
    udp_out->Payload [0] = ADR;
    UDPSendData ((BYTE *)&RemoteIP, 11222, 0, 1);
    break;

case 1082:    // Data through UDP port 1082
    // Do something here
    break;

case 1083:    // Data through UDP port 1083
    break;
}
}

/***************************************************/
Function : LinkTask
Parameters : None
Date : January 2001
Desc : This function synchronize the phone line with the PPP link.

/***************************************************************************/
void LinkTask (void) {
    if ((PPPStatus & LinkOn) && (!ModemOnLine())) {// PPP Link ON while Phone is // on-hook!
        PPPStatus &= ~LinkOn;       // Clear PPP link flag
        PORTC = 0x00;
        CommEventProc (ProcModemReceive); // Install Modem handler
    }
}

/***************************************************************************/
Function : ApplicationTask
Parameters : None
Date : January 2001
Desc : This function checks channel 2 of the A/D and sends a warning message to a remote server using UDP if a conversion is higher than hexadecimal 0x35.

/***************************************************************************/
void ApplicationTask (void) {
    ADSCR &= 0x02; // Test A/D channel 2
    while (!(0x80 & ADSCR)); // Wait for A/D conversion
    if (ADR > 0x35) { // If sample is above 0x35
        // Send a notification
    }
}
if (!ModemOnLine ()) { // Test if Modem on-line
NoOperation; // Modem Not on-line,
// we can re-dial here
}
UDPSendData ((BYTE *)&RemoteServer, 8010, "Warning from HC08!", 18);
}

///////////////////////////////////////////////////////////////////////////////
// M A I N
///////////////////////////////////////////////////////////////////////////////
void main(void) {
InitPLL (); // Init PLL to 4.9152MHz
CONFIG1 = (BYTE)0x0B; // LVI operates in 5-V mode,
// STOP instruction enabled
// COP Module Dissabled
CONFIG2 = (BYTE)0x03; // Oscillator enabled to operate during stop mode
// Use internal data bus clock as source for the SCI
PORTC = 0; // Set PortC to 0
DDRC = 0xFF; // Set PortC direction to output
IPInit (); // Initialize IP
#endif
#else
SLIPInit (); // Initialize SLIP implementation
IPBindAdapter (SLIP); // Send IP packets using SLIP format
#endif
PPPInit (); // Initialize PPP interface
IPBindAdapter (PPP); // Send IP packets using PPP format
UDPSetCALLBACK (UDPReceive); // Set Callback for incoming UDP data
ModemInit (); // Modem Init
ModemBindBuff (PPPGetInputBuffer()); // Set Modem Buffer for command reception
CommEventProc (ProcModemReceive); // re-direct incoming SCI characters to the
// modem interface
OpenComm (BAUDS_2400,
ENABLE_RX |
ENABLE_TX |
ENABLE_RX_EVENTS);
{ // Create some stack variables
BYTE Res = 0; // Create two temp vars in the stack
BYTE index;

for (index = 0; index <= 3; index++) { // Loop through Modem initialization
// commands
transmit (ModemCommand [index]); // Transmit modem command
Res = Waitfor ("OK", 30); // Wait for OK
if (!Res) { // Invalid response received
// Do something here

AN2120
For More Information On This Product,
Go to: www.freescale.com
ModemReset (); // Reset modem
index = 0;   // Loop again
}
}
Res = ModemDial("6842626"); // Dial ISP
ModemHandler (Res);  // Handle Modem response
}
EnableInterrupts;
for (;;) { // Application Loop
#ifdef USE_SLIP
SLIPEntry();  // Poll SLIP packets
#else
LinkTask ();  // Synchronize PPP link with Modem
PPPEntry ();  // Poll for PPP packets
#endif
ApplicationTask (); // Call application
}

CommDrv.C
Serial Communications Interface Driver

File Name : CommDrv.c
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : July 2000
Current Revision : 0.0
Notes : This file contains the code to drive the serial port
#include "CommDrv.h"
static void CommDrvDefaultProc (register BYTE value);
static void (* EvtProcedure) (register BYTE value) = CommDrvDefaultProc;

Function :  CommDrvDefaultProc
Parameters :
Date : July 2000
Desc :
static void CommDrvDefaultProc (BYTE value) {
    (void)value;
}

Function : UseDefaultCommProc

Parameters :

Date : July 2000

Desc :

**********************************************************************
void UseDefaultCommProc (void) {
    DisableInterrupts;
    EvtProcedure = CommDrvDefaultProc;
    EnableInterrupts;
}

Function : OpenComm

Parameters :

Date : July 2000

Desc :

**********************************************************************
void OpenComm (register BYTE BaudRate, register CommOptions Options) {
    SCBR = BaudRate; // Set the baud rate
    SCC1 = 0x40; // Enable baud rate generator //
    SCC2 = Options;
}

Function : CloseComm

Parameters :

Date : July 2000

Desc :
Function : AssignCommEventProc

Parameters :

Date : July 2000

Desc :

/*******************************************************************************
 Function : AssignCommEventProc
 Parameters :
 Date : July 2000
 Desc :
*******************************************************************************/

void AssignCommEventProc (EventProc Proc) {
    DisableInterrupts; // Disable Interrupts
    EvtProcedure = Proc; // Install service handler
    EnableInterrupts; // Enable interrupts
}

Function : WriteComm

Parameters :

Date : July 2000

Desc :

/*******************************************************************************
 Function : WriteComm
 Parameters :
 Date : July 2000
 Desc :
*******************************************************************************/

void WriteComm (BYTE c) {
    SCDR = c; // Write char to SCI data register
    while (!(SCS1 & 0x80)); // Wait until character gets transmitted
}

Function : ReadComm

Parameters :

Date : July 2000

Desc :

/*******************************************************************************
 Function : ReadComm
 Parameters :
 Date : July 2000
 Desc :
*******************************************************************************/

BYTE ReadComm (void) {
    while (!(SCS1 & 0x20));
    return SCDR;
}

Function : WriteCommStr
Parameters:

Date: July 2000

Desc:

***********************************************************************/
void WriteCommStr (char* string) {
    while (*string) {
        SCDR = *string++;
        while (!(SCS1 & 0x80));
    }
}

***********************************************************************/
Function: CommRx
Parameters:

Date: July 2000

Desc:

*******************************************************************************
void @interrupt UartRxISR (void) {
    SCS1; // acknowledge this IRQ
    EvtProcedure (SCDR); // Forward the character to a service routine
}

*******************************************************************************
SLIP.C
Serial Line Internet Protocol Implementation Module

/*******************************************************************************
File Name: SLIP.C
Author: Rene Trenado
Location: Freescale Applications Lab, Baja California
Date Created: September 2000
Current Revision: 0.0
Notes: This file contains the code for the SLIP module
*******************************************************************************
#include "CommdRV.h"
#include "slip.h"
#include "IP.h"
#include "Icmp.h"
#include "udp.h"

static BYTE *SLIP_Packet; // local pointer to the SLIP buffer */
BYTE SLIPStatus = 0;       // status and control byte of the
SLIP module */
static volatile BYTE FrameSize = 0; // provides internal control for SLIP buffer
management */

/******************************************************************************
Function : ProcSLIPReceive
Parameters : A Byte character to stream in a SLIP Packet
Date : August 2000
Desc : This function process a BYTE following SLIP popular
       specification. The Async event on input driver should
       call this function (usually the COMM ISR).
*******************************************************************************/
void ProcSLIPReceive (BYTE c) {
    if (SLIPStatus & IsFrame) return;

    if (SLIPStatus & ReSync) {// Ignore incoming data until a start of
                              // packet is found
        if (c != 0xC0) {
            return;
        }
        SLIPStatus &= ~ReSync; // Clear the synchronization flag to stream
        // incoming packet in SLIP buffer
        FrameSize = 0; // FrameSize records size of incoming
        // packets
    }

    if (SLIPStatus & IsESC) { // Is the byte received a control char?
        switch (c) { // if so decode it
            case ESC_END: // Store Special char on Input Buffer
                SLIP_Packet [FrameSize++] = SLIP_END;
                break;
            case ESC_ESC: // Store Special char on Input Buffer
                SLIP_Packet [FrameSize++] = SLIP_ESC;
                break;
            default: // SLIP Protocol violation
                break;
        }
    }
}
SLIPStatus &= ~IsESC; // Clear the special control character flag

else {
    switch (c) {
    case SLIP_ESC: // Special ESC Character received
        SLIPStatus |= IsESC;
        break;

    case SLIP_END: // Special END Character received
        if (FrameSize > 0) { // Avoid zero length packets
            SLIP_Packet[FrameSize] = 0; // Append a NULL character
            SLIPStatus |= IsFrame; // Signal Frame availability
            // Extra control processing can be done here
            /* ..... */
        }
        break;

    default: // Data of Packet received
        SLIP_Packet[FrameSize++] = c; // Store Byte
        // Avoid & discard large SLIP packets
        if (FrameSize > (SLIP_MAX_SIZE)) {
            FrameSize = 0;
            // Resynchronize SLIP packet reception
            SLIPStatus |= ReSync;
        }
        break;
    }
}

/*******************************************************************************
Function : SLIPInit
Parameters : None
Date : September 2000
Desc : Initialize the SLIP Module
*******************************************************************************/
void SLIPInit (void) {
    SLIPStatus |= ReSync;
    SLIP_Packet = (BYTE *)ip_in;
}

/*******************************************************************************
Function : ProcSLIPSend
Parameters : Buffer: a pointer to a buffer containing the IP packet to send
             len: the size of the SLIP packet
Date : September 2000
*******************************************************************************
Desc : Sends a BYTE array of len length following the popular SLIP format

********************************************
void ProcSLIPSend (BYTE *ptr, BYTE len) {
    WriteComm (SLIP_END); // Write start of SLIP frame
    while (len--) { // Send all buffer in SLIP format
        switch (*ptr) { // check to see if is a special character
            case SLIP_END:
                WriteComm (SLIP_ESC); // escape special character
                WriteComm (ESC_END);
                break;
            case SLIP_ESC:
                WriteComm (SLIP_ESC); // escape special character
                WriteComm (ESC_ESC);
                break;
            default:
                WriteComm (*ptr); // send raw character
                ptr++;
                break; // continue with next character send
        }
    }
    WriteComm (SLIP_END); // Write END of SLIP frame
}

 /***************************************************************************/
Function : SLIPEntry
Parameters : None
Date : August 2000
Desc : SLIP Module Entry, Applications should call SLIPEntry frequently in the main loop or in portions of the app code.

***************************************************************************/
void SLIPEntry (void) {
    if (SLIPStatus & IsFrame) {
        if (!IPCompare (&ip_in->DestAddress[0])) {
            /* Misrouted datagram or broadcast message received */
            /* Do extra processing here */
        } else {
            switch (ip_in->Protocol) { /* Select protocol handler */
                case UDP:
                    UDP_Handler ((UDPDatagram *)&ip_in->SourceAddress[0]);
                    break;
                case TCP:
                    break;
            }
        }
    }
}
case ICMP:
    IcmpHandler ((IPDatagram *)ip_in);
    break;

    default:
    break;

} 
} 

SLIPStatus &= ~IsFrame;    /* Acknowledge datagram processing */
SLIPStatus |= ReSync;      /* Synchronize packet reception */

PPP.C
Point-to-Point Protocol Implementation

(inflater)-------------------------------------------------------------------
File Name : PPP.C
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : September 2000
Current Revision : 0.0
Notes : This file contains the code for the PPP module
(inflater)="/*********************************************************/
#include <iogp20.h>
#include <string.h>
#include "CommDrv.h"
#include "ppp.h"
#include "IP.h"
#include "Udp.h"
#include "ICMP.h"

const char * User = "MyName";    // Username of ISP account
const char * Password = "MyPassword";  // Password of username

(inflater)="/***********************************************************/
static void HandleLCPOptions (void);
static void HandlePCPOptions (void);
static WORD PPPfcs16 (WORD fcs, BYTE *cp, int len);
static void RejectProtocol (BYTE *InBuffer);
Application Note

//////////////////////////// Protected ROM Data //////////////////////////////
static const BYTE PPPData[] = {
  0xff, 0x03, 0xc0, 0x21, 0x02, 0x01, 0x00, 0x04, 0x00, 0x00, 0x00
};

static const BYTE LCPTerminate[] = {
  0xff, 0x03, 0xc0, 0x21, 0x05, 0x04, 0x00, 0x04, 0x80, 0xfe
};

static const unsigned short fcstab[256] = {
  0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
  0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
  0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0x643e,
  0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
  0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
  0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0x76a5, 0x672c, 0x34ab, 0x2522,
  0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
  0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xeaa6, 0xd8fd, 0xc974,
  0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
  0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xaa72, 0xbbf9,
  0x7387, 0x620e, 0x5095, 0x411c, 0x3393, 0x221a, 0x56a5, 0x472c,
  0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
  0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
  0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0x76a5, 0x672c, 0x34ab, 0x2522,
  0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
  0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xeaa6, 0xd8fd, 0xc974,
  0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
  0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xaa72, 0xbbf9,
};

////////////////////////// Public RAM Data /////////////////////////////
volatile BYTE PPPStatus = 0;
BYTE InBuffer[PPP_BUFFER_SIZE + 1]; // Input Buffer for PPP data
BYTE OutBuffer[PPP_BUFFER_SIZE + 1]; // Output Buffer for PPP data

static volatile BYTE *PPP_Packet = InBuffer;
static volatile BYTE FrameSize = 0;
static EventProc PPPEntryProc;
/*******************************************************************************
Function : PPPInit
Parameters : None
Date : September 2000
Desc : Initialize the PPP Module
*******************************************************************************/
void PPPInit (void) {
    PPPStatus |= ReSync;
}

/*******************************************************************************
Function : PPPGetInputBuffer
Parameters : None
Date : September 2000
Desc : Returns a PPP Input Buffer pointer to caller
*******************************************************************************/
BYTE *PPPGetInputBuffer (void) {
    return &InBuffer[0];
}

/*******************************************************************************
Function : PPPGetOutputBuffer
Parameters : None
Date : September 2000
Desc : Returns a pointer to PPP Output Buffer to caller
*******************************************************************************/
BYTE *PPPGetOutputBuffer (void) {
    return &OutBuffer[0];
}

/*******************************************************************************
Function : PPPfcs16
Parameters : fcs: current fcs
cp: pointer to PPP data
*******************************************************************************
len: size of PPP data

Date: September 2000

Desc: Calculate a new fcs given the current fcs and the new data.

******************************************************************************
static WORD PPPfcs16 (WORD fcs, BYTE *cp, int len) {
    while (len--)
        fcs = (fcs >> 8) ^ fcstab[(fcs ^ *cp++) & 0xff];
    return (fcs);
}

******************************************************************************
Function: public PPPGetChecksum
Parameters: cp: A pointer to the PPP Packet
            len: Size of PPP Packet
Date: September 2000

Desc: Returns the Checksum of the PPP Packet pointed by cp
******************************************************************************
WORD PPPGetChecksum (register unsigned char *cp, register int len) {
    return ~PPPfcs16( PPPINITFCS16, cp, len );
}

******************************************************************************
Function: ProcPPPReceive
Parameters: A Byte character to stream in a PPP Packet
Date: August 2000

Desc: This function process a BYTE following HDLC - PPP specifications. The Async event on input driver should call this function (usually the COMM ISR).
******************************************************************************
void ProcPPPReceive (register BYTE c) {
    PPPStatus |= ByteRx;

    if (PPPStatus & IsFrame) return;

    if (PPPStatus & ReSync) {
        if (c != 0x7E) return;
        PPPStatus &= ~ReSync;
        FrameSize = 0;
    }

    if (PPPStatus & IsESC) {
        PPP_Packet [FrameSize++] = 0x20 ^ c;
    }
PPPStatus &= ~IsESC;
}
else {
    switch (c) {
    case ESC: // Special ESC (0x7D) Character received
        PPPStatus |= IsESC;
        break;
    case END: // Special END (0x7E) Character received
        // Avoid zero length packets (0x7F - 0x7F conditions);
        if (FrameSize > 0) {
            PPP_Packet [FrameSize] = 0;
            PPPStatus |= IsFrame; // Signal Frame availability
        }
        break;
    default:
        PPP_Packet [FrameSize++] = c;
        if (FrameSize > (PPP_BUFFER_SIZE - 6)) {
            FrameSize = 0;
            PPPStatus |= ReSync;
        }
        break;
    }
}

().'/***********************************************************************
Function : PPPSend
Parameters : Buffer: A pointer to a buffer containing the PPP packet to send
            len: the size of the PPP packet
Date : September 2000
Desc : Sends a BYTE array of len length following HDLC - PPP specifications
***********************************************************************/
void ProcPPPSend (BYTE *Buffer, BYTE len) {
    WORD Checksum = 0;
    Checksum = PPPGetChecksum (Buffer, Buffer[7] + 4);
    Buffer [Buffer[7]+5] = (Checksum >> 8) & 0xFF;

    WriteComm (0x7E);
    while (len--) {
        if ((signed char)*Buffer < (signed char)0x20) {
            WriteComm (0x7D);
            WriteComm (*Buffer ^ 0x20);
        } else {
switch (*Buffer) {
    case 0x7E:  
        WriteComm (0x7D);  
        WriteComm (0x5E);  
        break;  
    case 0x7D:  
        WriteComm (0x7D);  
        WriteComm (0x5D);  
        break;  
    default:  
        WriteComm (*Buffer);  
        break;  
}  
    Buffer++;  
}  
WriteComm (0x7E);  

/***********************************************************************************/
Function : public PPPFrameSize
Parameters : None
Date : August 2000
Desc : Returns the size of the current available PPP packet stored in InBuffer. Caller should call this function if needed only when the IsFram flag has been signaled.
/***********************************************************************************/
BYTE PPPFrameSize (void) {
   return FrameSize;
}

/***********************************************************************************/
Function : protected HandleLCPOptions
Parameters : None
Date : August 2000
Desc : State Machine that implements LCP packet negotiation
/***********************************************************************************/
static void HandleLCPOptions (void) {
BYTE *dest = OutBuffer;  // A pointer to the options of output buffer
BYTE *ptr = (BYTE *)&InBuffer[8];  // A pointer to the options of input buffer
switch (InBuffer[4]) {

    //++++++++++++++++++++++++++++++++++++++++++++++++++++
    case TERMINATE: //Server Terminate-Request received
        Move (InBuffer, OutBuffer, InBuffer[7] + 6);
        OutBuffer[4] = TERMINATE_ACK;
        ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
        PPPStatus &= ~LinkOn;
        break;

    //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    case REQ:

    /////////////////////////////////////////////////////////
    /////////// Server requesting option 2 ///////////
    /////////////////////////////////////////////////////////
    if ((InBuffer[8] == 0x02) && (InBuffer[7] <= 0x0A)) {
        if ((InBuffer[10] == 0xFF) &&
            (InBuffer[11] == 0xFF) &&
            (InBuffer[12] == 0xFF) &&
            (InBuffer[13] == 0xFF)) {
            InBuffer[4] = ACK;
            ProcPPPSend (InBuffer, InBuffer[7] + 6);
            return;
        }
    } else

    /////////////////////////////////////////////////////////
    /////////// Server requesting first options, reject all but 3 //////////
    /////////////////////////////////////////////////////////
    if ((InBuffer[8] != 0x03) && (InBuffer[7] > 9)) {
        BYTE OptionsSize;
        BYTE Option;
        BYTE Size;
        Move (InBuffer, OutBuffer, 8) // Move LCP header to output buffer
        OutBuffer[4] = REJ; // Output will be a reject packet
        dest += 8; // Offset output pointer to
        // LCP options
        OptionsSize = InBuffer[7] - 4; // Get size of LCP
        // options
        while (OptionsSize > 0) {
            // Is there options to
            // process?
            Option = *(ptr);
            // Get option number
            Size = *(ptr + 1); // Get size of this option
            OptionsSize -= Size; // Reduce the amount of
            // OptionsSize
            if (Option == 3) { // Is this option 3?
                // (authentication protocol)
ptr += Size; // Remove this option in output packet
// Set New Packet size
}
else {
    // Copy this option to the output buffer
    while (Size-- ) {
        *dest++ = *ptr++;
    }
}
}

/////////////////////////////////////////////////////////////////////
/////// Server Request CHAP protocol, We reply with
/////// a suggestion of the PAP protocol instead
/////////////////////////////////////////////////////////////////////
if ((InBuffer [8] == 0x03) && (InBuffer [10] == 0xC2)) {
    InBuffer [4] = NAK;         // NAK CHAP protocol
    InBuffer [10] = 0xC0; // We suggest PAP instead
    // Send the NAK reply
    ProcPPPSend (InBuffer, InBuffer[7]+6);
    return;
} else

/////////////////////////////////////////////////////////////////////
/////// Server Request PAP protocol //////////
/////// We Acknowledge this reply and then we start negotiating
/////// the Async-Control-Char..., Here we send both packets!!!
/////////////////////////////////////////////////////////////////////
if ((InBuffer [8] == 0x03) && (InBuffer [10] == 0xC0)) {
    Move (InBuffer, OutBuffer, InBuffer[7]+6);
    OutBuffer[4] = ACK;
    ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);

    OutBuffer[4] = REQ;
    OutBuffer[7] = 0x0A;
    OutBuffer[8] = 0x02;
    OutBuffer[9] = 0x06;
    OutBuffer[10] = 0xFF;
    OutBuffer[11] = 0xFF;
    OutBuffer[12] = 0xFF;
    OutBuffer[13] = 0xFF;
}
    ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
break;

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
case ACK:

// For More Information On This Product,
Go to: www.freescale.com
/// Server Acknowledge Async Control ///
if (InBuffer[8] == 0x02) {
    SendPAPPacket (REQ, InBuffer[5] + 1, User, Password);
} else {
    break;
}

case NAK:
break;

case REJ:
break;

case TERMINATE_ACK: // Terminate ACK!
    PPPStatus &= ~LinkOn;
    break;
} else {
    return;
}

/** Function : protected HandleIPCPOptions */
/** Parameters : None */
/** Date : August 2000 */
/** Desc : State Machine that implement IPCP packet negotiation */
static void HandleIPCPOptions (void) {
    BYTE *dest = (BYTE *)&OutBuffer[8];
    BYTE *ptr = (BYTE *)&InBuffer[8];
    BYTE FrameSize;
    BYTE Option;
    BYTE Size;
    switch (InBuffer[4]) {
        case REQ:
            if ((InBuffer[8] != 0x03) && (InBuffer[7] > 0x0A)) {
                OutBuffer[0] = 0xFF; // Build a IPCP header
                OutBuffer[1] = 0x03;
                OutBuffer[2] = 0x80; // Set IPCP protocol
                OutBuffer[3] = 0x21;
                OutBuffer[4] = REJ; // This will be a REJ packet for now
                OutBuffer[5] = InBuffer[5];
                FrameSize = InBuffer[7] - 4;
                // Ignore all but option #3
            }
    }
}
while (FrameSize > 0) {
    Option = *ptr;
    Size = *(ptr + 1);
    FrameSize -= Size;
    if (Option == 3) {
        ptr += Size;
        // Set New Packet size
    } else {
        while (Size-- ) {
            *dest++ = *ptr++;
        }
    }
}
else {
    // Acknowledge IP Address //
    Move (InBuffer, OutBuffer, InBuffer[7]+6);
    OutBuffer[4] = ACK;
    ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
    // Now Request IP address to complete 3-way handshake
    OutBuffer[4] = REQ; // Request command
    OutBuffer[10] = 0; // IP address is set // to 0 so ISP server
    OutBuffer[11] = 0; // can assing us one
    OutBuffer[12] = 0;
    OutBuffer[13] = 0;
}
ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
break;
case ACK:
    if (InBuffer[8] == 3) {
        // Reply of the only IPCP
        // Request we can send
        IPAddress[0] = InBuffer[10]; // ISP assigned IP
        IPAddress[1] = InBuffer[11];
        IPAddress[2] = InBuffer[12];
        IPAddress[3] = InBuffer[13];
        PORTC = 0xFF;
        PPPStatus |= LinkOn; // PPP Link is now up
    }
    break;
case NAK:
    if (InBuffer[8] == 0x03) {
        // Request IP Address //
        Move (InBuffer, OutBuffer, InBuffer[7]+6);
        OutBuffer[4] = 0x01;
    }
    break;
ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
}
break;
case REJ:
break;
}

/********************************************************************************
Function : public PPPSendPAPPacket
Parameters : Action: REQ, REJ, NAK
ID: Sequence number of PPP packet
user: User name for login
password: Password in plain text
Date : September 2000
Desc : Formats a PAP packet on Output Buffer. This function supports the type field for future implementation of the PPP module in server mode.
********************************************************************************/
void SendPAPPacket (BYTE Action, BYTE ID, char* user, char* password) {
    OutBuffer [0] = 0xFF;
    OutBuffer [1] = 0x03;
    OutBuffer [2] = 0xC0; // Format PAP packet header
    OutBuffer [3] = 0x23;
    OutBuffer [6] = 0;
    OutBuffer [7] = strlen (user) + strlen (password) + 6; // Set length of PAP
    OutBuffer [8] = strlen (user); // Set length of Username
    Move (user, &OutBuffer [9], strlen (user)); // Store Username
    OutBuffer [9 + strlen (user)] = strlen (password); // Set length of password
    Move (password, &OutBuffer [10 + strlen (user)], strlen (password));
    ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6); // Send PAP packet
}

/********************************************************************************
Function : Move
Parameters : src: A pointer to the data to copy
dest: A pointer to the destination location
numBYTES: Number of bytes to copy
Date : September 2000
Desc : Copies a block of numBYTES bytes from src pointer
********************************************************************************/
void Move (BYTE *src, BYTE *dest, register numBYTEs) {
    if ( numBYTEs <= 0 ) return;
    if ( src < dest ) {
        src += numBYTEs;
        dest += numBYTEs;
        do {
            *--dest = *--src;
        } while ( --numBYTEs > 0 );
    } else {
        *dest++ = *src++;
    } while ( --numBYTEs > 0 );
}

Function : protected RejectProtocol
Parameters : InBuffer -> A pointer to the buffer that has the PPP Packet to reject
Date : August 2000
Desc : Rejects the a PPP packet based on its Protocol field Stored on InBuffer

static void RejectProtocol (BYTE *InBuffer) {
    OutBuffer[0] = 0xFF;
    OutBuffer[1] = 0x03;
    OutBuffer[2] = 0xC0;
    OutBuffer[3] = 0x21;
    OutBuffer[4] = 0x08;
    OutBuffer[5] = 20;
    OutBuffer[6] = 0;
    Move (&InBuffer[2], &OutBuffer[8], InBuffer[7] + 2);
    ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
}

Function : protected PPPSendVoidLcp
Parameters : None
Date : September 2000
Desc : Sends a void LCP packet with no options to the PPP Server.
This will force the server to reply with his options to negotiate. Some ISPs require scripts to establish a connection thus a void LCP packet will try to force the server to negotiate PPP.

```c
void PPPSendVoidLCP (void) {
    WORD Checksum;
    Move (PPPData, OutBuffer, PPPData[7] + 6);
    ProcPPPSend ((BYTE *)OutBuffer, OutBuffer[7] + 6);
}
```

/**
 * Function : PPPTerminate
 * Parameters : None
 * Date : September 2000
 * Desc : Terminates a PPP link by sending a terminate LCP packet
 **/ void PPPTerminate (void) {
    Move ((BYTE *)LCPTerminate, OutBuffer, 10);
    ProcPPPSend (OutBuffer, 10);
}

/**
 * Function : PPPEntry
 * Parameters : None
 * Date : August 2000
 * Desc : PPP Module Entry, Applications should call PPPEntry frequently in the main loop or in portions of the app code.
 **/ void PPPEntry (void) {
    if (PPPStatus & IsFrame) {
        switch (*(WORD *)&InBuffer [2])) {
            case LCP_PACKET: // LCP Handler
                HandleLCPOptions ();
                break;
            case PAP_PACKET: // PAP Handler
                if (InBuffer [4] == 0x02) {// Authentication OK
                    NoOperation;
                }
                break;
            case IPCP_PACKET: // IPCP Handler
                HandleIPCPOptions ();
                break;
        }
    }
}
case IP_DATAGRAM: // IP Data Handler
    if (!IPCompare ((BYTE *)&InBuffer [20])) {
        // Misrouted datagram or broadcast
        // message received
    } else {
        switch (InBuffer [13]) {
            case UDP:
                UDP_Handler ((UDPDatagram *)&InBuffer[16]);
                break;
            case TCP:
                break;
            case ICMP:
                IcmpHandler ((IPDatagram *)&InBuffer[4]);
                break;
            default:
                break;
        }
        break;
    }
    default:
        RejectProtocol (InBuffer); // Cannot handle this type of packet
        break;
} // End of switch statement
PPPStatus &= ~IsFrame;
PPPStatus |= ReSync;
} // End of if IsFrame

ModemDrv.C
Modem Support Routines

/*================================================================----------*/
File Name : ModemDrv.C

Author : Rene Trenado

Location : Freescale Applications Lab, Baja California

Date Created : December 2000

Current Revision : 0.0

Notes : This file contains the functions required to handle an external modem
/*================================================================----------*/
#include <iogp20.h>
#include "CommDrv.h"
#include "ModemDrv.h"

#define MODEM_BUFFER_SIZE 32 // Size of Modem Buffer

#define DTR_ON PORTD &= 0xFE; // DTR Pin is PORTD0, Macro to set it ON
#define DTR_OFF PORTD |= 0x01; // Macro to set DTR OFF
#define DTR_PIN (PORTD & 0x01) // DTR Pin = Pin 0 of PORT D

// Byte pointers of the ring buffer (FIFO)
volatile BYTE mDataSlot = 0; // Points to the next available character
volatile BYTE mEmptySlot = 0; // Points to next available slot of the FIFO
static BYTE *ModemBuffer; // Pointer to Modem buffer

/******************************************************************************
Function : ModemInit
Parameters : None
Date : December 2000
Desc : Initializes the ring buffer & clears the DTR pin
*******************************************************************************/
void ModemInit (void) {
    mDataSlot = 0; // Initialize FIFO Modem pointers
    mEmptySlot = 0;
    DDRC |= 0x01; // DTR pin set to output
    DTR_OFF; // DTR Off
}

/******************************************************************************
Function : ModemBuffFlush
Parameters : None
Date : January 2001
Desc : Flushes the receiving FIFO (ring buffer)
*******************************************************************************/
void ModemBuffFlush (void) {
    mDataSlot = mEmptySlot;
}

/******************************************************************************
Function : ModemDial
Parameters : A string containing the phone number to dial
Date : December 2000
*******************************************************************************/
Desc: It sets the modem response mode to numeric (instead of verbose), then it dials a phone number & sets the DTR pin. This function returns a numeric code describing a response from the modem or a timeout. Applications should handle this response code.

***********************************************************************/
BYTE ModemDial (char * Number) {
    signed char delayCount = 80;
    transmit ("ATV0\r");  // Force a numeric response from modem
    if (!Waitfor ("0", 30)) { // Wait for an OK response
        return -1;
    }
    DTR_ON;  // Set DTR to ON
    transmit ("ATDT");  // Dial the ISP number
    transmit (Number);
    transmit ("\r");
    ModemBuffFlush ();  // Flush contents of buffer
    // Wait for a reply
    while (!!(ModemBuffNotEmpty()) && (--delayCount > 0)) {
        Delay (250);
    }
    if (delayCount) {
        return ModemGetch ();  // Return the numeric response to caller
    }
    return -1;  // No response received from modem
}

/***************************************************************************/
Function: ModemHangUp
Parameters: None
Date: December 2000
Desc: This function clears DTR to force the modem to hang up if it was on line and/or make the modem to go to command mode.
***************************************************************************/
void ModemHangUp (void) {
    DTR_ON;  // Make a DTR transition to hang-up
    Delay (40);  // Wait a couple of milliseconds
    DTR_OFF;  // Finish the DTR transition
}
Function : ModemOnLine
Parameters : None
Date : January 2001
Desc : Returns the status of the CD (carrier detect) signal.

BYTE ModemOnLine (void) {
    return (PORTD & 0x02) ^ 0x02; // Return the status of the CD line
}

Function : ModemBindBuff
Parameters : A pointer to a buffer in RAM
Date : January 2001
Desc : Binds the FIFO capabilities of this module to a buffer in RAM.

void ModemBindBuff (BYTE *lpInBuffer) {
    ModemBuffer = lpInBuffer;
    ModemBuffer [0] = 0;
}

Function : ModemReset
Parameters : None
Date : January 2001
Desc : Resets the Modem

void ModemReset (void) {
    ModemInit ();
}

Function : ModemBuffNotEmpty
Parameters : None
Date : January 2001
Desc : Returns True if modem buffer NOT empty, false otherwise.
/*
 * ModemBuffNotEmpty (void) {
 *     return !(mDataSlot == mEmptySlot);
 * }
 */

/*/ Function: ModemInBufferCount */

BYTE ModemInBufferCount (void) {
    if ((mEmptySlot - mDataSlot) >= 0)
        return (BYTE)(mEmptySlot - mDataSlot);
    else {
        return (BYTE)((mEmptySlot + MODEM_BUFFER_SIZE) - mDataSlot);
    }
}

/*/ Function: Waitfor */

BYTE Waitfor (char *String, BYTE Time) {
    BYTE c = 0;
    BYTE Offset = 0;
    while (Time-- > 0) {
        Delay (100); // Wait ~ 150 mSec
        while (ModemBuffNotEmpty()) { // Wait for characters
            c = ModemGetch (); // Extract a character from FIFO
            if (c == String [Offset]) { // Is C a part of the string?
                Offset++; // Compare with next character
                if (String [Offset] == 0) { // is this the end of string?
                    return True; // match = True
                }
            } else { // c does not belong to String
                Offset = 0; // Reset String pointer
            }
        }
    }
}
/**
 * Function : ProcModemReceive
 * Parameters : A character received from the SCI
 * Date : November 2000
 * Desc : Stores incoming characters in the Modem Queue
 * *********************************************************************************/
 void ProcModemReceive (BYTE c) {
     ModemBuffer [mEmptySlot++] = c;
     if (mEmptySlot > MODEM_BUFFER_SIZE) {
         mEmptySlot = 0;
     }
 }

/**
 * Function : ModemGetch
 * Parameters : None
 * Date : November 2000
 * Desc : Dequeue a previously stored character in the Modem Queue.
 * Returns a null character if the Queue is empty
 * *********************************************************************************/
 BYTE ModemGetch (void) {
     BYTE c = 0;
     if (mDataSlot != mEmptySlot) {
         c = ModemBuffer [mDataSlot];
         mDataSlot++;
         if (mDataSlot > MODEM_BUFFER_SIZE) mDataSlot = 0;
         return(c);
     } else {
         return (BYTE)0x00;
     }
 }

/**
 * Function : transmit
 * Parameters : A string to transmit to the Modem
 * Date : November 2000
 * Desc : Any data passed to this function will be sended to the Modem.
 * Applications can build complex scripts by calling transmit and

AN2120
Waifor functions however, its up to the application to control the appropriate flow of data when the Modem is on command mode and on-line mode.

*******************************************************************************/
void transmit (char *data) {
    Delay (250);
    while (*data) { 
        WriteComm (*data++);
    }
}

IP.C
Internet Protocol Implementation
/*******************************************************************************
File Name : IP.C
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : September 2000
Current Revision : 0.0
Notes : This file contains the Internet Protocol variables & support routines
*******************************************************************************/
#include "IP.h"
#include "PPP.h"
#include "SLIP.h"
extern BYTE InBuffer [PPP_BUFFER_SIZE + 1]; // Input Buffer for PPP data
extern BYTE OutBuffer[PPP_BUFFER_SIZE + 1];
BYTE IPAddress[4] = {220, 1, 141, 149}; // Default IP Address
static volatile char IPAdapter = PPP; // Default interface for IP output
IPDatagram *ip_in; // A pointer to received IP datagrams
IPDatagram *ip_out; // Global buffer for IP packet output

/*******************************************************************************
Function : IPInit
Parameters : None
Date : September 2000
Desc : Initializes the IP module pointers
*******************************************************************************
void IPInit (void) {
  ip_in = (IPDatagram *)&InBuffer [4];
  ip_out = (IPDatagram *)&OutBuffer [4];
}

Function : Bind adapter
Parameters : Interface: A Byte ID
Date : September 2000
Desc : Selects the output format of an IP packet

void IPBindAdapter (INTERFACE Interface) {
  IPAdapter = Interface; // switch to different output interface
}

Function : IPNetSend
Parameters : ip: A pointer to a IP datagram to transmit
Date : November 2000
Desc : Sends a IP datagram over the interface specified

void IPNetSend (IPDatagram* ip) {
  static WORD Id = 0xF0; // ID to be used in IP datagrams
  ip_out->Version_HLen = 0x45; // Header Forma=IPv4, Length = 5
  ip_out->Service = 0; // Always zero
  ip_out->LengthUpper = 0; // High byte of datagram Length
  ip_out->ID = htons(Id++); // Merge IP ID
  ip_out->Frag = 0; // No flags nor enable fragmentation
  ip_out->TTL = 0x80; // Time to live set to default
  ip_out->Checksum = 0; // Clear checksum to avoid
    // miscalculations
  // Get checksum of entire datagram
  ip_out->Checksum = htons(IPCheckSum ((BYTE *)ip_out, 10));
  switch (IPAdapter) {
    case PPP: // Output through PPP adapter
      OutBuffer [0] = 0xff; // Frame PPP packet
      OutBuffer [1] = 0x03;
      OutBuffer [2] = 0x00; // This is a IP datagram, set
        // protocol type
      OutBuffer [3] = 0x21;
  }
}
ProcPPPSend (OutBuffer, OutBuffer [7] + 6);

break;

case SLIP:  // Output through SLIP interface
    ProcSLIPSend ((BYTE *)ip_out, ip_out->Length);
    break;

case ETHERNET:  // Send datagram over ethernet
    break;

default:
    break;
}

/**************************************************************/
// Function :  IPCompare
// Parameters :  Ip: A pointer to a IP address to compare
// Date :  November 2000
// Desc :  Compares an IP address to the default IP address defined in this module
/**************************************************************/
BYTE IPCompare (BYTE *IPOne) {
    if (IPOne [0] != IPAddress[0]) return (BYTE)0x00;
    if (IPOne [1] != IPAddress[1]) return (BYTE)0x00;
    if (IPOne [2] != IPAddress[2]) return (BYTE)0x00;
    if (IPOne [3] != IPAddress[3]) return (BYTE)0x00;
    return (BYTE) 0x01;
}

/**************************************************************/
// Function :  IPChecksum
// Parameters :  Data: A pointer to an array of Words
// Size: Size of the array
// Date :  August 2000
// Desc :  Obtains the IP checksum of an array of 16-bit words of size "Size"
/**************************************************************/
DWORD IPCheckSum (BYTE* Data, WORD Size) {
    unsigned long Sum = 0;
    while (Size-->0) {
        Sum += ((unsigned long)((*Data++) & 0xFFFF);
    }
    return Sum;
}
Data+=2;
}
Sum = (Sum >> 16) + (Sum & 0xFFFF);
Sum += (Sum >> 16);

UDP.C
User Datagram Protocol Implementation

/**************************************************************************
File Name : UDP.c
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : December 2001
Current Revision : 0.0
Notes : This file contains the code to handle and create UDP transport packets.

/**************************************************************************/
#include "IP.h"
#include "UDP.h"
#include "Ppp.h"
#define UDP_HEADER_LENGTH 8
static WORD UDPLocalPort = 1080; // Default UDP port (can be set to anything)
static void UDPDefaultCallBack (BYTE *data, BYTE size, DWORD RemoteIP, WORD Port);
static UDPCALLBACK UDPCallback = UDPDefaultCallBack;
UDPDatagram *udp_in; // Pointer to incoming UDP packet
UDPDatagram *udp_out; // Pointer for output UDP packet

/**************************************************************************
Function : UDPSetCallbackProc
Parameters : Proc: A pointer to a function to call each time a UDP/IP packet is received from the Internet
Date : December 2000
Desc : Sets the callback function to call each time a UDP packet is received over the physical interface
AN2120
void UDPSetCALLBACK (UDPCALLBACK Proc) {
    DisableInterrupts;
    UDPCallback = Proc;
    EnableInterrupts;
}

Function : UDPDefaultCallback
Parameters : None
Date : December 2000
Desc : The default callback available after RESET not accessible from outside this module

static void UDPDefaultCallback (BYTE *data, BYTE size, DWORD RemoteIP, WORD Port) {
}

Function : UDPBind
Parameters : Port: local port to use in UDP packets to transmit
Date : November 2000
Desc : Specifies the local port to use for sending UDP packets over IP

void UDPBind (WORD Port) {
    UDPLocalPort = Port; // Set source UDP port
}

Function : UDP_Checksum
Parameters : udp: A pointer to the start of a udp/ip packet (0x45)
Date : November 2000
Desc : Calculates the pseudo-header checksum of a UDP packet

WORD UDP_Checksum (BYTE* udp) {
    DWORD Checksum = 0;
}
Checksum = IPCheckSum (&udp[12], (8 + udp[25]) >> 1);
Checksum = ~Checksum + 0x11;
Checksum += udp[25];
Checksum = (Checksum >> 16) + (Checksum & 0xFFFF);
Checksum += (Checksum >> 16);
return (WORD)~Checksum;

/***************************************************************
Function : UDPHandler
Parameters :
  udp: a pointer to the udp (struct UDPPacket) packet received
Date :
Desc :
  Invokes the callback proc so the application can handle the
  UDP data received
***************************************************************/
void UDPHandler (UDPPacket *udp) {
  udp_in = udp;
  udp_in->Payload[udp_in->Length - UDP_HEADER_LENGTH] = 0x00;
  UDPCallback ( // Invoque the CALLBACK function
    (BYTE *)udp_in->Payload,
    udp_in->Length - UDP_HEADER_LENGTH,
    *((DWORD *)&udp_in->SourceIP),
    udp_in->DestPort);
}

/***************************************************************
Function : UDPSendData
Parameters :
  BYTE Ip[]: The IP address of the remote host
  Port: UDP port of the remote host
  Payload: Data to send
  Size: Number of bytes to send to remote host
Date :
Desc :
  Sends data (payload) over UDP to a remote host specified by IP[] using
  Port as the destination UDP port.
***************************************************************/
void UDPSendData (BYTE Ip[], WORD Port, BYTE* Payload, BYTE size) {
  WORD Checksum = 0;
  ip_out->DestAddress[0] = Ip[0]; // Store source and destination
  ip_out->DestAddress[1] = Ip[1]; // IP addresses
  ip_out->DestAddress[2] = Ip[2];
  ip_out->DestAddress[3] = Ip[3];
ip_out->SourceAddress [0] = IPAddress [0];
ip_out->SourceAddress [1] = IPAddress [1];
ip_out->SourceAddress [2] = IPAddress [2];
ip_out->SourceAddress [3] = IPAddress [3];

udp_out = (UDPDatagram *) &ip_out->SourceAddress;

// Insert Data Payload if available as an argument
if (Payload)
    Move (Payload, &udp_out->Payload[0], size);

// Format payload as a null terminated string
udp_out->Payload[size] = 0x00;
if (size % 2) { // Pad the payload
    size++;
 }

udp_out->Length = size + UDP_HEADER_LENGTH;     // Calculate the UDP length
ip_out->Length = size + UDP_HEADER_LENGTH + 20;// get IP packet length
ip_out->Protocol = UDP; // Protocol set to UDP

udp_out->SourcePort = htons(UDPLocalPort);// Set source and destination ports
udp_out->DestPort = htons(Port);
udp_out->LengthUpper = 0; // Packet cannot be longer than 256
                          // bytes
                          // (in this implementation)
udp_out->Checksum = 0; // Set checksum to 0
Checksum = UDP_Checksum ((BYTE *)ip_out); // Obtain the packet checksum
udp_out->Checksum = htons (Checksum);

IPNetSend (ip_out); // Send the packet to the IP layer

---

ICMP.C
Internet Control Message Protocol Module Implementation

/**************************************************************************
File Name : ICMP.c
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : January 2001
Current Revision : 0.0
Notes : This file contains the code to handle and create ICMP messages

**************************************************************************/
#include "IP.h"
#include "ICMP.h"

/**
 Function : ICMPPing
 Parameters : IP Address to ping
 Date : September 2000
 Desc : Sends a ICMP ECHO message to a remote host
 **/

void IcmpPing (BYTE Ip[]) {
    WORD Value;
    static BYTE Seq = 0xAB;

    ip_out->SourceAddress [0] = IPAddress [0]; // Ping will have our source address
    ip_out->SourceAddress [1] = IPAddress [1];
    ip_out->SourceAddress [2] = IPAddress [2];
    ip_out->SourceAddress [3] = IPAddress [3];

    ip_out->DestAddress [0] = Ip[0]; // Set destination IP address
    ip_out->DestAddress [1] = Ip[1];
    ip_out->DestAddress [2] = Ip[2];
    ip_out->DestAddress [3] = Ip[3];

    ip_out->Payload [0] = ECHO; // ICMP message type set to ECHO
    ip_out->Payload [1] = 0; // ICMP code must by set to zero
    ip_out->Payload [2] = 0; // reset checksum
    ip_out->Payload [3] = 0;

    ip_out->Payload [4] = 1; // set ID of ICMP message
    ip_out->Payload [5] = 0;
    Seq++;
    ip_out->Payload [6] = (Seq >> 8) & 0xFF; // set sequence number of ICMP Msg
    ip_out->Payload [7] = Seq & 0xFF;

    ip_out->Protocol = ICMP; // IP datagram will carry ICMP data
    ip_out->Length = 28; // ECHO message doesn't include data

    Value = IPCheckSum ((BYTE *)&ip_out->Payload[0], (ip_out->Length - 20) >> 1);
    ip_out->Payload [2] = (Value >> 8); // obtain ICMP checksum
    ip_out->Payload [3] = (Value & 0xFF);

    IPNetSend (ip_out); // Net send to IP layer
}
/**********************************************
Function : ICMP_Handler
Parameters : IP Datagram containing ICMP data
Date : September 2000
Desc : Handles incoming IP datagrams according to the TYPE field of the ICMP message contained in the input IP datagram
***********************************************/
void IcmpHandler (IPDatagram* ip) {
WORD Value;

switch (ip->Payload[0]) {
    case ECHO:
        /* Move ping datagram to output buffer */
        Move ((BYTE *)ip, (BYTE *)ip_out, ip->Length);

        /* Swap source and destination IP addresses on Output Buffer */
        ip_out->DestAddress[0] = ip->SourceAddress[0];
        ip_out->DestAddress[1] = ip->SourceAddress[1];
        ip_out->DestAddress[2] = ip->SourceAddress[2];
        ip_out->DestAddress[3] = ip->SourceAddress[3];

        ip_out->SourceAddress[0] = ip->DestAddress[0];
        ip_out->SourceAddress[1] = ip->DestAddress[1];
        ip_out->SourceAddress[2] = ip->DestAddress[2];
        ip_out->SourceAddress[3] = ip->DestAddress[3];

        ip_out->Payload[0] = ECHO_REPLY; /* Echo reply */
        ip_out->Payload[1] = 0; /* Set ICMP Code to 0 */
        ip_out->Payload[2] = 0; /* Set ICMP checksum to 0 during checksum generation */
        ip_out->Payload[3] = 0;
        Value = IPCheckSum ((BYTE *)&ip_out->Payload[0], (ip->Length - 20) >> 1); /* Calculate ICMP checksum */
        ip_out->Payload[2] = (Value >> 8); /* Set ICMP checksum */
        ip_out->Payload[3] = (Value & 0xFF);
        IPNetSend (ip_out); /* Send ICMP packet over IP */
        break;

    case ECHO_REPLY:
        // Code to handle ping responses
        // goes here
        NoOperation;
        break;

    case TRACEROUTE:
        break;

    default:
        break;
}
}
PLL.C
Code of InitPLL Function

/**************************************************************************/
File Name: Pll.c
Author: Rene Trenado
Location: Freescale Applications Lab, Baja California
Date Created: September 2000
Current Revision: 0.0
Notes: This file contains the code of the InitPll function
/**************************************************************************/
#include "_pll.h"

/**************************************************************************/
Function: InitPll
Parameters: None
Date: September 2000
Desc: Initializes the PLL to operate at 4.91520 MHz
**************************************************************************/
#asm
"_InitPLL"
_InitPLL:
  BCLR 5,0x36 ; turn off PLL so it can be initialized
  MOV #0x00,0x38 ; Set multiplier for 4.9152MHz
  MOV #0x96,0x39 ; see manual for calculations
  MOV #0x80,0x3A ; Set range select
  BSET 7,0x37 ; Allow automatic acquisition & tracking
  BSET 5,0x36 ; turn PLL back on
HERE:
  BRCLR 6,0x37,HERE ; Wait for PLL to lock
  BSET 4,0x36 ; Select PLL as Source
#endasm
**Delay.C**

**Source Code of Variable Delay() Function**

/*************************************************************************
File Name : Delay.c

Author : Rene Trenado

Location : Freescale Applications Lab, Baja California

Date Created : July 2000

Current Revision : 0.0

Notes : This file contains the code for a variable Delay function
**************************************************************************/
#include "delay.h"

BYTE delayCounter;

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

Function : Delay

Parameters : A Byte containing the number of times _Delay will be called

Date : July 2000

Desc : This function blocks the CPU in multiples of _Delay times

/***************************************************************************/
void Delay (register BYTE times) {
   _Delay();
}

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

Assembly Function : __1msDelay

Parameters : None

Date : July 2000

Desc : This function blocks the CPU in multiples of 1.3mSecs
delayCount specifies the time base
   __1msDelay = delayCounter x 1.3 mSec

/***************************************************************************/
#asm
   xref.b _delayCounter
   xdef __Delay
BUSFREQ: EQU 2

_Delay:
PSHA ;2 cycles
LDA #BUSFREQ ;2
DBNZ DLoop ;3
BRA DLSub ;3
DLSub:
MOV #$FF,_delayCounter ;4
Here:
DBNZ _delayCounter,Here ;5
BRA DLoop ;3
DLoop:
PULA ;2
RTS ;4

/*******************************************************************************
Function : _Delay
Parameters : A Byte containing the number of times a base delay will be called
Date : July 2000
Desc : This function blocks the CPU in multiples (Acc value) of delay times
*******************************************************************************/
_Delay:
JSR _1msDelay
DBNZA _Delay
RTS

CommDrv.H
Header File for SCI Driver

/*******************************************************************************
File Name : CommDrv.h
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : July 2000
Current Revision : 0.0
*******************************************************************************/
Application Note

Notes : This file contains comm port specific definitions

/////////////////////////////////////////////////////////////////////////////////////////
#ifndef _H_COMMDRV_
define _H_COMMDRV_

#include "Notation.h"
#define BAUDS_2400 3 // 8 Divisor
#define BAUDS_4800 2 // 4 Divisor
#define BAUDS_9600 1 // 2 Divisor
#define BAUDS_19200 0 // 1 Divisor
typedef enum {
    ENABLE_RX = 0x04, // enable receiver
    ENABLE_TX = 0x08, // enable transmitter
    ENABLE_RX_EVENTS = 0x20, // enable receiver interrupts
    ENABLE_TX_EVENTS = 0x80 // enable transmitter interrupts
} CommOptions;
#define SCC1 *((BYTE *)0x13) // Status and contro registers
#define SCC2 *((BYTE *)0x14)
define SCS1 *((volatile BYTE *)0x16)
define SCDR *((volatile BYTE *)0x18)
define SCBR *((BYTE *)0x19)
extern void @interrupt UartRxISR (void);         // export ISR

////////////////////////////////////////////////////////////////////////////////////////

#endif

PPP.H
Header File for PPP Implementation

/***************************************************************************/
File Name : PPP.h
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : September 2000

For More Information On This Product,
Go to: www.freescale.com
Current Revision : 0.0

Notes : Definitions for the PPP implementation

////////////////////////////////////////////////////////////////////////////*/
#ifndef __PPP_H
#define __PPP_H 1
#include "Notation.h"
#endif

#ifndef NULL
#define NULL 0
#endif

#define ESC 0x7D
#define END 0x7E
#define REQ 1
#define ACK 2
#define NAK 3
#define REJ 4
#define TERMINATE 5
#define TERMINATE_ACK 6

typedef struct {
    WORD Framing;
    WORD Protocol;
    BYTE Request;
    BYTE Id;
    BYTE LengthHigh;
    BYTE Length;
    BYTE FirstOption;
    BYTE FirstOptionLength;
    BYTE Param;
    BYTE Data;
} PPPFrame;

#define PPPINITFCS16    0xffff  /* Initial FCS value */
#define PPPGOODFCS16    0xf0b8  /* Good final FCS value */

////////////// Functions to Export /////////
void PPPInit (void);
BYTE *PPPGetInputBuffer (void);
BYTE *PPPGetOutputBuffer (void);
void ProcPPPReceive (register BYTE c);
void ProcPPPSend (BYTE *Buffer, BYTE len);
WORD PPPGetChecksum (register unsigned char *cp, register int len);
void SendPAPPacket (BYTE Action, BYTE ID, char* user, char* password);
void Move (BYTE *src, BYTE *dest, register numBYTEs);
void PPPEntry (void);
void PPPTerminate (void);
void PPPSendVoidLCP (void);
extern volatile BYTE PPPStatus;
```c
#define IsESC 0x01 // Previous character received was a ESC char
#define ReSync 0x04 // Re Synchronize to avoid incomplete IP frame reception
#define IsFrame 0x08 // A full packet
#define ByteRx 0x10 // Receive a Byte
#define LinkOn 0x20 // PPP Link is On

extern BYTE IPAddress[4];

#define PPP_BUFFER_SIZE 88

#define LCP_PACKET 0xC021
#define PAP_PACKET 0xC023
#define CHAP_PACKET 0xC223
#define IPCP_PACKET 0x8021
#define IP_DATAGRAM 0x0021

#endif
```

---

**SLIP.H**

*Header File for SLIP Implementation*

```c
/*////////////////////////////////////////////////////////////////////////////
File Name : SLIP.h
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : June 2000
Current Revision : 0.0
Notes : Definitions for the SLIP implementation

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

#ifndef __SLIP_H
#define __SLIP_H

#include "Notation.h"

#ifndef NULL
#define NULL 0
#endif

#define SLIP_MAX_SIZE 88
#define SLIP_END 0xC0 //300 octal
#define SLIP_ESC 0xDB //333 octal
#define ESC_END 0xDC //334 octal

#endif
```
#define ESC_ESC 0xDD  // 335 octal

extern BYTE SLIPStatus;

void SLIPInit (void);
void ProcSLIPSend (BYTE *ptr, BYTE len);
void SLIPEntry (void);
void ProcSLIPReceive (BYTE c);

#define IsESC 0x01  // Previous character received was a ESC char
#define ReSync 0x04  // Re Synchronize to avoid incomplete IP frame reception
#define IsFrame 0x08  // A full packet
#define ByteRx 0x10  // Receive a Byte

#endif

ModemDrv.H
Header file for Modem driver

#ifndef __MODEMDRV_H
#define __MODEMDRV_H 1

#include "Notation.h"

void ProcModemReceive (BYTE c);
void ModemBindBuff (BYTE *lpInBuffer);
void ModemInit (void);
BYTE ModemDial (char * Number);
void transmit (char *data);
void ModemHangUp (void);
BYTE ModemOnLine (void);
BYTE ModemBuffNotEmpty (void);
void ModemBuffFlush (void);
BYTE ModemInBufferCount (void);
BYTE Waitfor (char *String, BYTE Time);
BYTE ModemGetch (void);
BYTE ModemInBufferCount (void);
void ModemReset (void);

#endif
IP.H
Internet Protocol Implementation de nitions

/******************************************************************************
File Name : IP.h

Author : Rene Trenado

Location : Freescale Applications Lab, Baja California

Date Created : September 2000

Current Revision : 0.0

Notes : Definitions for the IP implementation

/******************************************************************************/
#ifndef __IP_H
#define __IP_H
#endif

#include "Notation.h"

typedef struct {
    BYTE Version_HLen;
    BYTE Service;
    BYTE LengthUpper;
    BYTE Length;
    WORD ID;
    WORD Frag;
    BYTE TTL;
    BYTE Protocol;
    WORD Checksum;
    BYTE SourceAddress [4];
    BYTE DestAddress [4];
    BYTE Payload [64];
} IPDatagram;

extern IPDatagram *ip_in;
extern IPDatagram *ip_out;

typedef enum { RAW_SERIAL = 1, SLIP, PPP, PARALLEL, ETHERNET } INTERFACE;

#define TCP 0x06
#define UDP 0x11
#define ICMP 0x01

extern BYTE IPAddress[4];

/******************************************************************************/
IP Exported Functions
/*******************************************************************************/
BYTE IPCompare (BYTE *IPOne);
DWORD IPCheckSum (BYTE *Data, WORD Size);

/* For More Information On This Product, Go to: www.freescale.com */
void IPBindAdapter (INTERFACE Interface);
void IPInit (void);
#endif

UDP.H
UDP Header Definitions

/********************************************************************************
File Name : UDP.h
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : December 2001
Current Revision : 0.0
Notes : This file contains definitions needed by the UDP module.
/********************************************************************************/
#ifndef __UDP_H
#define __UDP_H
#include "Notation.h"

typedef struct {
    BYTE SourceIP [4];
    BYTE DestinationIP [4];
    WORD SourcePort;
    WORD DestPort;
    BYTE LengthUpper;
    BYTE Length;
    WORD Checksum;
    BYTE Payload[54];
} UDPDatagram;

typedef void (*UDPCALLBACK)(BYTE *data, BYTE size, DWORD RemoteIP, WORD Port);
void UDPSetCALLBACK (UDPCALLBACK Proc);
void UDP_Handler (UDPDatagram *udp);
WORD UDP_Checksum (BYTE* udp);
void UDPBind (WORD Port);
void UDP_SendData (BYTE Ip[], WORD Port, BYTE* Payload, BYTE size);

#endif
ICMP.H

ICMP Header Definitions

/******************************************************************************
File Name : Icmp.h
Author : Rene Trenado
Location : Freescale Applications Lab, Baja California
Date Created : January 2001
Current Revision : 0.0
Notes : This file contains Icmp module specific definitions
*******************************************************************************/

#ifndef __ICMP_H
#define __ICMP_H

#include "Notation.h"

typedef struct {
    BYTE Type;
    BYTE Code;
    WORD Checksum;
    WORD Identifier;
    WORD SeqNumber;
} ICMPDatagram;

#define ECHO 8
#define ECHO_REPLY 0
#define TRACEROUTE 30

void IcmpHandler (IPDatagram *ip);
void IcmpPing (BYTE Ip[]);

#endif

PLL.h

Header Definitions for the PLL.c Module

#ifndef __PLL_H
#define __PLL_H

extern void InitPLL (void);

#endif
Delay.h
Header Definitions for Delay() Function Support

#ifndef __Delay_H
#define __Delay_H

#include "Notation.h"

extern void Delay (register BYTE Time);

#endif

Notation.h
Notation Used in the Source Code

#ifndef __NOTATION_H
#define __NOTATION_H 1

#define BIG_ENDIAN

#if defined(BIG_ENDIAN)
#define htons(A) (A)
#define htonl(A) (A)
#define ntohs(A) (A)
#define ntohl(A) (A)
#elif defined(LITTLE_ENDIAN)
#define htons(A) (((A) & 0xFF00) >> 8) | ((A) & 0x00FF) << 8)
#define htonl(A) (((A) & 0xFF000000) >> 24) | (((A) & 0x00FF0000) >> 8) | (((A) & 0x0000FF00) << 8) | (((A) & 0x000000FF) << 24))
#define ntohs htons
#define ntohl htonl
#else
#error "User Must define LITTLE_ENDIAN or BIG_ENDIAN!!"
#endif

#define DWORD unsigned long
#define BYTE unsigned char
#define WORD unsigned int

Freescale Semiconductor, Inc.
For More Information On This Product, Go to: www.freescale.com
#define False 0
#define True 1

typedef void (*EventProc)(BYTE c);

typedef struct {
    unsigned char b0;
    unsigned char b1;
    unsigned char b2;
    unsigned char b3;
    unsigned char b4;
    unsigned char b5;
    unsigned char b6;
    unsigned char b7;
} TByteBits;

typedef union {
    unsigned char Value;
    TByteBits Bits;
} TByte;

#define AppLoop while(1)

#define EnableInterrupts _asm("CLI\n");  //Enable interrupts
#define DisableInterrupts _asm("SEI\n");  //Enable interrupts
#define NoOperation _asm("NOP\n"); // No operation

CommDrv.C
Serial Communications Interface Driver for the PC

#include <dos.h>
#include "CommDrv.h"

static void CommDrvDefaultProc (BYTE value);

static void (*EvtProcedure) (BYTE value) = CommDrvDefaultProc;

static void interrupt UartISR (void);
static void interrupt (*IsrOriginal)();

static Word Port = COM1;
static Byte IRQMask;


void InitCommDriver (void) {
    EvtProcedure = CommDrvDefaultProc;
}
WORD CommPort (void) {
    return Port;
}

/******************************************************************/
// Assigns an Event Handler for Comm Driver
/******************************************************************/
void CommEventProc (EventProc Proc) {
    disable ();
    EvtProcedure = Proc;
    enable ();
}

/******************************************************************/
// Default Event Handler for Comm Driver
/******************************************************************/
static void CommDrvDefaultProc (BYTE value) {
    (void) value;
}

/******************************************************************/
void OpenComm (Word CommPort, BYTE Bauds) {
    disable ();
    Port = CommPort;
    // Configura el puerto "CommPort" a 9600,n,8,1
    outportb (Port + LCR, LATCH_DIVISOR);
    outportb (Port + DIVISOR_BAJO, Bauds);
    outportb (Port + DIVISOR_ALTO, 0x0);
    outportb (Port + LCR, 0x03);
    outportb (Port + MCR, HABILITA_INT);
    outportb (Port + IER, RX_ENABLE | MODEM_STATUS);
    if (Port == COM1) {
        IsrOriginal = getvect (COM1_ISR);
        setvect (COM1_ISR, UartISR);
    } else {
        IsrOriginal = getvect (COM2_ISR);
        setvect (COM2_ISR, UartISR);
    }
    IRQMask = inportb (PIC_IMR);
    outportb (PIC_IMR, (Port == COM1) ? (IRQMask & 0xEF):(IRQMask & 0xF7));
    enable ();
}

void CloseComm (void) {
    if (!Port) return;
}
outportb (Port + MCR, 0);
outportb (Port + IER, 0);
outportb (PIC_IMR, IRQMask);

if (Port == COM1) {
    setvect (COM1_ISR, IsrOriginal); }
else {
    setvect (COM2_ISR, IsrOriginal); }
}

void WriteComm (Byte c) {
    while (!(inportb(Port + LSR) & 0x20));
    outportb (Port + THR, c);
}

void WriteCommStr (char * string) {
    while (*string) {
        WriteComm (*string++);
    }
}

void interrupt UartISR (void) {

    switch (inportb (Port + IIR) & 0xFE) {
    case 0x00: //Modem Status
        // if a change in CD line
        if (inportb (Port + MSR) & 0x08) {
            outportb (Port + MCR, inportb (Port + MCR) | 0x02); // Set RTS line to high
        }
        else {
            outportb (Port + MCR, inportb (Port + MCR) & ~0x02); // Clear RTS line to high
        }
        break;

    case 0x04: //Rx Char
        EvtProcedure (inportb (Port + RBR));
        break;
    }

    outport (PIC_ICR, 0x20);//Ack this IRQ
CommDrv.H
Serial Communications Interface Definitions for the PC

#ifndef __COMM_H
#define __COMM_H

#include "Notation.h"

#define COM1 0x3F8
#define COM2 0x2F8
#define COM4 0x2E8

#define RBR 0 // Receive Buffer
#define THR 0 // Transmitter Buffer
#define DIVISOR_BAJO 0 // Latch divisor low
#define DIVISOR_ALTO 1 // Latch divisor high
#define IER 1 // Interrupt Enable Register
#define IIR 2 // Interrupt ID Register
#define LCR 3 // Line Control Register
#define MCR 4 // Modem Control Register
#define LSR 5 // Line Status Register
#define MSR 6 // Modem Status Register

#define LATCH_DIVISOR 128
#define HABILITA_INT 8

#define RX_ENABLE 1 // RxRDY Enable IRQ
#define TX_ENABLE 2 // Tx Buffer Empty IRQ
#define MODEM_STATUS 8 // Modem handshake lines have changed

#define PIC_ICR 0x20 // PIC address
#define PIC_IMR 0x21 // PIC IRQ Mask Register
#define COM1_ISR0x0C // COM1 Vector Table Index
#define COM2_ISR0x0B // COM2 Vector Table index
#define COM4_ISR0x08 + 9 // COM2 Vector Table index

#define ASCII 0
#define BINARY 1

//////////////////////////////// Functions to Export //////////////////////
void InitCommDriver (void);
void OpenComm (Word CommPort, BYTE Bauds);
void OpenComm (Word CommPort, BYTE Bauds);
void CloseComm (void);
void CommEventProc (EventProc Proc);
void WriteComm (Byte c);
void WriteCommStr (char * string);
WORD CommPort (void);

#define BAUDS_2400 0x30
#define BAUDS_4800 0x18
#define BAUDS_9600 0x0C
#define BAUDS_19200 0x06

AN2120
#define BAUDS_38400 0x03
#endif

#define BAUDS_38400 0x03
#endif
How to Reach Us:

Home Page:  
www.freescale.com

E-mail:  
support@freescale.com

USA/Europe or Locations Not Listed:  
Freescale Semiconductor  
Technical Information Center, CH370  
1300 N. Alma School Road  
Chandler, Arizona 85224  
+1-800-521-6274 or +1-480-768-2130  
support@freescale.com

Europe, Middle East, and Africa:  
Freescale Halbleiter Deutschland GmbH  
Technical Information Center  
Schatzburger 7  
81829 Munchen, Germany  
+44 1296 380 456 (English)  
+46 8 5220080 (English)  
+49 89 92103 559 (German)  
+33 1 69 35 48 48 (French)  
support@freescale.com

Japan:  
Freescale Semiconductor Japan Ltd.  
Headquarters  
ARCO Tower 15F  
1-8-1, Shimo-Meguro, Meguro-ku,  
Tokyo 153-0064  
Japan  
0120 191014 or +81 3 5437 9125  
support.japan@freescale.com

Asia/Pacific:  
Freescale Semiconductor Hong Kong Ltd.  
Technical Information Center  
2 Dai King Street  
Tai Po Industrial Estate  
Tai Po, N.T., Hong Kong  
+800 2666 8080  
support.asia@freescale.com

For Literature Requests Only:  
Freescale Semiconductor Literature Distribution Center  
P.O. Box 5405  
Denver, Colorado 80217  
1-800-441-2447 or 303-675-2140  
Fax: 303-675-2150  
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document. Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typical" must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.