An Embedded Web Server for National Semiconductor's CR16MCS9 *CannonBall*

What datagrams may come…

Abstract:

These days it seems, everybody and their brother is talking about the need of becoming "Internet aware" - a new catch phrase sounding eerily similar to something said in Eden some time ago. The explosive growth and appeal of the Internet has everyone scrambling to get onboard, or be thought of as somehow "20th century". Today, Internet accessibility in one form or another, if not an a priori requirement, is at least a highly desirable option in many embedded applications. Previously the sole domain of mainframes, PCs, and workstations, TCP/IP stacks and other networking applications are now being written by the dozens for embedded microprocessors and microcontrollers, providing them the smarts to hook into the "matrix". This brief article will examine one such TCP/IP stack and Web server implemented on National Semiconductor's *CannonBall* RISC microcontroller. We'll also seek to resonate some understanding of the basic issues one needs to consider when deciding on which approach is best suited for their embedded Internet application.

Introduction:

The rapid advances in semiconductor technology throughout the last decade of the 20th century have enabled the development of powerful new microprocessors and microcontrollers, bringing the embedded world computational power previously found only in mainframes, supercomputers, and idiot savants like the Rainman. RISC architectures, previously untenable in a cost sensitive embedded word (due to their thirst for expensive memory), have now been adopted as somewhat standard fare. National Semiconductor's *CompactRISC* is one such architecture, having at once both the power to run the Networking software necessary to connect your embedded application to the Internet, and the memory efficiency still requisite in the embedded world.

(It's a little known fact that the Internet was originally developed by a bunch of professors who were hoping to make the arduous job of homework grading easier. Professors and their Teaching Assistants (TA's) were in desperate need of a system to allow them to quickly and easily exchange their student's homework assignments. If Professor Z (hailing from Kokomo) could wire his or her TA *ungraded* homework (on vacation in Las Vegas), well then the Professor could spend more of his/her valuable time writing obscure and arcane papers to secure tenure (read "early retirement"). While some might argue that this is a bunch of hooey, that's the way *I* read it.)

Discussion:

Among the myriad of embedded Internet solutions being touted today, all fall neatly into one of five fundamental groups:

1. Embedding a fully functional (or nearly so), third party TCP/IP stack into your application, enabling direct Internet access…
2. Using a third party's external TCP/IP gateway device, such as *NetSilicon's Net+ARM™ solutions*…
3. Writing your own TCP/IP stack, or some functional subset thereof…
4. Using your own, or a third party's, "lightweight", proprietary communication protocol to talk with an external Gateway device, which is itself connected to the Internet (e.g. EmWare)…
5. Everything else – for those which don't fall that neatly.

The decision as to which strategy to adopt will generally lie along the vector sum of the two orthogonal vectors of **price** and **performance**. That is, increasing levels of Internet "interoperability" will generally come with increased cost. Simply put, what Internet communication capabilities must your application possess - and just how much are you willing to pay for it.

At one end of the scale lay both the full-featured stacks offered by many RTOS vendors and the external gateway devices offered by the likes of *NetSilicon* and *Seiko*. Applications requiring fully RFC-compliant Internet communication capabilities, or those needing to provide an industry standard API (whatever that means), may opt for either of these solutions. If your application's resources (i.e. RAM, ROM, and HP - *horsepower*) are limitless (or there abouts), licensing networking software from a third party is an excellent choice. Alternatively, if you still need a high level of Internet interoperability but lack the onboard resources to embed one of these stacks, the external gateway solutions like Seiko's S-7600A and NetSilicon's Net+ARM™ are attractive. Both of these options offload the engineer the onerous task of becoming intimately familiar with the multifaceted and sundry networking issues he or she would otherwise be required to master. Naturally, all this service comes at no mean price.

However, a subtle fact often left unconsidered when evaluating these third party stacks is that, many of the features which they include to ensure RFC compliance - *are simply not required* in many embedded environments. Simply put, while seeking to closely comply with the spirit of the RFC's, many third party stacks are prohibitively large for many typical embedded environments, requiring more code and data space than many microcontrollers can afford. What is more, a close examination of the capabilities of some of the external devices, Seiko's S-7600A TCP/IP device for example, reveals many areas on *noncompliance* with the specifications! For example, despite the fact that all IP's are *required* to support *fragmented* datagrams, Seiko's device does not.

Conversely, at the bottom end of this scale lay the many applications lacking the resources, budget, or the *need* to speak TCP, but nonetheless need some limited form of connectivity. These applications would be well served by options 3 or 4 – writing your own functional *subset* of TCP/IP, or adopting an integrated, proprietary, "lightweight", non-TCP protocol. *EmWare* offers such a solution. *EmWare's* approach is at once adequate and appropriate for many of these low-end applications. The only real drawbacks to such an approach are:

1. Similar to the full-blown stacks, you must license the non-standard, proprietary communication kernel…
2. Direct Internet connectivity is not possible. Your application accesses the Internet via a Gateway device (PC).

Nevertheless, for many low-end applications these may not be show-stopping issues. Often low-end applications require only the intermittent transfer of small amounts of data, and have no need to speak directly with an Internet host or router. (Actually, UDP - the User Datagram Protocol was developed for just this scenario. UDP is used by many networking applications needing only to transmit periodic datagrams containing requests for service, or answers to these requests. For example, most DNS requests issued by browsers use UDP.)

Finally, for those applications whose requirements lay somewhere between these two extremes, a suitable compromise may be that of writing your own networking software. What is needed is an acceptable subset of the standard TCP/IP capabilities - affording the application direct Internet connectivity, while avoiding the costs associated with most of the high-end solutions. Make no mistake; this too comes at a price – and one that many engineers may not wish to countenance.

The task of writing networking software is by no means trivial (at least with those with nominal IQ's, like that of the author's), and is fraught with numerous crevasses and pitfalls. The learning curve for those unfamiliar with the subject can be considerable. However, it is the author's opinion that such investments return significant long-term benefits for the companies and their engineers whose products are evolving to include networking facilities. Like any other area of technology, one cannot consistently and successfully apply it, without having at least *some* level of expertise in that area. While "time-to-market" is certainly an important and even *critical* consideration, if you wish to be around in the long run, you'll need to equip yourself for the journey (i.e. acquire the necessary knowledge).

To that end, let us begin with a brief overview of this voluminous subject…

I. TCP Backgrounder:

The primary purpose of any transport protocol is to provide a "…reliable, securable, logical (i.e. *virtual*) connection between pairs of processes". As per RFC 1122, TCP is the primary *virtual-circuit* transport protocol for the Internet suite." By "virtual-circuit", what is meant is that, although TCP establishes what appears to be an actual *circuit-switched* or, physical connection (just like the one you make when you phone in your pizza order), TCP is actually a *packet-switched* protocol. Unlike the direct point-to-point circuit established between a pair of telephones when placing a call, each packet in a packet switched protocol may be routed through *different* circuits, or paths, in reaching its destination. In a packet switched protocol, every packet contains a source and destination *address*. This enables the *dynamic* routing of packets. As circuits become available or unavailable, the several packets of any single message may be routed through the Internet using many *different* paths before reaching their final destination.

TCP is used by applications requiring a reliable, connection-oriented transport service, such as Web browsers (HTTP), electronic mail (SMTP/POP), and file transfer programs (FTP). What does all that mean? Well, as per the venerable RFC, providing this *Quality of Service* (QoS) over an unreliable network requires facilities in the following areas:

- **Basic Data Transfer:**

TCP manages the transfer of data between peers by encapsulating the data into *segments*, which are then carried in IP *datagrams* through the Internet. TCP attaches a header as shown in Figure 1 to each segment, carrying parameters necessary for addressing, flow-control, and other important functions.

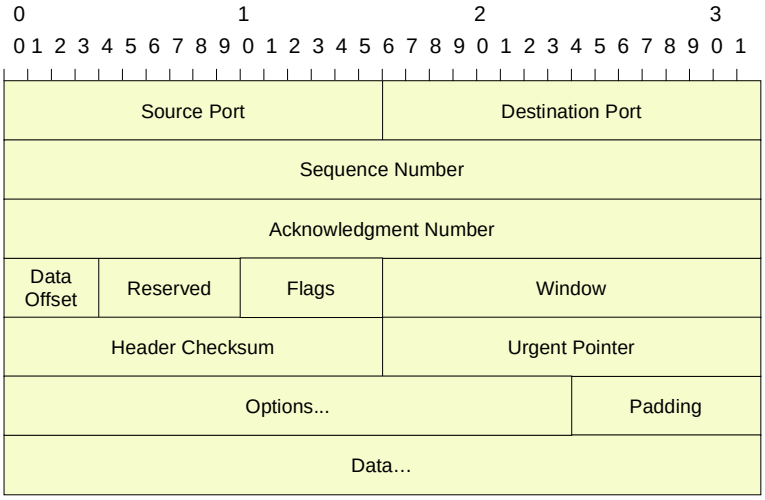

Figure 1. TCP Header Format

- **Reliability:**
  TCP includes mechanisms to recover data that has been damaged, lost, duplicated, or received out of order. These mechanisms include:

  1. Assigning a number to each byte transmitted (the *sequence* number), and requiring an acknowledgement (or, "ACK") from the receiving TCP for all bytes sent. If such an acknowledgment is not received within a predefined timeout interval, the data is retransmitted. At the receiver, these sequence numbers are used to reconstruct the original data.

     It is possible for segments to be received out of order, should they be routed through paths having unequal transit times. In addition, consequent to the varying and potentially unequal delays incurred by different segments, a transmitting TCP may not receive a timely acknowledgement should a segment be unduly delayed. In that case, the transmitter would resend this segment - *incorrectly* inferring that it had been either lost or damaged, resulting in the reception of duplicated segments by the receiving TCP.

     In both of these cases, the segments' sequence numbers help ensure that the data reconstructed by the receiving TCP exactly matches that originally sent. Segments received out of order are correctly reordered, and duplicate segments are discarded.

2. Including a checksum for each segment transmitted. This checksum must be confirmed by the receiving TCP. Should a segment's checksum fail, it is not acknowledged. In this case, the sending TCP will eventually resend this segment.

- **Flow Control:**
  TCP utilizes a method of flow control called a *window*. The window is a 16-bit value transmitted in every segment header indicating the maximum number of bytes that the sender may transmit before receiving further permission. More on this later…

- **Multiplexing:**
  In a typical *host* (all systems using TCP/IP attached to the Internet - except *routers*), multiple resident applications (e.g. a Web browser, and an e-mail client) may simultaneously utilize TCP's services. Within each host, each application is assigned a *port* number, thereafter used by TCP as a "handle" to identify the application. Using this port number, TCP is able to determine which segment goes to which application.

- **Connections:**
  Not to be confused with the physical, point-to-point connection mentioned earlier, TCP is a *connection-oriented* protocol. That is, in order to achieve the reliability and implement the flow-control mechanism mentioned above, TCP must establish, manage, and maintain a *connection* between the two peers exchanging data. A *connection* is defined as a pair of *sockets*, and a socket is defined as the concatenation of the application's *port* number with its host's IP address. These data, plus each TCP's sequence numbers, window sizes, etc., specifies the connection. When two hosts wish to communicate, their respective TCP's first establish a connection (initialize the status information on each side). When their communications are complete, the connection is terminated, or *closed*, to free the resources for other uses.

- **Precedence and Security:**
  TCP includes features that allow applications to indicate certain levels of security and precedence for their communications. Default precedence and security values are required to be used when these features are not explicitly indicated (which is most of the time).

TCP and Window Management

Though initially perhaps somewhat confusing, TCP's use of "*the Window*", and just how this Window is able to *slide* is one of TCP's fundamental concepts. Essentially, TCP's Window is a means of flow control, somewhat analogous to the XON/XOFF mechanism used in asynchronous serial links. However, TCP's Window augments this basic flow control mechanism by including means to maximize the *efficiency* of the communication channel. In the TCP context, *efficiency* is defined as the *maximum* potential data flow between peers - in the shortest possible time. That is, the transmission of data in a manner that utilizes the least amount of network traffic.

Every TCP segment sent out over a network contains a dynamic Window value in the header whose purpose is to inform the other end of the connection just how much data it is currently prepared to accept. At first glance, this may seem a little redundant since during the SYN process each TCP explicitly or implicitly advertises its Maximum Segment Size (MSS). Once

the other end knows the maximum number of bytes it can transmit in a segment, why does it require yet another parameter called a Window? The answer is quite simple: the MSS value advertised during SYN is usually totally *unrelated* to the buffer capacity of the receiving Application. That is, the MSS value stated during SYN is governed by the underlying Link layer's maximum *Frame size*. Ethernet frames, for example, are limited to 1500 bytes. Consequently, a TCP sitting on top of an Ethernet would likely advertise a MSS of no greater than about 1460 bytes (to account for lower layer header overhead). Nevertheless, the Application's receive buffers may be larger than Ethernet's maximum frame size, and as a consequence, may be capable of receiving more than one frame at a time. This is desirable in that it reduces the number of ACK packets the receiver must send, again improving network efficiency. In this case, the sender may send several segments without waiting for a confirmatory "ACK" after each segment.

Since the delays encountered by datagrams traversing the internet are highly variable, requiring a transmitter to wait for the peer to ACK every segment before sending another would result in a great deal of wasted time. The judicious use of the Window helps minimize such waste by allowing the transmitter to send as much data as the peer is capable of accepting, without having to wait for an ACK of the individual segments. Certainly, the receiver must still "ACK" every segment, but this can be done in an aggregate manner instead of one at a time.

One important consideration for any TCP's Window management scheme is the nefarious *Silly Window Syndrome,* or SWS. Since first encountered by a Professor on acid, it has generated a great deal of press and seems to be a favorite buzzword of many armchair Internet experts. SWS is an unforeseen weakness in a literal, straightforward implementation of the window management scheme as suggested in RFC 793, somehow or other exploited by the original Telnet Application. Subsequent studies led to the development and standardization of both sender and receiver algorithms to preclude it (for those interested, see RFC 1122: 4.2.3.4 and 4.2.3.3).

Simply defined, Silly Window Syndrome is a "…stable pattern of small incremental window movements resulting in extremely poor TCP performance." It occurs when a sending TCP gets fooled into sending only tiny data segments, although both sender and receiver have a large total buffer space available. *SWS can only occur during the transmission of large amounts of data*, and will disappear once the connection goes "quiescent".

II. IP Backgrounder:

The Internet Protocol puts the "IP" in TCP/IP. It is TCP/IP's *Network* protocol. IP comprises two basic functions: *addressing* and *fragmentation*. Just like TCP, IP *encapsulates* its data by prepending it with a header as illustrated in Figure 2.

It's easy to get confused as to just why we need an *IP* address in the first place. If your PC sits on an Ethernet or other LAN (Local Area Network), isn't its MAC address unique? Why not simply use this address instead of requiring yet another one?

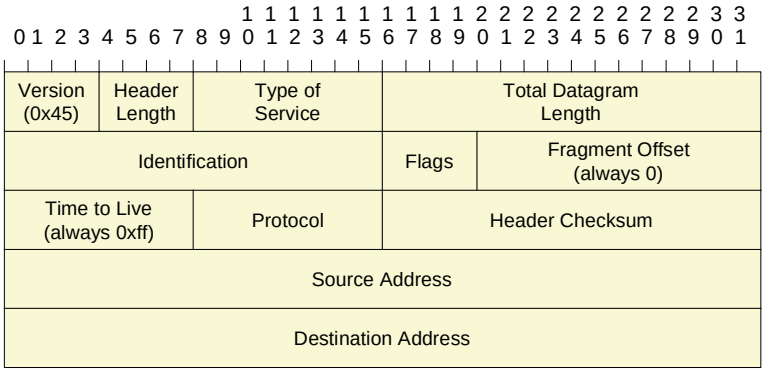

Figure 2. IP Header Format

- **IP addressing**
  The answer is straightforward. Remember that the Internet is not simply one big LAN; rather the Internet is defined as a *network of networks,* or perhaps better stated, a *network of LANs*. If everybody were a node on one great big homogenous network, all running the same Link layer protocol (Ethernet, for example), there would be no need for a separate addressing scheme. The fact is however, that many *disparate* networks exist, all operating incompatible Link layer protocols.

  Every host on a LAN is uniquely identified at the Data Link layer by its Link layer, or *MAC* address (Media Access Control). Neighboring nodes on any given LAN communicate with each other based on this *physical* address. However, a node on an Ethernet cannot *directly* communicate with a node on a Token Ring network – and vice versa. Likewise, nodes speaking ATM, FDDI, etc. are all unintelligible to an Ethernet node. The purpose and design of the Internet Protocol is to allow nodes sitting on these dissimilar LANs to *internetwork*. It does so by *abstracting* their conflicting Link layer protocols, providing a uniform communication interface for all hosts. This permits hosts residing on disparate networks to communicate, even though they may speak a different L^3 (Link layer language).

  This is where the IP address comes in. Whereas every *node* on a LAN is uniquely identified at the Data Link layer by its *MAC* address, each *host* on the Internet is uniquely identified by its *IP address*. IP addresses (i.e. Ipv4) are 32-bit numbers, comprising two subfields: a *network identifier* and a *host identifier* (also referred to as the *netid* and *hostid*). Figure 3 illustrates this hierarchical addressing scheme.

  The *netid* field of the address uniquely identifies a specific LAN, WAN, or other group of linked computers, such as one of the networks shown in Figure 2. The *hostid* field of the address uniquely identifies a host on the addressed network. (Actually, the hostid specifies a

unique *NIC*, or Network Interface Card. An individual computer usually - but not necessarily, has only one such NIC.)

Version 4 of the Internet Protocol (IPv4) has been in use since 1981 and is slowly being supplanted by IPv6. Version 6 improves upon IPv4 in several areas, not the least of which is the extension of IP addresses to 128 bits.

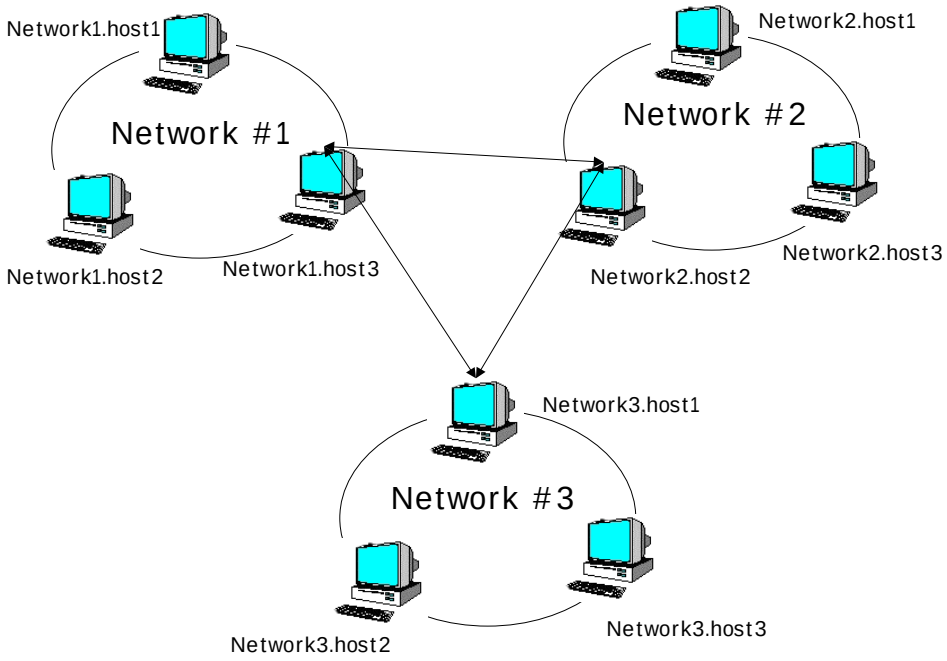

Figure 3 – Ipv4 Hierarchical Addressing scheme

- **IP fragmentation**

  Don't confuse an IP *fragment* with a TCP *segment*; an IP fragment is a piece of a TCP segment whose size precludes it from being transmitted over a network in one piece. The Internet Protocol was designed to be independent of both the underlying Data Link protocol and the overlying Transport protocol. This flexibility is critical because of the large numbers of incompatible Transport and Link layer protocols. However, this independence carries with it certain difficulties, one of which is how to transmit a datagram whose size exceeds that of the underlying Link layer's frame size, also referred to as the Maximum Transmission Unit (MTU).

  To accommodate this eventuality, an IP should be capable of *fragmenting* a segment (received from the overlying transport layer) whose size exceeds this MTU, into multiple datagrams, whose sizes allow them to fit into the *frame* size below it…simple. Not really – in fact, most IP's avoid the nastiness of fragmentation by determining the underlying frame size limitation and reporting that to the transport layer ahead of time by means of what is termed a *path discovery mechanism*. In fact, this is the recommended procedure. However, ALL IP's are

required to be capable of accepting and reassembling incoming fragmented datagrams. IP manages this process by assigning an identification number and fragment offset number to every datagram.

II. Particulars of this Implementation

The TCP/IP stack and embedded Web server described herein consumes less than 20K of code space and requires approximately 2.5K of RAM. The stack operates under uC/OS-II; an RTOS recently ported for use with the CR16B RISC core, chosen for its small kernel size and minimum RAM usage. Each layer is implemented as an independent task. Rather than implementing the "call-return" mechanism used by many stacks, a shared data structure scheme is used. These structures comprise each layer's "API", or interface. Protocol layers are scheduled by the OS according to each layer's priority, and the layers themselves may cause other layers to run to improve efficiency.

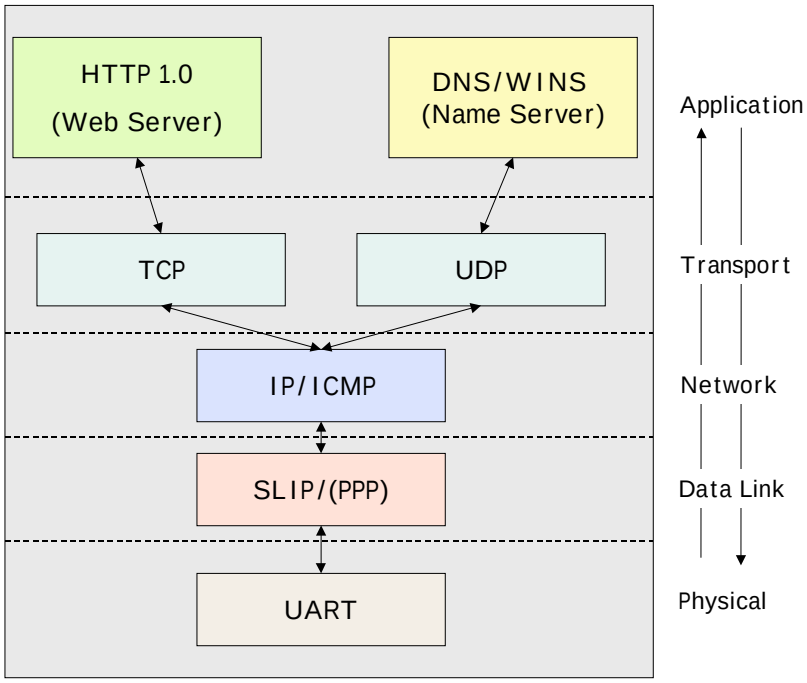

Figure 4 – Protocol Layer Model

You may note in Figure 5, illustrating the sequencing of the various layers during a typical HTTP request, that these priorities somewhat follow the progression of a typical segment as it proceeds up and down the protocol stack. Task priorities are indicated by the circled number in the upper left corner of each task box. Each layer is assigned a unique priority that seeks to maximize the efficiency of the sequencing process. Judicious assignment of layer priorities help to minimize response time to client requests by vectoring received packets directly to relevant layers.

Rather than keeping the layers suspended until they're needed, all protocol layers run continually. Upon receiving control of the CPU from the OS, each layer examines certain flags in its API, as well

as its state variable, to determine what – if any, service it need perform. If these indicators are such that no service is required, the task will delay itself for one OS tick, yielding the CPU to the next layer. Although *dynamic* priority alteration could be used to further improve sequencing efficiency, doing so would have increased the OS kernel size, as well as packet processing latency. As a compromise, certain layers call the OS function *OSTimeDlyResume*, thereby allowing a previously delayed, but required layer to run immediately. Other sequences are possible, depending upon the nature of the request.

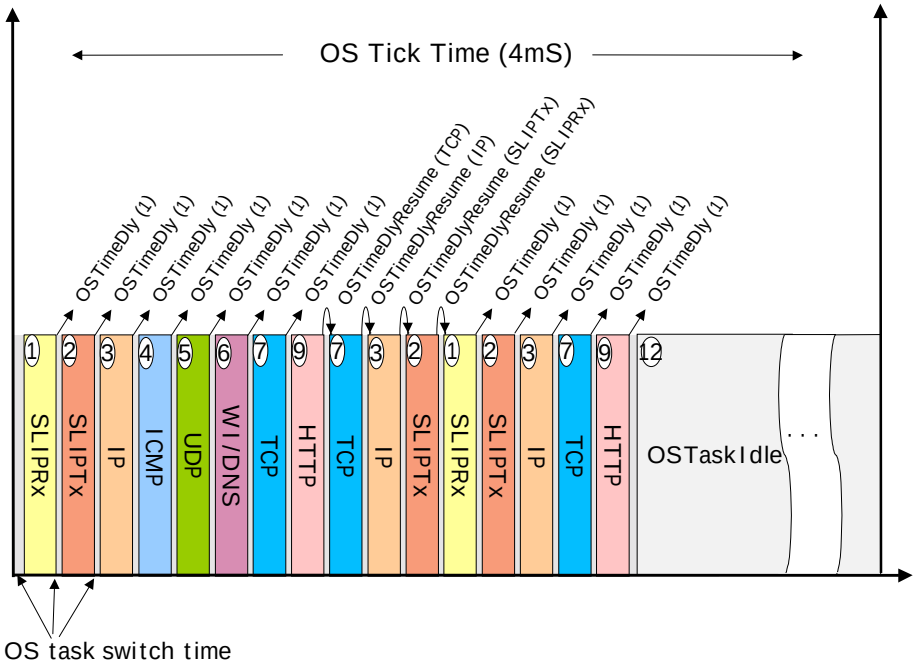

Figure 5. Protocol Task flow (typical HTTP request)

A. The Transport Layer – TCP

Several data structures are created and maintained by TCP to facilitate coherent operation. The RFC explicitly refers to one such data structure, namely the TCB, or *Transmission Control Block*. TCP must create and maintain one of these structures for each active socket. It is instructive to examine the nature and operations of these structures, which will help shed some light on the interworkings of the various protocol layers. We will therefore discuss each of these structures in some detail.

1. The TCP interface, or, "API"

The Application layer interfaces with TCP by means of the data structure listed in Figure 6. Similar to the "call-return" API's found in many stacks, this *API* includes all necessary parameters to permit concurrent TCP utilization by multiple applications.

The TCP task reacts to events (occurring at the IP level), and responds to commands (issued by the application(s)). Each application wishing to use TCP's services must first issue a **TCP_OPEN** command. If successful, TCP will return a unique connection name (CRSOCK_T *Socket) thereafter used by the application for all future interactions with TCP. Specifically, an event is defined as either:

1. The reception of a TCP segment from the underlying IP layer as indicated by the **IPRECV** flag in the **Cmd** field of IP's API data structure…
2. The successful passing of a segment to IP for transmission as indicated by the **IPSEND** flag in that same **Cmd** field.

```
typedef struct tcpapi_t{
    UWORD       LocPort;        /* Our (local) Port number             */
    UWORD       ForPort;        /* Foreign (peer's) Port number        */
    QUADB_T     ForIP;          /* Peer's IP address                   */
    UWORD       Cmd;            /* Command from Application layer      */
    UWORD       Status;         /* Status of the TCP layer             */
    UWORD       Tout;           /* Timeout for various TCP states      */
    UBYTE       *RxBuf;         /* Pointer to Application's Rx buffer  */
    UWORD       RxCount;        /* Number of bytes recd                */
    UWORD       RxBufLen;       /* Application's Rx buffer length      */
    UBYTE       *TxBuf;         /* Pointer to RAM data to send         */
    UWORD       TxCount;        /* Number of RAM bytes to send         */
    const       UBYTE   *html;  /* Pointer to ROM-based data to send   */
    UWORD       HtmLen;         /* Number of ROM bytes to send         */
    CRSOCK_T    *Socket;        /* Pointer to Transmission Control     */
                                /* Block for specified connection name */
} TCPAPI_T;
```

Figure 6. TCP's API data structure

Commands are issued by the overlying Application layer via flags in the **Cmd** field of the **TCPAPI_T** data structure. TCP recognizes the following commands (although not all are currently supported):

```
#define TCP_OPEN    BIT0    /* OPEN a connection                */
#define TCP_RECV    BIT1    /* Data has been received           */
#define TCP_CLOSE   BIT2    /* CLOSE this connection            */
#define TCP_STATUS  BIT3    /* Not implemented                  */
#define TCP_ABORT   BIT4    /* ABORT this connection            */
#define TCP_SEND    BIT5    /* SEND specified buffer(s)         */
#define TCP_SENDACK BIT6    /* SEND an ACK only                 */
#define TCP_ACTIVE  BIT8    /* Used in conjunction w/ OPEN      */
#define TCP_MORE    BIT9    /* Indicates whether the application*/
                            /* is finished sending all data     */
```

Figure 7. TCP's command definition

2. **The TCP Socket.**

Upon receiving the **TCP_OPEN** command, TCP will create a TCP Control Block (if one is available) and provide the application with a pointer to this structure. Thereafter, this pointer is used by the application(s) as a "handle" to uniquely identify itself from among the other (if there are others) applications requesting service from TCP. As illustrated in Figure 8, TCP Control Blocks are linked together with any others that may have been created, forming a list of active sockets. (*Note that, due to the RAM requirements of the HTTP layer, only *one* such socket may be created on the Cannonball.)

The application may request TCP to open an *active* or *passive* connection, indicated by setting or clearing the **TCP_ACTIVE** flag. An *active* open is one in which TCP will initiate a connection by sending a **SYN** segment to a peer. In contrast, a passive open is one where TCP *waits* for such a SYN segment from a peer. Although both active and passive opens are supported, a *server*, by definition, awaits connection requests from a peer, and will therefore issue a passive open command to TCP.

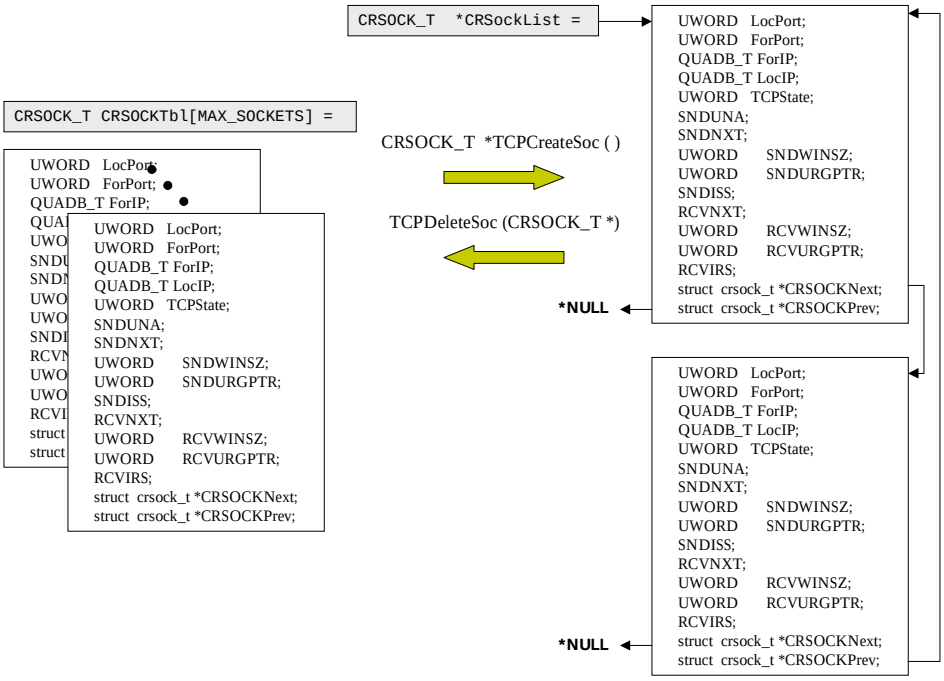

Figure 8 - TCP Socket List

3. **The TCP Segment.**

Once receiving data to transmit, TCP will partition the data into one or more segments, depending upon both the size of the data and the availability of such segments. Just like the TCP Control Block, segments are created and deleted as needed. Once a segment is created it is placed on the active queue (**XmitQList)** to await ACKnowledgment. If and when it is acknowledged, it is deleted, or, returned to the list of available segments

(**XmitQFreeList)**. As shown in Figure 9, as they are created, TCP segments are linked together, forming the linked-list of active segments.

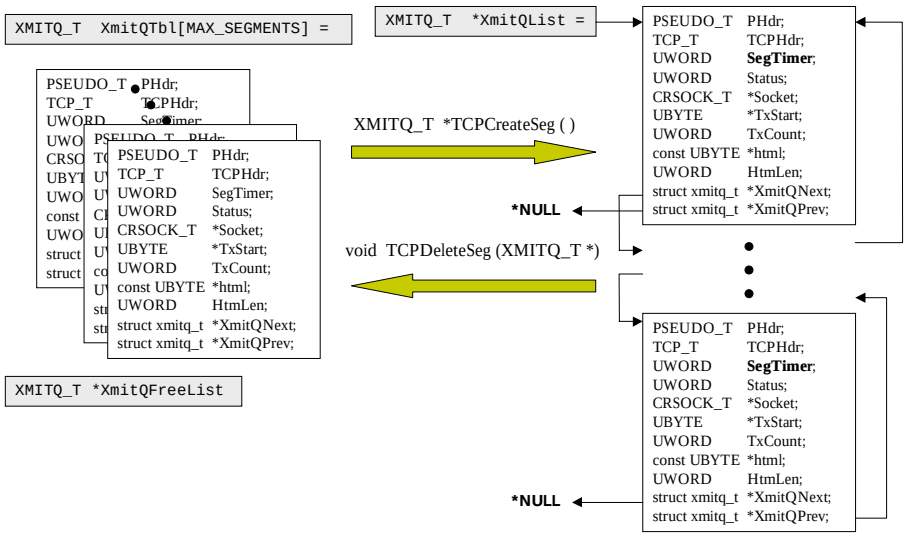

Figure 9. TCP Segment Queues

Each queued segment holds only those parameters necessary to ensure that the segment can be accurately retransmitted in the event it is not properly acknowledged. The segment data itself is *not* queued, since this would require enormous amounts of RAM. Only the segment's relevant parameters are queued. Upon creation, each segment is timestamped with the current OS time. When the TCP has sent all available segments, it periodically updates and checks the timer field of every "unACK'd" segment on the queue. If any or all of the segments awaiting acknowledgment on the queue timeout, they will be retransmitted once. If the same segment times out again, a reset is sent to the peer, and the TCP closes.

Again, due to the RAM consumption of the HTTP layer, this implementation creates only a few such segments (4-6) before having to wait upon an ACKnowledgement from the peer. Upon receiving data bearing or certain other control segments, the receiving TCP is required to inform the sending TCP that it has successfully received these segments. This is accomplished by sending ACKnowledge segments whose Acknowledgement Number field indicates the Sequence Number of the last byte successfully received. This permits the sending TCP to continue sending segments unabated. Should a segment not be acknowledged by the peer in a timely fashion (due to some error or other damnable event), the sending TCP will take note and resend this segment. Upon receiving an "ACK" segment from the peer, the previously transmitted segments sitting on the **XmitQList** queue are checked against the received Acknowledgement number to determine whether this ACK affects them. If it does, they are removed and returned to the **XmitQFreeList** queue, making them available for reuse. If it does not, they remain on the queue until they are acknowledged, or they timeout and are resent.

In the event that all segments have been allocated and yet more application data remains to be sent, TCP must wait for the peer to ACKnowledge all/any of the previously sent segments. During this time, TCP updates the **SegTimer** fields of the segments awaiting ACKnowledgement. Should a segment timeout it will be retransmitted. Should it timeout a second time TCP will infer that an error of some sort has occurred at the peer, and will issue a RESET segment. After sending the reset segment, TCP will revert to the listening state and attempt to send the entire data again. The timeout values used are fully flexible and determined by macros in the TCP header file.

4. **Transmitting Data.**

An application wishing to transmit data must first determine whether TCP is available by checking the **TCP_SEND** flag. If TCP is available, the application must fill in the necessary fields in TCP's API (indicating for instance, the location(s) of the data buffers to transmit), and then set the **TCP_SEND** flag. When the TCP task next runs, it will check for any commands from the application(s). Upon noting that the **TCP_SEND** flag is set, TCP will:

1. Use the ***Socket** (connection name) field to determine which TCP Control Block to access…
2. Determine whether the total data length exceeds that of the peer's Maximum Segment Size (**MSS**). If it does exceed this limit, TCP will grab a "MSS-sized" portion of the data, to ensure that the receiving TCP can handle the segment…
3. Acquire a TCP segment (if one is available) from the TCP segment manager, who places the segment in the **XmitQList**. If no segments are available, TCP must wait…
4. Compute the total segment length and record this number in the **Length** field…
5. Transcribe the current Sequence numbers from the socket's control structure…
6. Compute the segment checksum and record it in the **HdCkSum** field…
7. Record the segment's timeout value in its **SegTimer** field…
8. Copy the appropriate buffer addresses to the pointers in IP's API …
9. Determine whether this is the final segment to send. If so, set the **FIN** flag in the TCP segment header. Otherwise, only the **PSH** and **ACK** flags are set…
10. After ensuring that IP is available (by checking IP's DLSEND flag), signal the IP layer to transmit this segment by setting the **DLSEND** flag in IP's API …
11. Repeat steps *two* through *nine* above until all application data has been sent.
12. Once all application data has been transmitted, TCP will await ACKnowledgement of the FIN segment, as well as any other un-ACK'd segments…
13. Upon receiving proper ACKnowledgements, the connection is closed.

This process is illustrated in Figure 10. The application (HTTP in this case) collects the various data and passes the relevant parameters on to TCP.

5. **Receiving Data.**

TCP monitors the **DLRECV** flag in the **Cmd** field of IP's API data structure for received segments. Upon receiving a segment from IP, TCP will:

1. Confirm the segment's checksum. If incorrect, the segment is *silently discarded* (i.e. ignored)…
2. Compute the payload length (application data) for use by the application …
3. Copy the protocol number to the API for multiplexing purposes…
4. Flag the Application layer(s) that data has been received…
5. Delay itself by one tick to allow the application layer protocol(s) to run.

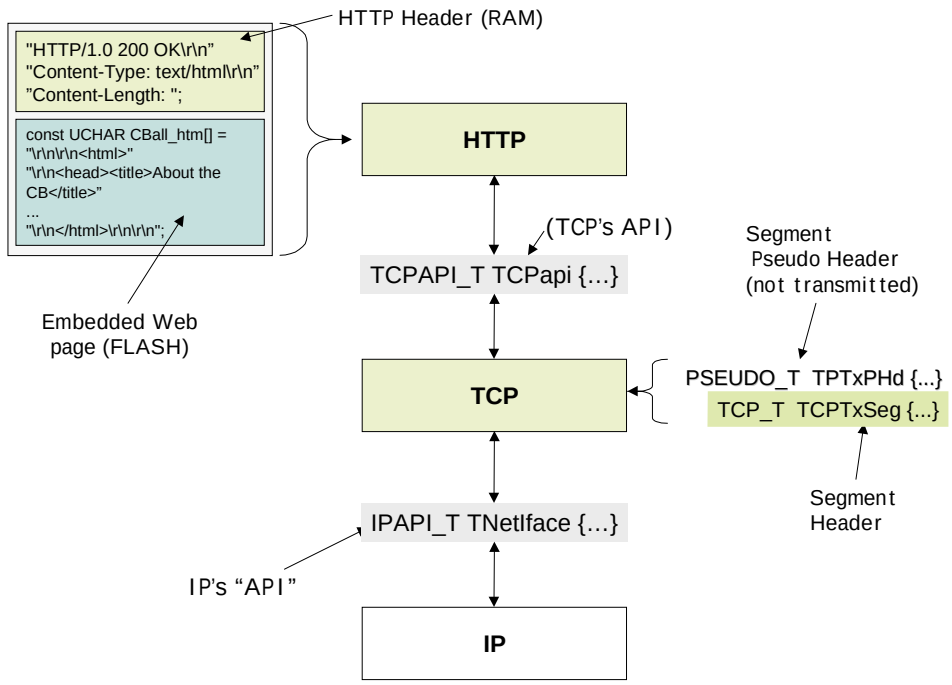

Figure 10. Application/TCP interlayer communication

B. The Transport Layer – UDP

UDP is supported to facilitate limited NetBIOS, DNS, and WINS services. These applications use UDP to send and receive service requests and responses. The User Datagram Protocol is significantly simpler than its big brother TCP. In contrast, UDP is by definition a "connectionless" transport protocol. This means that UDP establishes no connection with the peer prior to sending or receiving datagrams, and that no state information is maintained to ensure reliable delivery of these datagrams. The UDP layer interfaces with applications in a manner similar to TCP. Its API is defined in Figure 11.

```
typedef struct udpapi_t{
    UWORD   LocPort;            /* Our Port number                    */
    UWORD   ForPort;            /* Peer's Port number                 */
    QUADB_T LocIP;              /* Structure holding local and Dest IP */
    QUADB_T ForIP;              /* addresses                          */
    UWORD   Cmd;                /* Application Command                */
```

```
    UWORD   Status;             /* Holds the Status of the UDP         */
    UBYTE  *RxBuf;              /* Pointers to the Rx and Tx buffers   */
    UWORD   RxCount;            /* Number of bytes recd                */
    UWORD   RxBufLen;           /* Rx buffer length                    */
    UBYTE  *TxBuf;              /* Pointer to data to send             */
    UWORD   TxCount;            /* Number of bytes to xmit             */
    const   UBYTE   *html;      /* Points to any const data to xmit    */
    UWORD   HtmLen;             /* Number of bytes in HTML page        */

} UDPAPI_T;
```

Figure 11 - UDP's API structure

Where these are the command semaphores…

```
      #define     UDP_RECV    BIT1
      #define     UDP_SEND    BIT5
```

C. The Network Layer (IP)

The IP layer interfaces with Transport protocols sitting above it, such as TCP and UDP (and logically ICMP), and Data link protocols sitting below it, such as SLIP, PPP, and Ethernet. The Transport layer interfaces with the IP layer by way of two data structures:

1. **IPAPI_T** holds the command from the Transport layer, as well as several pointers necessary to locate the various data.

```
typedef struct ipapi_t {
    UBYTE   Cmd;            // Command flags from Transport layer
    UBYTE   RxProtoc;       // Protocol number of Rx datagram
    UBYTE   TxProtoc;       // Protocol number of Tx datagram
    UBYTE   TTL;            // Time To Live value
    QUADB_T ForIP;          // Source and Dest IP ddresses
    QUADB_T LocIP;
    UBYTE  *RxBuf;          // Pointers to the Rx and Tx buffers
    UWORD   RxCount;        // Number of bytes recd
    UWORD   RxBufLen;       // Rx buffer
    UBYTE  *TxBuf;          // Pointer to RAM-based Tx data
    UWORD   TxCount;        // RAM-based data length
    const   UBYTE   *html; // Pointer to ROM-based Tx data (HTML page?)
    UWORD   HtmLen;         // ROM-based data length
    TCP_T   *TCPTxSeg;      // Pointer to TCP Tx segment header
    TCP_T   *TCPRxSeg;      // Pointer to TCP Rx segment
    UDP_T   *UDPTxSeg;      // Pointer to UDP Tx segment header
    UDP_T   *UDPRxSeg;      // Pointer to UDP Rx segment header
    ICMP_T  *ICMPTxSeg;     // Pointer to ICMP Tx segment header
    ICMP_T  *ICMPRxSeg;     // Pointer to ICMP Rx segment header
}IPAPI_T;
```

Figure 12. IP's API structure

IP responds to the following commands...

```
#define IPSEND  BIT0
#define IPRECV  BIT1
```

2. **IPv4_T** holds the IP header information.

```
typedef struct ipv4_t{
   UBYTE       Ver_HL;          // Version and Header length byte
   UBYTE       ToS;             // Type Of Service
   UWORD       Length;          // Total datagram length
   UWORD       Ident;           // Fragment Identification field
   UBYTE       FlgsOffst;       // Flags and MS offset bits
   UBYTE       Offst;           // LS offset bits
   UBYTE       TTLive;          // Time-To-Live byte
   UBYTE       Protocol;        // Transport Protocol byte
   UWORD       HdCkSum;         // One's compliment header checksum
   QUADB_T     SrcIP;           // Source IP address
   QUADB_T     DestIP;          // Destination IP address
   } IPv4_T;
```

Figure 13. IP's Header structure

Two **IPv4_T** structures are created during initialization – one for transmit and one for receive. Operation is straightforward: the flags IPSEND and IPRECV are monitored for input from TCP, while also monitoring the flags DLRECV and DLSEND for the Link layer. When a segment is readied to send by TCP (IPSEND flag is set), IP will:

1. Compute the total datagram length and record this number in the Length field…
2. Compute the header checksum and record it in the HdCkSum field…
3. Flag the Link layer by setting the **DLSEND** flag in **DLAPI_T** to send the datagram.

Upon receiving a datagram from the Link layer (**DLRECV** flag set), IP will:

1. Confirm that the header checksum is correct. If incorrect, the datagram is ignored…
2. Compute the TCP segment length for use by the Transport layer (TCP or UDP), or ICMP
3. Copy the protocol number for multiplexing…
4. Flag the appropriate layer (Transport or ICMP) that a datagram has been received.

NOTE*: This IP does not currently support fragmentation*.

D. ICMP (Internet Control Message Protocol)

ICMP (Internet Control Message Protocol) is a required and integral part of IP, although it logically sits above it. ICMP messages are transmitted in IP datagrams just like TCP and UDP messages. All IP's are required to implement certain minimum ICMP features. This implementation supports the following ICMP message types:

- 8 - Echo Request (used by ping)
- 10 - Router Solicitation

To facilitate future expansion, ICMP functions are conveniently tabulated as shown in Figure 15. Every host MUST implement an ICMP Echo server function that receives Echo Requests and sends matching Echo Replies. The IP source address in an ICMP Echo Reply MUST be the same as the specified destination address (defined in Section 3.2.1.3) of the corresponding ICMP Echo Request message.

Router Solicitations received are responded to by advertising an appropriate Router IP address. This router address will be based upon the same class B network address of the requester. Although not necessary for most applications, this feature is included for the potential uses it may find by our audience.

```
UWORD   (*const ICMP_Table[])(void*, void*, void*) = {ICMPEchoReply,
                                                      NULL,
                                                      NULL,
                                                      NULL,
                                                      NULL,
                                                      NULL,
                                                      NULL,
                                                      NULL,
                                                      ICMPEchoReq,
                                                      NULL,
                                                      ICMPRouterSlct};
```

Figure 15. Array (Table) of supported ICMP message types

E. Data Link Layer (SLIP/PPP)

This version of the stack includes support for the Serial Line Interface Protocol (SLIP). SLIP is a very simple protocol used primarily for its quick and easy implementation. One major drawback associated with SLIP is its lack of any accommodation for software flow control (ASCII control characters XON and XOFF). RAM is a precious and limited resource in most embedded controllers, and the Cannonball is no exception. The lack of flow control requires that we maintain a sufficiently large Rx buffer to prevent unexpected overflows.

Subsequent versions will include support for the Point-to-Point Protocol (PPP). PPP affords the user with flow control by what is termed "transparency". Transparency allows one to utilize standard ASCII control characters by "escaping", or, encoding them. This, and various other PPP options, is configured at the beginning of the PPP connection using the Link Control Protocol (LCP).

```
typedef struct dlapi_t{          // Common Data Link Layer API
    UBYTE   Cmd;
    TCP_T   *TCPTxSeg;           // Pointer to TCP Tx segment
    TCP_T   *TCPRxSeg;           // Pointer to TCP Rx segment
    UDP_T   *UDPTxSeg;           // Pointer to UDP Tx segment
    UDP_T   *UDPRxSeg;           // Pointer to UDP Rx segment
    ICMP_T  *ICMPTxSeg;          // Pointer to ICMP Tx segment
    ICMP_T  *ICMPRxSeg;          // Pointer to ICMP Rx segment
    IPv4_T  *IPTxDatagram;       // Pointer to IP Tx Header
```

```
    IPv4_T  *IPRxDatagram;       // Pointer to IP Rx Header
    UBYTE  *RxBuf;               // Pointers to the Rx and Tx buffers
    UWORD   RxCount;             // Number of bytes recd
    UWORD   RxBufLen;            // Rx buffer length
    UBYTE  *TxBuf;               // Points to RAM buffer for dynamic data
    UWORD   TxCount;             // Number of bytes to xmit
    const   UBYTE   *html;       // Points to ROM-based data
    UWORD   HtmLen;              // Number of bytes in HTML page

}DLAPI_T;
```

Figure 14. Data Link API structure

SLIP responds to the following commands...

```
#define     DLSEND      BIT0
#define     DLRECV      BIT1
#define     ENDRECD     BIT7
```

Figure 15 represents a schematic of the IP-Data Link interface.

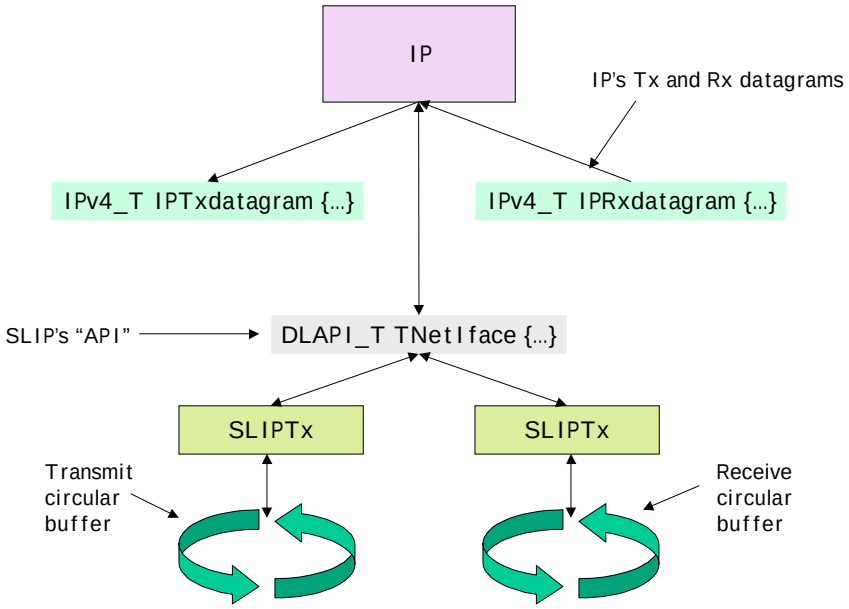

Figure 15. IP/Data Link Interface.

F. The Application Layer (HTTP 1.0)

A limited version of HTTP 1.0 is included. The supported *methods* (or, commands) are GET and POST - all others are ignored. A schematic of this layer is shown in Figure 16. This HTTP can operate with both static (FLASH-based) and dynamic (RAM/EE-base) user defined pages.

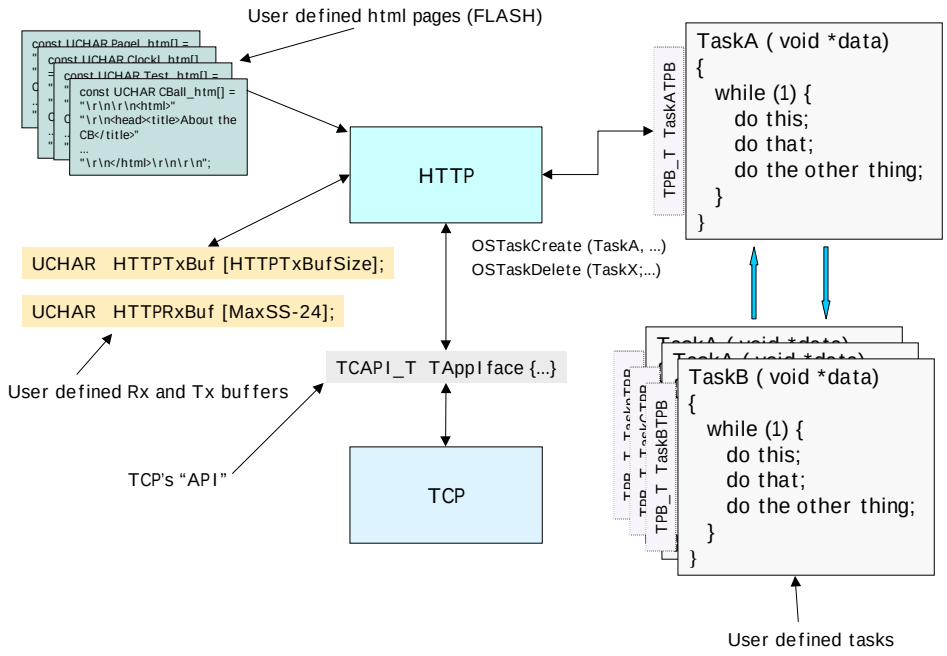

Figure 16. Application Layer (HTTP) schematic

In addition to simply returning static Web pages, this server is capable of responding to *forms*. Similar in function to cgi's, the user may define application-specific "scripts" (tasks) which may be spawned by the Server to perform user-specified functions. As the user "submits" form data to the server, these user-defined tasks operate on this data and return results or take user-specified actions. These tasks may, or may not, return a new or modified Web page. These built-in features provide a user with a great deal of flexibility in creating fully interactive applications.

User tasks are written to operate under the RTOS. Special control blocks and data structures are used to manage all this activity. These are defined and maintained in a central location, making development a straightforward process. Users may create their own embedded Web pages in a number of ways. One easy method is to first write them in standard HTML and then add the necessary formatting to transform them into an array of characters, making them amenable to the 'C' compiler. Once understood, this process is remarkably quick and easy.

This server was developed with efficiency in mind. To that end, Web pages may share common headers and footers, thereby giving them a uniform look and feel. Alternatively, users may opt

to adopt a monolithic style. Both are supported in this server. Moreover, Web pages may include provision for dynamic data – i.e. data returned by one of the user-specific tasks may be inserted into the page by the server prior to transmission. To support this feature, all embedded pages follow a defined format, and predefined constructs are utilized to organize and manage the transmission of each page.

Formatting your own embedded html page proceeds as follows:

1. Web pages may comprise one or more discrete sections, each of which will assume the form of a array of characters. Sections should be given a common logical name, similar to that of Figure 17.
2. Begin the page with 2 (two) CR/LF (\r\n) sequences. (The user-defined portion is the HTTP "entity body". The body follows the HTTP header, and must be delimited by 2 carriage return/line feed sequences.)
3. You may optionally begin each line with a CR/LF sequence (\r\n) to make the browser's "source view" easier to read (these may be omitted to save space).
4. Escape any "C" specific characters (such as quotes or percentage signs). For example, any quotation marks within your html code must be replaced with \".
5. Delimit each line with quotes to form one string literal.
6. Terminate each array (one or more) with a semicolon as per "C" coding requirements.

As a simple example, consider the following html listing using the above approach. Figure 21 shows just how Internet Explorer renders this very short section of embedded html code.

```
const   UBYTE   err504_htm[]  =
"\r\n\r\n<HTML><HEAD>"
"\r\n<TITLE>HTTP 504</TITLE>"
"\r\n</HEAD>";

const   UBYTE   err504_htm2[]  =
"<BODY><H1>HTTP Error 504 - Remote Node Error</H1>"
"\r\nNode ";

const   UBYTE   err504_htm3[]  =
" responded with error: ";

const   UBYTE   err504_htm4[]  =
"\r\n<P><HR><ADDRESS>Jeffs/1.3.9 Server at www.cr16.com "
"\r\n Port 80</ADDRESS>"
"\r\n<p><a href=\"index.htm\">HomePage</a>"
"\r\n</BODY></HTML>";
```

Figure 17. Simple embedded page.

7. Once your page is written, you now insert pointers to the page's addresse(s) and other pertinent parameters into a control structure as defined in Figure 18. (See Figure 19 for this page's data structure.)
8. Finally, you must give your page an Identifier and insert it into the list of pages as shown in Figure 20. This is simply the name you wish to use for the object. HTTP resources are commonly identified by what they call a URI, or *Uniform Resource Identifier*. For example,

assuming as your domain *www.yourserver.com*, this object would assume a URI something like *www.yourserver.com/err504.htm*.

```
typedef const struct html_t {
    TPB_T *const tpb;         // Pointer to task's Task Parameter Block
    const  UWORD NumBlocks;   // Number of discrete blocks in this page
                              // Pointer to array of page pointers
    const  UBYTE *const (*BlkAddrs)[];
    UWORD  const (*BlkSz)[];  // Pointer to array of page sizes
    UWORD  const PageSize;    // Total page size in bytes
}HTML_T;
```

Figure 18. HTML control structure.

```
      const UBYTE *const err504Ptrs[] = {err504_htm,
                                         err504_htm2,
                                         err504_htm3,
                                         err504_htm4};

      const UWORD err504Szs[]         = {sizeof (err504_htm) - 1,
                                         sizeof (err504_htm2) - 1,
                                         sizeof (err504_htm3) - 1,
                                         sizeof (err504_htm4) - 1};

      HTML_T err504_html              = {&HTTP504TPB,
                                          4,
                                          &err504Ptrs,
                                          &err504Szs,
                                          sizeof (err504_htm)
                                        + sizeof (err504_htm2)
                                        + sizeof (err504_htm3)
                                        + sizeof (err504_htm4) - 4};
```

Figure 19. 504 Page's HTML control structures.

```
const   UCHAR   err504Name[] = "/err504.htm";

const   UCHAR   *const  PageNamePtrs[]  = {IndexName,
                                           err504Name,
                                           .,
                                           .,
                                           .,
                                           NULL};
```

Figure 20. List of embedded objects (pages).

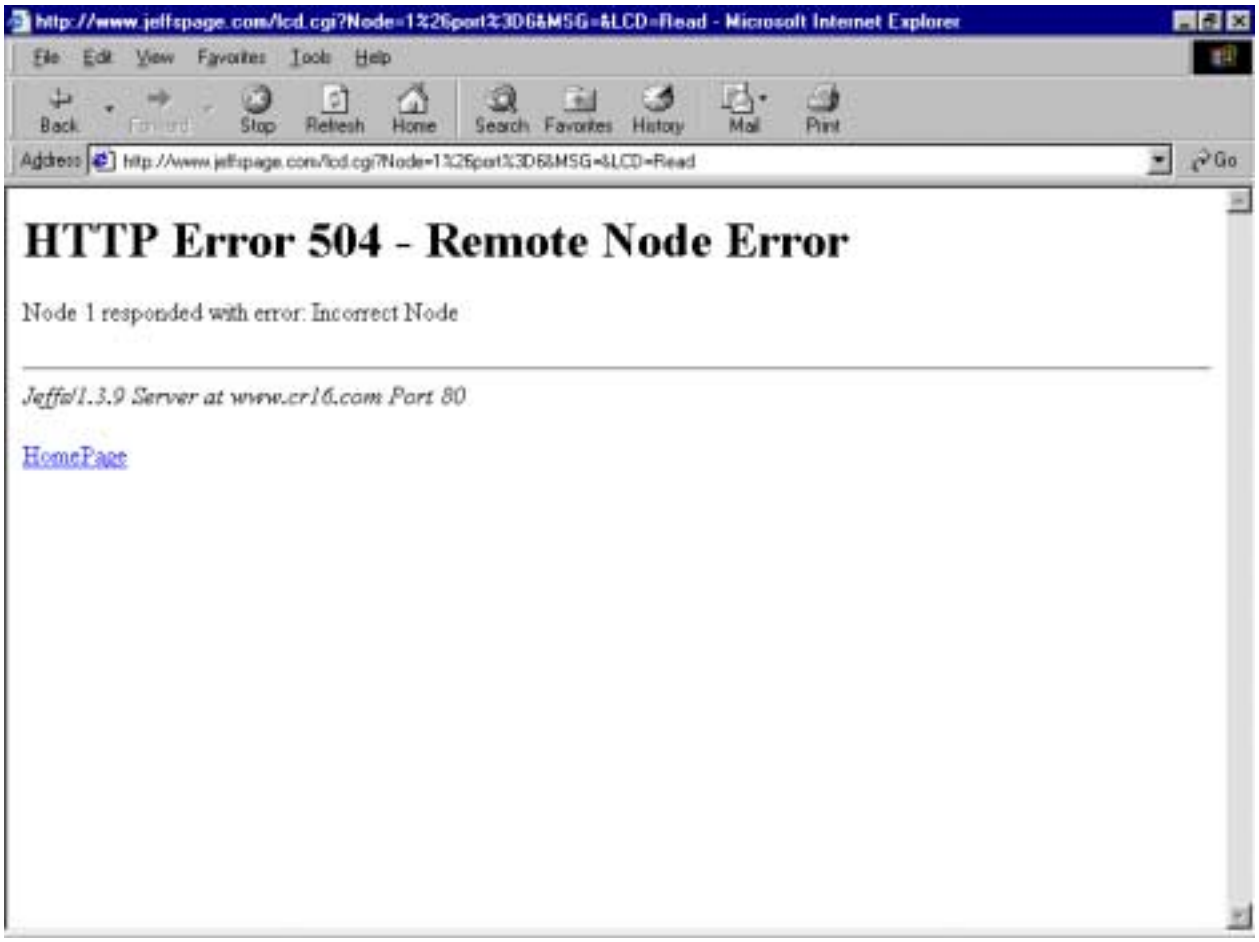

Figure 21. 504 Page as Rendered with Internet Explorer.

G. Naming Services (WINS/DNS)

Some aspects of both WINS and DNS are supported to allow the use of url based resource addressing, and to respond to NetBIOS name registration requests. Included primarily as a development tool, these services may find use in actual applications. Both WINS and DNS use UDP as a transport layer. This implementation currently supports only those features necessary to resolve urls into IP addresses, and vice versa. This implementation also supports the following Windows-based network tools:

- **Tracert** – determines and lists the route taken to a destination by sending ICMP echo requests with varying TTL (Time-To-Live) values to the destination. One may specify either an IP address or a URL as the destination. This implementation currently resolves all IP addresses to the *cannonball.com.* Conversely, urls are resolved into Ipv4 addresses.
- **Route** – displays current routing table.

In a manner similar to ICMP, provision is made for easy expansion by defining an array of supported functions. Each request "type" is used as an offset into these tables. Non-supported types are given NULL pointers.

```
UWORD   (*const DNS_Table[])(UBYTE *, UBYTE *, UWORD)   =   {DNSQuery,
                                                             DNSIquery,
                                                             DNSCquery};

UWORD   (*const WINS_Table[])(UBYTE *, UBYTE *, UWORD)  =   {WINSQuery,
                                                             WINSIquery,
                                                             WINSCquery,
                                                             NULL,
                                                             NULL,
                                                             WINSRquery};
```

Figure 22. Arrays (Tables) of supported DNS and WINS functions (msg types)

III. A CR16-based Embedded Web Server Demo

As a simple example of the many potential uses for such an embedded server, a demonstration system was constructed as illustrated in Figure 24. The system comprises several CannonBall evaluation boards all networked over CAN. One board assumes the role of a gateway from the TCP/IP spoken by the PC, to the CAN spoken by the network. Each node's peripherals may be controlled and accessed from a standard Web browser such as Netscape or Internet Explorer.

As Figure 24 indicates, every node may be configured to run a variety of tasks. Nodes operate in a manner similar to that illustrated in Figure 16. The HTTP Web server present on the gateway is replaced by a "high-level" CAN driver in the nodes. This CAN driver is responsible for interpreting browser requests and spawning any required tasks, as well as formatting task results for transmission over the CAN bus. At the gateway, the high-level CAN driver maintains a list of active nodes and their respective configurations. This allows the user to remotely manage nodes from a single control point. Figure 25 is a sample page from the project (as rendered by IE 5.0). You'll notice that the page includes several *gif* images. These were included to enhance the look of the pages, and were located on the PC's hard drive. All other html code is embedded within the FLASH memory of the CannonBall.

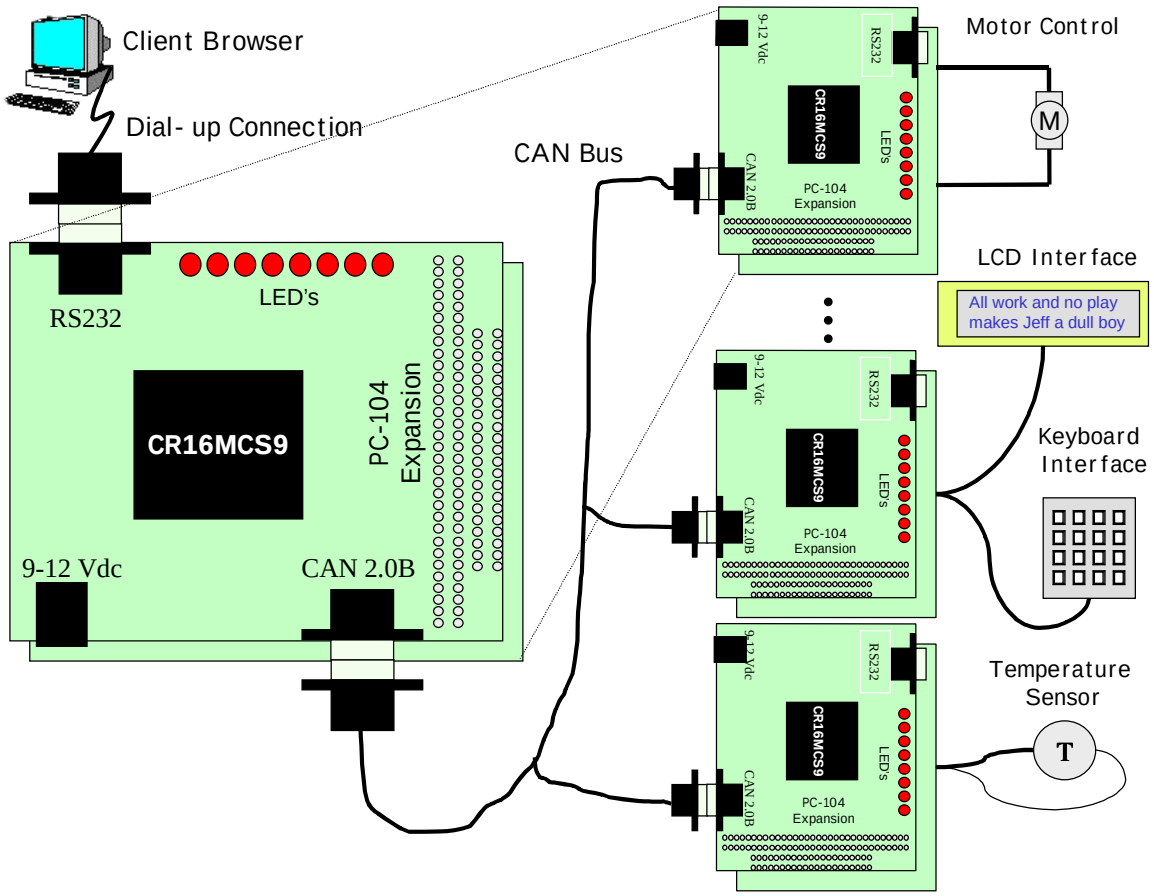

Figure 24. Embedded Web Server Demo system

Modeled after the *Dynamic Host Configuration Protocol* (DHCP) used by PC's to dynamically acquire its network configuration, a similar mechanism is employed to allow nodes to acquire certain needed network configuration data automatically. Using what we refer to as "Dynamic Node Configuration" (DNC), nodes may be added and deleted in a manner similar to the plug-and-play mechanism used in your PC. As a node is added, it issues a DNC request to the DNC server (located on the gateway), advertising it's randomly generated ID. The gateway's DNC server layer will then assign a new node number for this node and add it to its list of active nodes. If space is available, and the ID requested by the node is valid, DNC will acknowledge the request with a confirmation. Subsequent communications directed at that node will use the negotiated ID. Figure 26 illustrates this table after adding the first node. The number of nodes used in the demonstration was four, however, the only real limitation to the number of nodes that may be added is that of the size of the EE, where this node configuration table is kept.

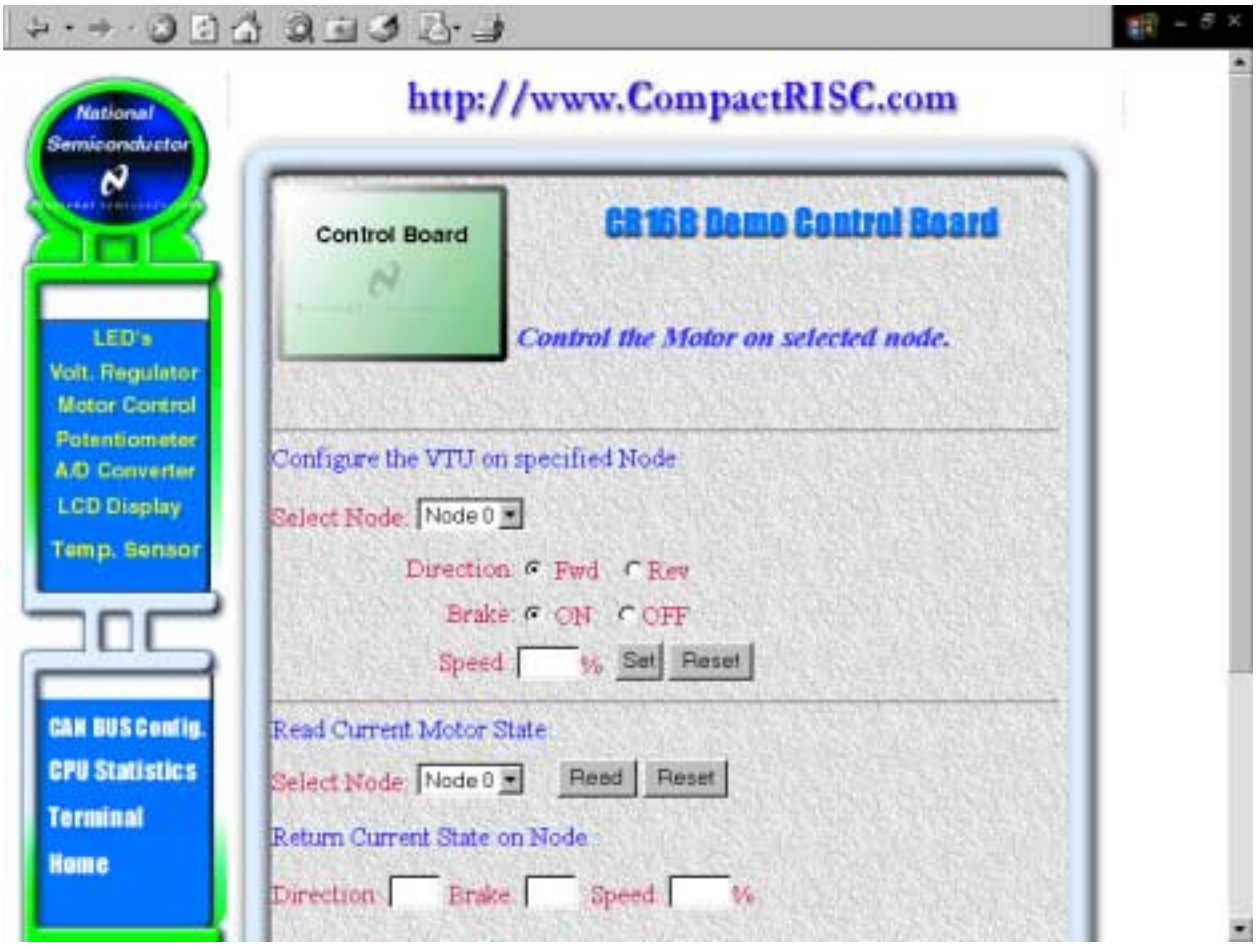

Figure 25. An Embedded Motor Control Page from the Project

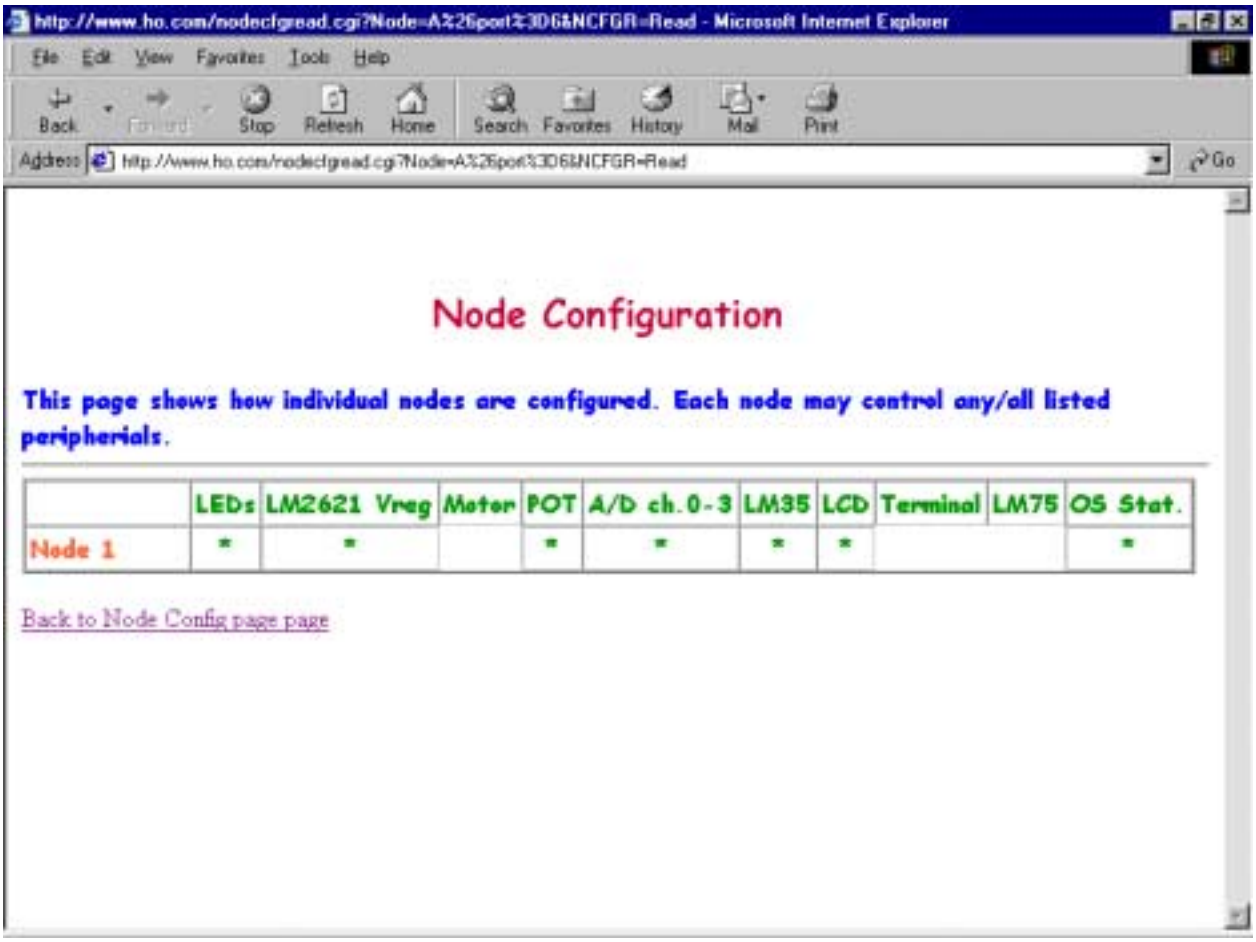

Figure 26. Node Configuration Table.

III. This Implementation's Compliance with the TCP Specification

RFC 1122 augments and obsoletes the original TCP, UDP, and IP specifications (RFC's 793, 768, and 791), by summarizing requirements and correcting various errors and shortcomings detected over the years. It conveniently delineates those portions of the specification that MUST or SHOULD be implemented in order to be in compliance (see Appendices A and B). Many of these features are wholly, or in part unnecessary in a controlled, embedded environment. As a result, some of these requirements are not implemented. In the following paragraphs We'll briefly touch on a few of the more important of these MUST's and SHOULD's, and just what is and isn't included in this stack - and whether it matters. You will note that many of the features listed are supported, but not in a comprehensive manner. Doing so would result in a prohibitively large and resource hungry implementation, ill suited for most microcontrollers. If full compliance is required, third-party software and widgets are available in abundance.

- **Window Management** 

Limited memory space does not permit the queuing of received segments, and so our window size is effectively one segment. However, since we expend every effort to keep pace with the incoming data, we advertise a larger window size, *and never adjust it.* This effectively precludes a window going silly on us. On the receive side, TCP checks the peer's window size before transmitting data. Should the peer's window be too small, TCP will delay transmissions until such time that the peer's window opens.

- **Support for the "zero window probe"** 

This is another tool utilized to augment the sliding window flow control mechanism. If we are currently sending data to a peer and the peer advertises a window size of zero (for some reason it cannot currently accept any more data), we should periodically transmit *probe segments* to discover if the connection is still viable or has been prematurely aborted. If zero window probing is not supported, the connection may hang forever in the unfortunate event that the peer's ACK segment that re-opens the window is lost (not an improbable event).

This implementation does not queue received segments. Rather, upon reception every segment is immediately passed to the Application. We therefore *never* advertise a Window size of zero. Probe segments, if received are ignored. If the peer advertises a zero window, data transmission is suspended. However, if the window remains closed beyond the segment timeout intervals, a reset will be issued.

- **Support for Timeout with Retransmission** 

Integral to TCP is the retransmission of segments that have not been Acknowledged (ACK'd) in a timely fashion. The RFC mandates the implementation of an algorithm developed by *Jacobson* for computing the smoothed *round-trip-time* (RTT), and an algorithm by *Karn* for computing the *retransmission timeout* ("RTO"). Additionally, all implementations must include *exponential back off* for successive RTO values for the same segment.

Sorry - this implementation deals with retransmission in a slightly *less expensive* fashion. Each transmitted segment is "queued". Actually, the segment data itself is *not* queued, since this would require enormous amounts of RAM. Only the segment's *relevant parameters* are queued. This allows for many more segments than the limited RAM would otherwise allow. Upon creation, each segment is "timestamped" with the current OS time. When the TCP has sent all available segments, it periodically updates and checks the timer field of every "unACK'd" segment on the queue. If any or all of the queued segments awaiting acknowledgment timeout, they will be retransmitted. If a segment times out a second time, a reset is sent to the peer and the TCP closes.

The RFC does briefly mention the fact that in small-host implementations (such as ours), segment queuing is often precluded due to limited buffer space. They remind us that this omission may be expected to *adversely affect TCP throughput*, since the loss of a single segment causes all later segments to appear to be "out of sequence". *Noted*.

- **Generating Acknowledgments** 

TCP requires that all queued data be processed before sending an ACK. Furthermore, if aggregate acknowledgement is implemented, the ACK interval may not exceed 0.5 seconds.

Since segments are not queued, segments are individually acknowledged. Maximum ACK delay is less than 0.5 seconds.

- **Support for Urgent Data** 

  Although the Urgent mechanism may be used in any application, it was originally used to send *interrupt* - type commands to a Telnet program. We ignore the Urgent flag and never set it in outgoing segments.

- **TCP Options** 

  All TCP's are required to be capable of receiving TCP options attached to any segment. In addition, all TCP's MUST ignore without error any TCP option it does not implement, and it MUST be prepared to handle an illegal option length (e.g., zero) without crashing or other undesirable activity. TCP mandates a default MSS value of 536 bytes in the event one side does not advertise to the contrary. Support for the Maximum Segment Size option is important because it permits limited RAM applications (i.e. most low-cost embedded apps) to limit the size of the incoming segments to one commensurate with its smaller buffers. If a MSS option is not received at connection setup, TCP MUST assume a default send MSS of 536 (576-40).

  We recognize and apply the peer's MSS. All other TCP options, though received, are ignored.

- **TCP Checksums** 

  TCP requires that the sender compute a checksum for every segment sent, and requires that the receiver confirm the checksum on all incoming segments. Some implementations ignore the checksum on received segments since PPP provides ample protection via its CRC. However, since TCP may not always run over PPP (it may run over SLIP, for instance), support for the checksum must be included even if it can be optionally omitted.

  Our implementation complies by allowing the user to optionally include the TCP checksum. Any received segments failing the checksum test are silently discarded. In the event a segment does fail the checksum, no acknowledgement is sent. The sending peer is expected to retransmit the segment after an appropriate timeout.

- **Use clock-driven Initial Sequence Number (ISN) selection** 

  TCP dictates that the Initial Sequence Number be generated by a 32-bit clock running at 250Khz. This is to guard against overlapping segments between two peers that may reestablish

a connection after a previous one was prematurely aborted. Our ISN is based on the 32-bit time-base used by the RTOS, but operates at the system tick rate, which may be anywhere from 20Hz to 200Hz. Although posing no realistic threat in most/all embedded environments, this feature may easily be amended to comply fully with the RFC, should it be deemed necessary in any specific application.

- **Use of the push flag** 

  TCP requires that if the implementation does not allow the Application to control the Push flag, it *MUST* be set by TCP in the final segment at a minimum.
  This implementation does *not* afford the application with control of this flag; TCP sets the Push flag in the final segment of every transmission.

IV. Compliance with the IP Specification

As with TCP, RFC 1122 additionally supplements the original IP specification (RFC-791), by summarizing requirements and correcting various errors and shortcomings detected over the years. It conveniently lists those portions of the specification that MUST or SHOULD be implemented in order to be in compliance (see Appendix B). Only a few of these features are really necessary in a controlled, embedded environment (such as ours) and are consequently omitted in whole or in part from this, as well as many 3rd party stacks. A few of the more common of these features are briefly discussed in the following paragraphs, and just what is and isn't included in this stack - and whether it matters.

- **IP Version Number:** 

  Currently, only Version 4 (Ipv4) type datagrams are supported. Support for IPv6 is TBD. If any other version number appears in the header of a received datagram, it is noisily discarded.

- **Checksum:** 

  A host MUST verify the IP header checksum on every received datagram and silently discard those datagrams that fail the test. As in the case of the TCP checksum, many stacks omit this computation since the CRC performed by PPP is more than adequate. However, if used over SLIP or another Link layer protocol that does no such error checking, the checksum is necessary. For this reason, it is optionally included in the IP layer.

- **IP Fragmentation and Reassembly:** 

  This IP does *not* support any form of fragmentation. Path discovery mechanisms are encouraged to preclude fragmentation. In our case, the MSS is determined at compile time, and is, for all practical purposes, equivalent to the MTU. This IP will never attempt to fragment a segment. On a dial-up connection, the MTU is essentially irrelevant. PPP does impose an arbitrary MTU of 1500 bytes, presumably to mimic that of Ethernet. Since our TCP advertises

its MSS, the peer should never decide to fragment a segment. This IP does not examine the More Fragments (MF) bit. Should a datagram be received with this bit set, this IP will not treat it any differently that any other datagram. This will eventually lead to checksum errors.

- **IP Options** 

IP defines several options, many or most of which never find any real use in common applications, let alone an embedded one. These Options provide IP with useful control functions needed in some situations, but for the most part, are simply not implemented in many stacks. Options include - provisions for timestamps, security, and special routing.

No IP options are supported in this stack, although provision is made to receive them. If received they are ignored.

- **ICMP** 

ICMP (Internet Control Message Protocol) was defined to be integral to IP, although logically it sits above it. What that means is that ICMP messages are transmitted in IP datagrams just like TCP and UDP messages. Nonetheless, ICMP is technically part of IP and all IP's are required to implement certain of its features. ICMP messages are grouped into two classes:

1. Error messages:
   - Destination Unreachable
   - Redirect
   - Source Quench
   - Time Exceeded
   - Parameter Problem
2. Query messages:
   - Echo
   - Information
   - Timestamp
   - Address Mask

- **3.2.2.6 Echo Request/Reply:** 

Every host MUST implement an ICMP Echo server function that receives Echo Requests and sends matching Echo Replies. The IP source address in an ICMP Echo Reply MUST be the same as the specific-destination address (defined in Section 3.2.1.3) of the corresponding ICMP Echo Request message.