Protocol Implementations for Web Based Control Systems

Sugoog Shon

Abstract: We describe the MiniWeb[7] TCP/IP stack (mIP), which is an extremely small implementation of the TCP/IP protocol suite running 8 or 32-bit microcontrollers intended for embedded control systems, and satisfying the subset of RFC1122 requirements needed for host-to-host interoperability over different platforms. Our TCP/IP implementation does sacrifice some of TCP's mechanisms such as fragmentation, urgent data, retransmission, or congestion control. Our implementation is applicable to web based controllers. The network protocols are tested in operational networks using CommView and Dummynet where the various operational parameters such as bandwidth, delay, and queue sizes can be set and controlled.

Keywords: CommView, Dummynet, HTTP, newtrok driver, protocol stack, TCP, UDP.

1. INTRODUCTION

Recently, the interest in connecting even small devices to an existing IP network such as the global Internet has increased greatly. In order to be able to communicate over the Internet, implementation of the TCP/IP protocol stack is needed. With the success of the Internet, TCP/IP protocol suite has become a global standard for communication. For embedded systems, being able to run TCP/IP makes it possible to connect the systems directly to the Internet.

The RFC1122[1] specifies requirements for host system implementations of the Internet protocol suite. This RFC covers the communication protocol layers: link layer, IP layer, and transport layer. Its companion RFC1123, "Requirements for Internet Hosts -- Application and Support", deals with the application layer protocols. A TCP/IP implementation that violates the requirements of host to host communication may not be able to communicate with other TCP/IP implementations and may even lead to network failures.

Traditional TCP/IP implementations are derived from adaptations of the Berkeley BSD TCP/IP implementation. The TCP/IP implementations have required far too many resources both in terms of code size and memory usage. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with RAM of a few kilobytes of data and ROM for less than 100 kilobytes of code.

There are numerous small TCP/IP implementations for embedded systems. The target architectures range from small 8-bit microcontrollers to 32-bit RISC architectures. An example of an 8 bit based TCP/IP implementation is the uIP[3][8], which needs around 5164 bytes of code space on an Atmega128 AVR system. The iPic match-head sized server [4], Jeremy Bentham's PICmicro stack [2], and the Atmel TCP/IP stack [5] are examples of implementations for embedded TCP/IP stacks. Some of the embedded TCP/IP implementations are simplified TCP/IP implementation and some are standards compliant TCP/IP implementation

Our goal is to realize the TCP/IP stack for a specific application such as a web controller, which performs simple jobs like switch on/off, controller settings, and variable readings. One instance of the embedded control systems is an Internet based RF tag reader that can read RF tag information over the Internet.

For some considerations for embedded control systems, a web application does not require support for urgent data and does not need to actively open TCP connections to other hosts. Removing those mechanisms can reduce the complexity required to put it into practice. Retransmissions may not be made by the TCP module in the embedded control system because nothing is known about the active connections. Omission of TCP's congestion control mechanisms may also be a consideration while an implementation with no congestion control might work well when connected to a single Ethernet segment, because congestion is primarily caused by the amount of packets in the network. No support for reassembling fragmented IP packets if fragmented IP

Manuscript received September 7, 2004; accepted January 27, 2005. Recommended by Editorial Board member Dong-Ho Cho under the direction of Editor Keum-Shik Hong. This work was supported from the Basic Research Program of the Korea Science and Engineering Foundation under Grant KOSEF R05-2002-000-00119-0.

Sugoog Shon is with the Department of Information and Telecommunication Engineering, Suwon University, San 2-2 Wau-Ri, Bongdam-Up, Hwasung-Si, Kyungki-Do 445-743, Korea (e-mail: sshon@suwon.ac.kr).

packets are quite infrequent [6] is also a consideration. For the embedded systems, the other issue of the TCP/IP implementation is to maintain both the code size and the memory usage to a minimum. TCP/IP is an order of magnitude smaller than any traditional generic TCP/IP stack today. Our implementation is an extremely simplified TCP/IP implementation that includes implementations of IP, ICMP, UDP and TCP. Our implementations have been ported to numerous 8- and 32-bit platforms such as the 8032, AVR, and 80386 CPUs. Finally, we apply the protocol to the embedded Web RF reader and discuss the code size for the implementation. We also verify the performance and the correctness of a specification.

2. TCP/IP PROTOCOL SUITE

The full TCP/IP suite [9] consists of numerous protocols, ranging from low level protocols such as ARP, which translates IP addresses to MAC addresses, to application level protocols such as HTTP that is used to transfer web pages as shown below in Fig. 1. Each layer's input/output can be modeled as data buffers related to the interface layer. For instance, data arrives asynchronously from both the network and the application, and the TCP/IP stack maintains data buffers in which packets are kept while waiting for service.

2.1. Ethernet network driver

The network driver is used to transfer data from the receive buffer ring of the NIC to the host PC's memory, or inversely. There are NIC controllers that facilitate removing data from a ring buffer by providing a remote DMA channel to transfer data from the ring to an I/O port that is readable by the host system. Because of the asynchronous nature of the operation, the driver must be interrupt driven. Typically, packet reception is given high priority since delaying of packet removal may overflow the receive buffer ring. If several packets in the ring have been queued, all packets should be removed in one process.

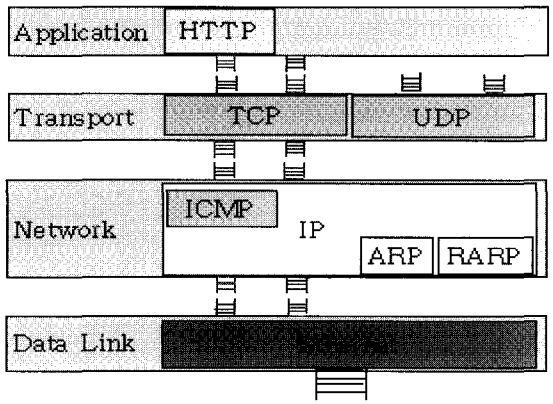


Fig. 1. TCP/IP layer and data buffer model.

Next, the packet receives interface between the IP layer and the data link layer processes when the incoming packet is addressed to a link-layer broadcast address [1].

2.2. IP

Usually, an IP package involves eight components: a header-adding module, a processing module, a routing module, a fragmentation module, a reassembly module, a routing table, and a reassembly table. The IP layer also includes input and output queues. The IP receives an IP packet, either from the data link layer or from a higher level protocol. Usually (Bidirectional case), the design uses two types of queues: input queues and output queues.

The lack of flow control can create a major problem in the operation of IP: congestion. The source host never knows if the routers or the destination host have been overwhelmed with datagrams. The source host never knows if it is producing datagrams faster than they can be forwarded by routers or processed by the destination host. However, embedded hosts have a limited-size queue (buffer) for incoming datagrams waiting to be forwarded or to be processed. If the datagrams are received much faster than they can be processed, the queue may overflow. In this case, the host has no choice but to discard some of the datagrams.

2.3. TCP

Transport layer protocols lie between user applications and the network. Although they offer user oriented services, their design is based on assumptions about network characteristics. TCP is a stream-service, connection-oriented protocol that utilizes a state transition diagram. It is so complex that its actual code is tens of thousands of lines. TCP operation supports flow and congestion control, and segmentation and reassembly of the user data stream. A TCP package involves a table called Transmission Control Blocks, a set of timers, and three software modules: a main module, an input processing module, and an output processing module [9].

In order to implement full TCP, periodic timeouts are required to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. The TCP/IP stack has an acknowledgment mechanism for the received packets. Very simply, when data arrives at the recipient, the protocol requires that it send back an acknowledgement of this data. The protocol specifies that the bytes of data are sequentially numbered, so that the recipient can acknowledge data by naming the highest numbered byte of data it has received, which also acknowledges the previous bytes. The protocol contains only a general assertion that data should be acknowledged promptly, but gives no more specific

indication as to how quickly an acknowledgement must be sent, or how much data should be acknowledged in each separate acknowledgement. TCP has mechanisms for limiting the amount of data that is sent over the network, and each connection has a queue on which the data is held while waiting to be transmitted. The data is not removed from the queue until the receiver has acknowledged the reception of the data. If no acknowledgment is received within a specific time, the data is retransmitted.

TCP breaks the byte stream into appropriately sized segments and each segment is sent in its own IP packet. The IP packets are sent out on the network by the network device driver. If the maximum packet size of the other network is smaller than the size of the IP packet, the packet is fragmented into smaller packets by the router. If possible, the size of the TCP segment is chosen so that fragmentation is minimized. The final recipient of the packet will have to reassemble any fragmented IP packets before they can be passed to higher layers. In many TCP implementations, including the BSD implementation, the urgent data feature increases the complexity of the implementation because it requires an asynchronous notification mechanism.

2.4. Application program interface

The Application Program Interface (API) defines the way the application program interacts with the TCP/IP stack. The most commonly used API for TCP/IP is the BSD socket API, which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API.

The BSD socket interface is used in multithread operating systems. A multithreaded environment is significantly more expensive to run not only because of the increased code complexity involved in thread management, but also because of the extra memory needed for maintaining the per-thread state.

2.5. Measurements

For TCP/IP implementations for high-end systems, processing time is dominated by the checksum calculation loop, the operation of copying packet data and context switching [10]. The network performance is analyzed to enhance QoS of TCP [11]. The performance of a network in the case of a TCP/IP is highly affected depending upon the Retransmit Time Out (RTO). In order to increase the network capability, the Round Trip Time (RTT) for a packet should be reduced.

For testing and verifying RTT for networking protocols, Dummynet is used as a tool that runs in operational networks, and/or through simulations [12].

3. PROTOCOL IMPLEMENTATIONS

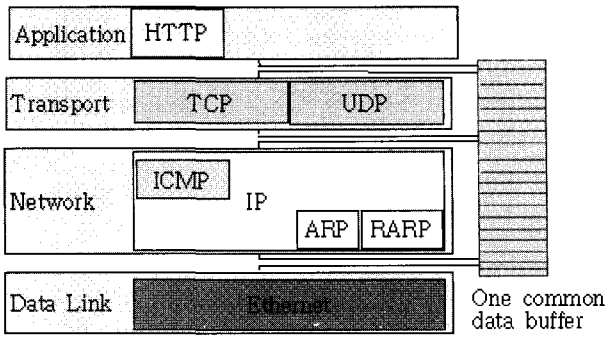


Fig. 2. Proposed data buffer model.

Our implementation of the subset of RFC 1122 or 1123 requirements can also provide interoperability of host-to-host communication. Many TCP mechanisms are essential if an embedded device is to communicate with another. However, it is possible to remove certain TCP/IP mechanisms that are very rarely used in special situations such as embedded control systems. The protocols in the TCP/IP protocol suite are designed in a layered fashion where each protocol performs a specific function and the interactions between the protocol layers are strictly defined.

One design goal for the MiniWeb TCP/IP(mIP) implementation is to minimize memory resources. In the mIP implementation, the protocols are tightly coupled in order to save code space. Of course, the interoperability to the Internet has to be satisfied. For mIP, we proposed one global buffer mechanism, instead of dynamic buffer and memory allocation. Thus, the incoming or outgoing data within each layer is common for each layer as shown in Fig. 2. The single global buffer can hold packets and has a fixed size for holding connection state. The size of the buffer is determined by a configuration option. The global packet buffer is large enough to contain one packet of maximum size.

The mIP can be run either as a task in a multitasking MicroC/OS system based on Intel80386 or Atmega128, or as the main program in a single tasking system based on 8032.

3.1. Network driver

When a packet arrives from a network, the Ethernet device driver places it in the global buffer and calls the mIP stack. If the packet contains data, the mIP stack will notify the corresponding application. Packets that arrive when the application is processing the data must be queued by the network device driver.

The network driver initiates a transmission whenever the upper layer software passes a packet to the driver or the inverse.

The mIP polls to check for packets from NIC's Interrupt Status Register (ISR) every time the periodic timer fires. The main loop consists of a packet transmitted routine and a packet received routine,

where the loop is concerned with interrupts originating from receptions, transmissions, and erroneous transmissions.

After the NIC initialization code completes successfully, the main function of the firmware goes into the loop. If the microcontroller pin INT0 goes high, this apparently signifies that a good packet has been received into the RTL8019AS's receive buffer ring. If a packet has arrived, the input handler of the mIP stack is invoked. The RTL8019AS's receive buffer ring precedes each incoming packet with a 4-byte header. This header contains useful information concerning the packet such as its length. Checksum in the link layer is calculated by taking the 16 bit sums. The sum is then one's complemented.

Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. RTL8019AS has a buffer with the size of 16 Kbytes on the chip. This RAM is completely controlled by the chip and offers buffering for multiple packet data. If the buffer is full, the incoming packet is dropped. This will cause performance degradation because mIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection. The driver's packet receives commands and automatically takes the packet off the receive buffer ring. The received packet is next stored into the global buffer.

On the other hand, if the driver is unable to transmit the packet in the global buffer immediately (i.e. receiver is busy), the packet in the global buffer is paused to transmit. After the NIC controllers interrupt the microcontroller to signal the end of the reception and indicate status information in the Transmit STATUS register, the packet in the global buffer can be transmitted.

3.2. Internet protocol (IP)

The IP layer code in mIP has two responsibilities: verifying the correctness of the IP header of incoming packets and demultiplexing the packet between the ARP, ICMP, UDP, and TCP protocols.

The ARP protocol maps between 32-bit IP addresses and 48-bit Ethernet MAC addresses and is needed for TCP/IP operation on an Ethernet. ARP requests for an IP address is to be overwritten to the outgoing IP packet for which the request is sent. When mIP gets an ARP reply, only the ARP mapping is updated. Implementation must support the specific-length packets. The packet receive interface between the IP layer and the link layer is summarized in Table 1. The mIP column indicates the item implemented for the link layer.

The ICMP protocol is used for reporting soft error conditions and for querying host parameters. Its main use is, however, the echo mechanism that is used by the ping program. The ICMP implementation in mIP is very simple as we have restricted it to only implement ICMP echo messages. Replies to echo messages are constructed by simply swapping the source and destination IP addresses of incoming echo requests and rewriting the ICMP header with the Echo-Reply message type. The ICMP checksum is adjusted using standard techniques [13]. Since only the ICMP echo message is implemented, there is no support for path MTU discovery or ICMP redirect messages. Neither of these is strictly required for interoperability; they are performance enhancement mechanisms. Table 2 shows the item implemented for the ICMP.

When incoming packets arrive, the IP layer performs a few simple checks, such as whether the destination IP address of the incoming packet matches any of the local IP address and verifies the IP header checksum. IP fragment and the fragment reassembly mechanism are not implemented in mIP. The current implementation only has a single buffer for holding packets to be reassembled, and therefore does not support simultaneous reassembly of more than one packet. We believe this to be a reasonable decision, since fragmented packets are uncommon. However, extending our implementation to support multiple buffers would be straightforward. Since there are no IP options that are strictly required and since they are very uncommon, mIP drops any IP options in received packets. Table 3 shows the items implemented for the IP protocol.

Table 1. Link layer implementations.

Link Layer Requirements	RFC 1122 (MUST)	mIP
ARP	0	0
Ethernet and IEEE 802 Encapsulation	0	0
Link layer report broadcasts to IP layer	0	x
IP layer pass TOS to link layer	0	x

Table 2. ICMP implementations.

ICMP Requirements	RFC 1122 (MUST)	mIP
ICMP Error message: -Destination Unreachable, Redirect, Source Quench, Time Exceeded, Parameter Problem	0	x
ICMP Echo Request or Reply	0	0
ICMP Timestamp and Timestamp Reply	0	x
ICMP Address Mask Request and Reply	0	x

Table 3. IP protocol implementations.

IP Requirements	RFC1122 (MUST)	mIP
Version check	0	0
Verify IP checksum	0	0
Addressing	0	0
Identification	0	x
Fragmentation and Reassembly	0	x
TOS	0	x
TTL	0	x
IP Options	0	x

3.3. Transmission control protocol (TCP)

TCP establishes the logical equivalent of a physical connection between two points. Data then passes bidirectionally along this connection. Both points must keep track of the data sent and received so that they can detect any omissions or duplications in the data stream.

A TCP software is implemented as a finite state machine that goes through a limited number of states. At any moment, the machine is in one of the states. It remains in that state until an event happens. The event can take the machine to a new state. We use cases to handle the state simply. Each state is implemented as defined in the state transition diagram [9].

Each TCP connection requires a certain amount of state information in the embedded device. Commonly used techniques [2] are listed such as lock step method, block sequencing, and byte-sequencing scheme. Because the state information uses RAM, we have aimed towards minimizing the amount of state needed for each connection in our implementations.

The TCP implementation in mIP is driven by an incoming packet. The packet is parsed by TCP. After verifying the TCP checksum, the source and destination port number and IP address are used to demultiplex the packet. When the sequence and acknowledgment numbers have been checked, the packet will be handled differently depending on the current TCP state. If the packet is a connection request, TCP allows a connection to listen for the incoming connection request that is identified by the 16-bit port number.

For a flow control, mIP does not implement sliding window mechanism, instead, it just uses a simple window mechanism. The receiver of data returns to the sender a number, which is the size of the packet that the receiver currently has available for additional data. This number of bytes, called the window, is the maximum that the sender is permitted to transmit until the receiver returns an additional window. It is important to note that even though most TCP implementations use the sliding window algorithm, it is not required by the TCP specifications. mIP allows only a single TCP segment per connection to be acknowledged at any given time. The mIP implementation with the proposed global buffer data model requires far less state information and still has interoperability to other hosts even if the sliding window mechanism does not exist.

Table 4. TCP implementations.

TCP Requirements	RFC 1122 (MUST)	mIP
Flow control	0	0
Urgent Data	0	x
TCP Options	0	x
TCP Checksums	0	0
Multiple Connections	0	0(single)
RTT (timer)	0	x
Retransmissions	0	x
Congestion control	0	x
Error Control	0	x
Receiving ICMP Messages from IP	0	0
Application Interface (for HTTP)	0	0

If data has been lost in the network, the application will have to resend the data. However, the retransmission operation is not considered in mIP.

TCP continuously estimates the current Round-Trip Time (RTT) of every active connection in order to find a suitable value for the retransmission time-out. In order to estimate RTT estimation, TCP requires a periodic timer. Karn's algorithm [15] is used to ensure that retransmissions do not skew the estimates. However, mIP does not support RTT estimation for retransmission. Table 4 shows the items implemented for the TCP protocol.

The congestion control mechanism is used to limit the number of simultaneous TCP segments in the network. Our mIP does not need to control the congestion mechanism because only one in-flight TCP segment per connection is assumed.

We do not implement in the case of TCP's urgent data mechanism because our implementations already use an asynchronous event based API.

3.4. Web server program interface

If the packet contains data that is to be delivered to the web server, the web server is invoked by the means of a tight couple call. If the incoming packet acknowledges previously sent data, the connection state is updated and the application is informed, allowing it to send out new data.

We have chosen an event driven interface where the web server is invoked in response to certain events. TCP/IP stacks buffer the transmitted data in memory until the data has been successfully delivered to the other end of the connection. The data must be buffered in memory while waiting for an acknowledgment. The event based interface fits well in the event based structure used by operating systems such as MicroC/OS.

4. RESULTS

We used an isolated and standalone network for our test to eliminate unwanted network traffic. For our experiments we connected a 466 MHz, Intel Celeron 128MB PC running Dummynet to a MiniWeb board [7] through a dedicated 10 Megabit/second Ethernet network. The MiniWeb board has three different versions that are an embedded system equipped with a RealTek RTL8019AS Ethernet controller. The hardware details for each of the MiniWeb boards are as follows:

1) TS80C32 CPU, 11.0592 MHz, 256Byte RAM, 64K Byte EPROM, no Operating System

2) Atmel Atmega128 AVR microcontroller, 11.0592 MHz, 32 KBytes RAM, 128 KBytes ROM, MicroC/OS

3) Intel386Ex CPU, 33.0 MHz, 128 KByte RAM, 1MByte FlashROM under MicroC/OS

We can build a TCP/IP Protocol Test system using an embedded target. Fig. 3 describes the setup configuration. By loading Dummynet [12] on the host PC, we can easily run a simple network simulator.

Fundamental to TCP's time-out and retransmission is the measurement of the round-trip time experienced on a given connection. We expect this can transform over time, as routes might alter and as network traffic changes. TCP should track these changes and modify its time-out accordingly [15].

The host PC with IP address 192.168.0.55 sends the ping packets to the MiniWeb target. The target board with IP address 192.168.0.51 replies to the ICMP ping request from the host. We measured RTT using Dummynet, where ICMP packet data is 64bytes and packet is repeated 300 times (i.e., ping –c 300 –s 64 192.168.0..51). For Dummynet, we utilized a PicoBSD system version with FreeBSD. Table 5 presents the results. RTT for 8032 microcontroller appears slower because the microcontroller uses external RAM, where it has smaller internal RAM less than 1 packet size.

The mIPl with one global data buffer does not need to copy data between the TCP/IP stacks. Also, there is no context switch between the TCP/IP stacks for an event based operation. The TCP/IP processing is dominated by the copying of data from the network device to host memory, and checksum calculation. Table 6 summarizes the features for the embedded TCP/IP implementation.

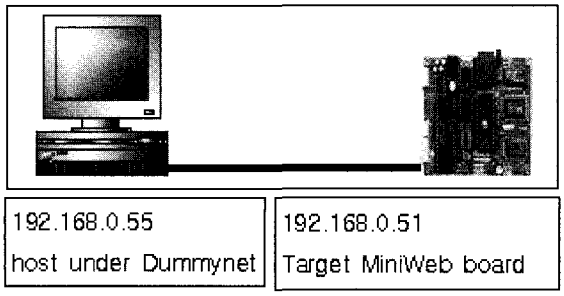


Fig. 3. Test system configuration.

The ping utility can be useful for testing whether TCP/IP is installed and configured properly. If this works, it indicates that the computer is able to route packets to itself. This is reasonable assurance that the IP layer is setup correctly. You could also ask someone else running TCP/IP for their IP address and try pinging them.

The correctness for the embedded mIP is verified using the CommView software [14] of TamoSoft Inc. Some state transitions during TCP connection are also observed correctly. Fig. 4 shows an example for the verification for the TCP protocol correctness.

Memory usage depends heavily on the applications of the particular device in which the implementations are to be run. It is possible to run the mIP with as little as 200 bytes of RAM, but such a configuration will provide extremely low throughput. The 1165 bytes of additional on-chip memory for the 8032 microcontroller is required as external data memory.

Table 5. Test results for RTT measurements.

RTT (ms)	80C32	Atmega128 (MicroC)	Intel386Ex (MicroC)	Pentium 4 (Windows XP)
min/	67.76/	6.97/	1.26/	0.39/
avg/	70.35/	32.43/	11.31/	0.51/
max/	72.86/	67.83/	24.81/	0.71/
std	1.45	14.72	5.78	0.08

(Here, Intel Pentium 4 with 1.8GHz, 512 MByte RAM is under Microsoft Windows XP)

Table 6. TCP/IP features implemented by mIP.

Features		mIP
Network Driver		o
IP	ARP	o
	IP fragment reassembly	x
	IP options	x
	IP checksum	o
	Multiple interfaces	x(no need)
	ICMP	o
TCP	UDP	o
	UDP checksum	o
	Multiple connections	x
	TCP options	x
	Variable TCP MSS	x
	TCP checksum	o
	RTT estimation	x
	TCP flow control	o(
	Congestion control	x(no need)
	Out-of-sequence data	x
	TCP urgent data	x(no need)
HTTP		o

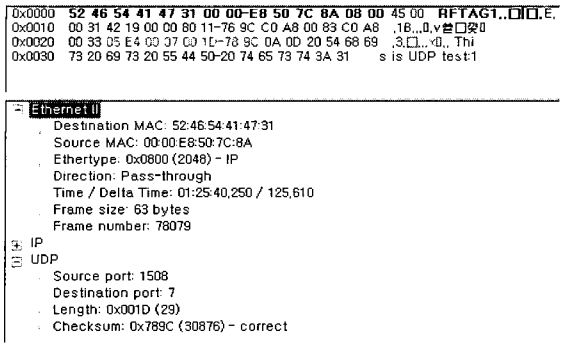


Fig. 4. TCP/IP state transition by CommView.

Table 7. Memory usage for mIP.

Functions	Size (bytes) (for 8032)		Size (bytes) (Atmega 128)		Size (bytes) (80386)	
	CODE	DATA (RAM)	CODE	DATA (RAM)	CODE	DATA (RAM)
Checksum, IP, ICMP, UDP, ARP, Network Driver, HTTP	5260	Internal = 200 External = 1165	3759	1142	4581	1081

Because the protocol implementations in mIP are tightly coupled, the individual sizes of the implementations are not calculated. The code for each of the three platforms was compiled using IAR ICC8051, CodeVisionAVR, and Borland BCC, respectively. Table 7 summarizes the code size for the implementations.

Finally, operation of the embedded RF tag reader is shown by a web browser in Fig. 5. A tag ID reading through the web based tag reader is shown as SW3437. Fig. 6 presents the web based RF tag reader developed by our research lab.

5. CONCLUSIONS

We have implemented the simplified TCP/IP stack, which can support a web-based control system, by using a global common data buffer model for each of the network layers, and have shown that it is possible to implement TCP/IP protocol suite within 8 or 32-bit CPU. We have studied how the code size and RAM usage of a TCP/IP implementation can be reduced and how some TCP/IP mechanisms can be sacrificed. Finally, we have indicated that the web based RF tag reader applications can communicate without any interoperability problems over the Internet.

The main contribution of our work is to implement any subset of TCP/IP stacks that is small enough in terms of code size and memory with only a few kilobytes of RAM based on the 8032, Atmega128, and 80386 processors. We may need additional memory for the full TCP/IP stack.

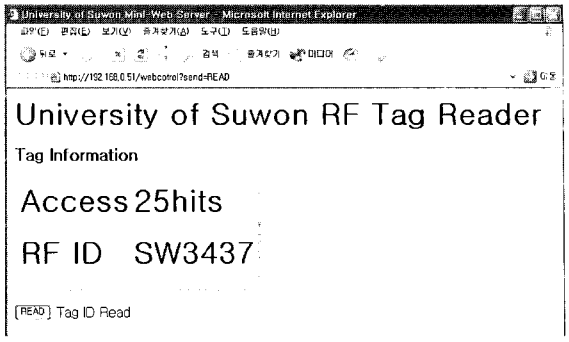


Fig. 5. Web based RF tag reader.

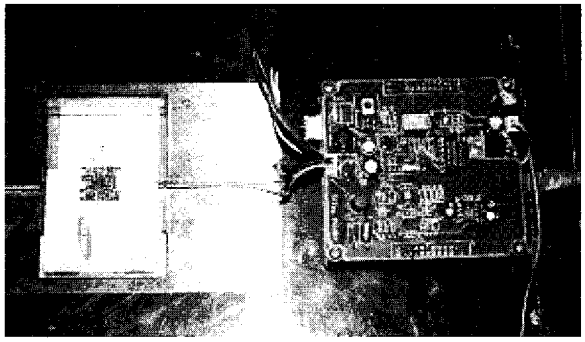


Fig. 6. Web based RF tag reader.

REFERENCES

[1] R. Braden, *Requirements for Internet Hosts - Communication Layers,* RFC 1122, Internet Engineering Task Force, October 1989.

[2] J. Bentham, *TCP/IP Lean: Web Servers for Embedded Systems,* CMP Books, October 2002.

[3] A. Dunkels, "Full TCP/IP for 8-bit architecture," *Proc. of the First International Conference on Mobile Systems, Applications and Services,* San Francisco, May 2003.

[4] H. Shrikumar. "IPic - a match head sized web server," http://www-ccs.cs.umass.edu/~shri/iPic.html.

[5] Atmel Corporation, "Embedded web server. AVR 460," http://www.atmel.com, January 2001.

[6] D. D. Clark, J. Romkey, V. Jacobson, H. Salwen, "An analysis of TCP processing overhead," *IEEE Communications Magazine,* vol. 27, no. 6, pp. 23-29, June 1989.

[7] S. Shon, *MiniWeb Ethernet Kit,* Embedded system Lab., University of Suwon.

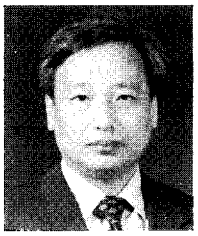
[8] A. Dunkels, "uIP - a TCP/IP stack for 8- and 16-bit microcontrollers," http://dunkels.com/adam/uip/.

[9] B. A. Forouzan, *TCP/IP Protocol Suite,* McGraw-Hill International, pp. 297-299, 2000.

[10] J. Kay and J. Pasquale, "The importance of non-data touching processing overheads in TCP/IP," *Proc. of the ACM SIGCOMM '93 Symposium,* pp. 259-268, September 1993.

[11] C. Partridge, J. Hughes, and J. Stone, "Performance of checksums and CRCs over real data,"

Proc. of ACM SIGCOMM '95, pp. 68-76, 1995.
[12] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review,* vol. 27, no. 1, pp. 31-41, 1997.
[13] A. Rijsinghani, *Computation of the Internet Checksum via Incremental Update,* RFC 1624, Internet Engineering Task Force, May 1994.
[14] TamoSoft inc. CommView®, http://www.tamos.com/products/commview/.
[15] P. Karn and C. Partridge, "Improving round-trip time estimates in reliablie transport protocols," *Proc. of the SIGCOMM '87 Conference,* Stowe, Vermont, August 1987.



Sugoog Shon received the Ph.D. degree in Electrical and Computer Engineering from the University of Texas, Austin in 1996. His research interests are embedded systems and their applications.