uIP - A Free Small TCP/IP Stack

Adam Dunkels
adam@dunkels.com

November 25, 2001

Abstract

This document describes the uIP TCP/IP stack. The uIP TCP/IP stack is an extremely small implementation of the TCP/IP protocol suite and is intended for embedded systems running low-end 8 or 16-bit microcontrollers. The code size of uIP is an order of magnitude smaller than other generic TCP/IP stacks today.

In this document, the philosophy behind uIP is described and detailed information of the implementations of the individual protocols is presented. The code size and memory usage in uIP is discussed. Finally, information on how to use and configure uIP in an actual system is given.

The uIP code and new versions of this document can be downloaded from the uIP homepage at `http://dunkels.com/adam/uip/`.

This document describes uIP version 0.5.

1 Introduction

In recent years, the interest of connecting even small devices to an existing IP network such as the global Internet has increased. In order to be able to communicate over the Internet, an implementation of the TCP/IP protocol stack is needed. uIP is an implementation of the most important parts of the TCP/IP protocol suite. The goal of the uIP implementation is to reduce both the code size and the memory usage to a minimum. uIP is an order of magnitude smaller than any existing generic TCP/IP stack today. uIP is written in the C programming language and is free to distribute and use for both non-commercial and commercial use.

In an ordinary TCP/IP stack, memory is often used to buffer data while waiting for an acknowledment signal that the data has successfully been delivered. In case a data packet is lost, the data and has to be retransmitted. Typically, the data is buffered in RAM, even though the application may be able to quickly regenerate the data if a retransmission is needed. For instance, an HTTP server serving mostly static or semi-static pages does not need to buffer the static content in a RAM buffer. Instead, the HTTP server can easily reproduce the data if a packet is lost. The data is simply read it back from the original location. uIP takes advantage of this by allowing the application to take part in doing retransmissions.

This document is structured as follows. Section 2 describes the protocols that are implemented in uIPand Section 3 discusses the code size and memory

usage in uIP. Section 4 describes how uIP interfaces with device drivers, timers, and applications. Section 5 covers how uIP is configured, and finally Section 6 describes the architecture specific portions of uIP.

2 Protocols

uIP implements three of the basic protocols in the TCP/IP protocol suite; IP [Pos81b], ICMP [Pos81a] and TCP [Pos81c]. Underlying protocols such as SLIP, PPP or ARP can be implemented as a device driver under uIP. Higher level protocols such as HTTP, FTP or SMTP can be implemented as an application running on top of uIP.

2.1 Internet Protocol — IP

The IP layer code in uIP has two responsibilities: verifying the correctness of the IP header of incoming packets and demultiplexing the packet between the ICMP and TCP protocols. The IP layer code is very simple and consists of 9 `if` statements.

2.2 Internet Control Message Protocol — ICMP

In uIP, only one type of ICMP messages are implemented: the ICMP echo message. ICMP echo messages are frequently used by the well known **ping** program to check if a host is online. In uIP, ICMP echos are processed in a very simple fashion. The ICMP type field is changed from the "echo" type to the "echo reply" type, and the ICMP checksum is adjusted accordingly. Next, the IP addresses in the IP header are exchanged and the packet is sent back to the original sender.

2.3 Transmission Control Protocol — TCP

In order to reduce memory usage, the TCP in uIP does not implement a sliding window for sending and receiving data. Incoming TCP segments are not buffered by uIP, but must be processed immediately by the application. Note that this does not prevent the application from buffering the data by itself. For outbound data, uIP cannot have more than one outstanding TCP segment per connection.

2.3.1 Connection state

In uIP, the complete state of each TCP connection consists of the local and remote TCP port numbers, the IP address of the remote host, three sequence numbers, and the value of the retransmission timer. In addition to this, each connection also may hold some application state (see Section 4.3.3). The three sequence numbers are the sequence number of the byte that is expected to be received next, the sequence number of the first byte in the last segment sent, and the sequence number of the next byte to be sent. The connection state is represented by the `uip_conn` structure that can be seen in Figure 1.

```
struct uip_conn {
  u8_t tcpstateflags; /* TCP state and flags. */
  u16_t lport, rport; /* The local and the remote port. */
  u16_t ripaddr[2];   /* The IP address of the remote peer. */
  u8_t rcv_nxt[4];    /* The sequence number that we expect to receive
                         next. */
  u8_t snd_nxt[4];    /* The sequence number that was last sent by
                         us. */
  u8_t ack_nxt[4];    /* The sequence number that should be ACKed by
                         next ACK from peer. */
  u8_t timer;         /* The retransmission timer. */
  u8_t nrtx;          /* Counts the number of retransmissions for a
                         particular segment. */

  u8_t appstate[UIP_APPSTATE_SIZE];
};
```

Figure 1:

An array of `uip_conn` structures is used to hold all connections in uIP. The size of the array is equal to the maximum amount of simultaneous TCP connections and is configured at compile time (see Section 5).

2.3.2 Input processing

TCP input processing starts with verifying the TCP checksum. If the checksum is found to be correct, the source and destination port numbers and IP addresses are used to demultiplex the packet between the currently active TCP connections. If no active connection that matched the incoming packet was found, the packet must be an incoming connection request. Hence the packet must have the SYN flag set and if not, the packet is dropped and a reset packet is sent in response. If the packet had the SYN flag set, the destination port in the packet's TCP header is matched against the list of passively open (listening) ports.

If a listening port is found, the array of `uip_conn` structures is scanned for any inactive connections. If one is found, this `uip_conn` is filled in with the port numbers and IP addresses of the new connection. The initial TCP sequence number is set to 0 in this version of uIP. The TCP specifications require the initial sequence number to be more or less random, and this behavior may be changed in future versions of uIP. Finally, a response packet is sent to acknowledge the opened connection. This packet carries the TCP MSS (Maximum Segment Size) option informing the remote host of the currently configured TCP MSS.

Should the incoming packet be destined for an already active connection, the sequence number of the packet is checked with the next expected sequence number from the remote host (the `rcv_nxt` variable in the `uip_conn` structure shown in Figure 1). If the sequence number is not the next expected, the packet is dropped and an ACK is sent to indicate the next sequence number that is expected. Next, the acknowledgment number in the incoming packet is checked

to see if it acknowledges any outstanding data for the connection. If it does, the application will be made aware of this fact.

When the sequence and acknowledgment numbers have been checked, the packet will be handled differently depending on the current TCP state. If the connection is in the SYN-RCVD TCP state and the incoming packet acknowledged the previously sent SYNACK packet, the connection will enter the ESTABLISHED state, and the application is made aware of the fact that the connection has been fully established (see Section 4.3.1 for more information on this).

For connections in the ESTABLISHED state, the global variables `uip_len` and `uip_flags` will be setup to prepare for the application call (see Section 4.3.1 for a description of how those variables are used). If the incoming packet included new data for the application or if it acknowledged data previously sent by the application, the application function is called. When it has returned, the `uip_flags` variable is checked to see if the application wishes to close (or abort) the connection. If so, the appropriate steps are taken for closing (or aborting) the connection.

2.3.3 Output processing

Output processing is straightforward and quite a lot simpler than the input processing. Basically, all fields of the TCP and IP headers are filled in with values from the `uip_conn` structure and the TCP and IP checksums are calculated. When the `uip_process()` function returns, the packet is sent by the network device driver.

2.3.4 Retransmissions

Retransmissions are handled when uIP is invoked by the periodic timer (see Section 4.2). Connections that have outstanding data (i.e., data that is sent but not yet acknowledged) is flagged by the `UIP_OUTSTANDING` bit in the `tcpstateflags` variable in the `uip_conn` structure (Figure 1). For those connections, the `timer` variable is increased until it reaches the RTO value for the connection. This is calculated by shifting the configurable value `UIP_RTO` n times left, where n is the `nrtx` variable in the `uip_conn` structure. When this value is reached, the outstanding data is assumed to be lost in the network and should be retransmitted. If the number of retransmissions of a particular segment exceeds a configurable threshold, the connection is dropped. In this version of uIP, the application is not notified of this. In future uIP versions, this will be changed so that the application will be given this information.

Unlike other TCP/IP stacks, uIP requires assistance from the application to do a retransmission. uIP does not buffer the previously sent data, but lets the application decide wether the data was important enough to buffer or if it could be reproduced on the fly for a retransmission.

When uIP has concluded that data should be retransmitted, the application function is called with the special `UIP_REXMIT` flag set (see Section 4.3.1).

2.3.5 TCP resets

The TCP specification requires that a packet with the RST (reset) flag set must kill a connection if the sequence and acknowledgment numbers in the TCP

header falls within the current receive window for the connection. In order to reduce the code size, uIP does not strictly adhere to this. Instead, if a packet with the RST flag set arrives in a connection, the connection will be killed regardless of the values of the sequence and acknowledgment numbers. This might become a problem in particular in combination with the non-randomness of the initial sequence numbers. Therefore, this behavior might be revised in future versions of uIP.

3 Size

uIP is designed to be small both in terms of compiled code size and memory usage. Memory usage is configurable and depends on the maximum number of active connections that should be possible as well as the maximum size of incoming and outgoing packets. The size of the compiled code depends on the C compiler that is used and the CPU architecture for which uIP is compiled. As a rule of thumb, memory usage is on the order of hundreds of bytes, and code size is a few kilobytes.

3.1 Memory usage

Memory usage is on the order of hundreds of bytes, and is configured at compile time. The memory usage depends on the size of the packet buffer and on the maximum number of simultaneous connections. With a packet buffer size of 240 bytes and a maximum of 10 simultaneous connections, uIP uses a little more than 300 bytes of memory.

3.2 Stack usage

In order to keep call stack usage down to a minimum, uIP do not nest function calls. Since functions are not nested or called recursively, function parameters need not be passed on the stack. Instead, uIP uses global variables to pass information between functions.

4 Interfacing uIP

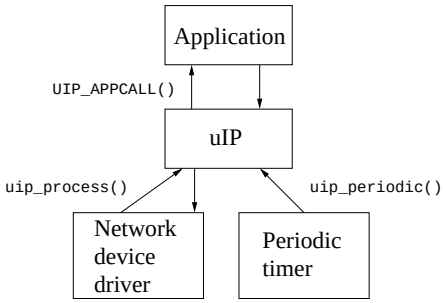

Figure 2: How uIP fits into a system.

An overview of how uIP fits into a system can be seen in Figure 2. Processing is initiated either by the network device driver or by the periodic timer. When an IP packet is received by the network device driver, control is transfered to uIP by the function call `uip_process()`. This function may be called directly from the device driver or from a supervisory function. In `uip_process()`, uIP will process the packet, possibly pass it over to the application, and return to the caller. When `uip_process()` returns, the network device driver may have a packet that it should send. This is shown in Figure 3.

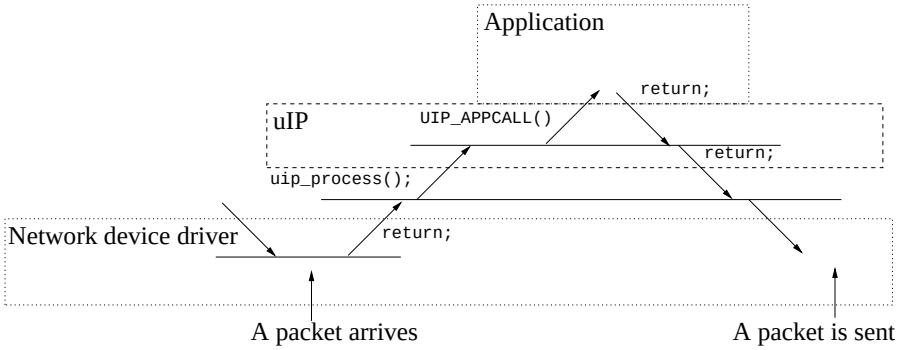

Figure 3: The function calls in the network device driver, uIP, and the application when a packet arrives and a reply is sent.

4.1 The uIP/device driver interface

When the device driver as a reaction to an incoming IP packet calls `uip_process()`, uIP expects the incoming packet to be present in the global array called `uip_buf`. The length of the packet should be placed in the global variable `uip_len`. The reason for using global variables instead of function arguments to communicate this information is that global variables are slightly more efficient in terms of memory usage and execution time.

When the `uip_process()` function returns, an outgoing packet is present in the same `uip_buf` array. The size of the packet is kept in the `uip_len` variable. If `uip_len` is equal to zero, no outgoing packet should be sent when `uip_process()` returns.

4.2 The uIP/periodic timer interface

The periodic timer is used for driving all uIP internal timer events such as packet retransmissions. When the periodic timer fires, the uIP function `uip_periodic()` should be called once per TCP connection. The connection number is passed as an argument to the `uip_periodic()` function.

The `uip_periodic()` function may have to retransmit a packet. If so, the global variables `uip_len` and `uip_buf` are used in the same manner as when processing incoming packets.

Figure 4 shows an example code snippet that shows how to interface a periodic timer with uIP. The constant `UIP_CONNS` is the amount of configured

```
for(i = 0; i < UIP_CONNS; i++) {
  uip_periodic(i);
  if(uip_len > 0)
    netdev_send();
}
```

Figure 4: Example code for interfacing the periodic timer with uIP.

TCP connections. If the variable `uip_len` is larger than zero after the function `uip_periodic()` has returned, a packet that should be sent is present in the global array `uip_buf`. In this particular example, the function `netdev_send()` is part of the network device driver and will send the contents of the `uip_buf` array out on the network.

4.3 The uIP/application interface

The BSD socket interface used in most operating systems is not suitable for small systems since it forces a thread based programming model on the application programmer. 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 keeping per-thread state. The execution time overhead in task switching also contributes to this. Small systems may not have enough resources to implement such a multithreaded environment, and therefore an application interface which requires this would not be suitable for uIP.

Instead, uIP provides an event based programming model to the application. uIP calls the application when data is received, when data has been successfully delivered to the other end of the connection, when a new connection has been set up, or when data has to be retransmitted. The application is also periodically polled for new data to send.

uIP is different from other TCP/IP stacks in that it requires help from the application when doing retransmissions. Other TCP/IP stacks would buffer the transmitted data in memory until the data is known to be successfully delivered to the remote end of the connection. If the data needs to be retransmitted, the application would not be notified. Using this approach, the data has to be buffered in memory while waiting for an acknowledgment, even if the application might be able to quickly regenerate the data if a retransmission would have to be made. In order to reduce memory usage, uIP utilizes the fact that the application often is able to regenerate sent data and lets the application take part in retransmissions.

uIP and the application communicates through a number of global variables and one function call. uIP is the initiator of all communication with the application.

4.3.1 Receiving and sending data or control information

The application must implement a function, `UIP_APPCALL()`, that uIP calls whenever one of six things happen:

1. A packet arrives that acknowledges previously sent data.

2. A packet arrives with new data for the application.
3. The retransmission timer expires.
4. The periodic polling timer expires.
5. The remote host has closed the connection.
6. A remote host has connected to a listening connection.

A bit pattern is set in the global variable `uip_flags` to indicate which of the above conditions that occurred. The bits that can be set are `UIP_ACKDATA`, `UIP_NEWDATA`, `UIP_REXMIT`, `UIP_POLL`, `UIP_CLOSED`, and `UIP_ACCEPT`, respectively. Note that the first two conditions (`UIP_ACKDATA` and `UIP_NEWDATA`) can occur simultaneously.

When `UIP_NEWDATA` is set, the pointer `uip_appdata` points to the data sent by the remote host. The global variable `uip_len` contains the length of the data. The `UIP_ACKDATA` flag is set to indicate that the remote host has acknowledged all previously sent data. This serves as an indication that new data now can be sent.

The `UIP_REXMIT` flag is set when the retransmission timer has expired, meaning that the previously sent data must be retransmitted. It is the responsibility of the application to reproduce the data for the retransmission. This is done in the same way as when data is sent normally (see below).

When no data is outstanding, uIP will periodically poll the application. This is done by calling the application function with the `UIP_POLL` flag set in the `uip_flags` variable. The purpose of the polling is to allow the application to send data even when no data or acknowledgments are coming from the remote host. Data is sent in the same way as when sending data in response to other events.

When the remote host has closed the connection, the application is informed of this by the `UIP_CLOSED` bit in the `uip_flags` variable. Notice that it is not possible for the application to continue sending data after the other host has closed the connection; the connection is automatically closed by uIP.

The last event that can occur, is that a connection in the LISTEN state (i.e., a connection that is waiting for others to connect to it) can be connected to a remote host. In this case, the application is called with the `UIP_ACCEPT` flag set.

Whenever the application is called, uIP sets the global variable `uip_conn` to point to the `uip_conn` structure for the current connection. This can be used to distinguish between different services. A typical use would be to inspect the `uip_conn->lport` (the local TCP port number) variable to which service the connection should provide. For instance, an application might decide to act as an HTTP server if the value of `uip_conn->lport` is equal to 80 and act as a TELNET server if the value is 23. This `uip_conn` structure also can be used to keep state information for the application; see Section 4.3.3.

4.3.2 Sending data or control information

Before the function `UIP_APPCALL()` returns, the `uip_flags` variable and the `uip_appdata` array together with the `uip_len` variable can be used to send data or control information to the remote host. In order to send data to the remote end of a connection, the application sets in the `uip_appdata` pointer to point

to the data that should be sent and sets the `uip_len` variable to the size of the data to be sent. Notice that this behavior changed from with version 0.4 of uIP.

In order to close a connection, the `UIP_CLOSE` bit should be set in the `uip_flags` variable. To abort a connection, the `UIP_ABORT` bit can be set in the same variable. This will cause the connection to be aborted and a reset segment to be sent to the remote host.

4.3.3 Application state

The application might need to keep certain state for each connection. For this purpose, the `appstate` variable in the `uip_conn` structure (Figure 1) is used. The application can access this variable through the global `uip_conn` pointer.

5 Configuring uIP

The configuration for uIP is kept in a single `.h`-file called `uipopt.h`. This file contains definitions for the IP address of the uIP node, the number of possible simultaneous connections, the maximum number of listening (passively opened) connections, the size of the `uip_buf` array as well as options that are CPU architecture and C compiler specific. There are well commented example `uipopt.h` files in the uIP distribution.

The configuration parameters are:

UIP_APPCALL the name of the application function. This function must return void and take no arguments (i.e., C type `void appfunc(void)`).

UIP_CONNS The maximum number of simultaneously active connections.

UIP_LISTENPORTS The maximum number of simultaneously listening TCP ports.

UIP_BUFSIZE The size of the buffer that holds incoming and outgoing packets.

UIP_IPADDR The IP address of this uIP node.

UIP_LLH_LEN The link level header length; this is the offset into the `uip_buf` where the IP header can be found. For Ethernet, this should be set to 14.

UIP_TCP_MSS The TCP maximum segment size. This should be set to at most `UIP_BUFSIZE` - 40.

UIP_TTL The IP TTL (time to live) of IP packets sent by uIP.

UIP_RTO The retransmission timeout counted in timer pulses (i.e., the speed of the periodic timer).

UIP_APPSTATE_SIZE The size of the application-specific state stored in the `uip_conn` structure. See Section 4.3.3 for more information.

UIP_STATISTICS A flag to indicate whether statistics gathering code should be compiled in.

UIP_LOGGING A flag which indicates if code for logging events should be compiled in. *This option is not supported in the current version of uIP.*

Furthermore, since the number of bits in the standard C types `short`, `int`, and `long` differ between compilers, the `uipopt.h` file defines the two types `u8_t` and `u16_t`. Those are the basic 8- and 16-bit unsigned integer types, respectively. Finally, the `uipopt.h` file defines the C macro `BYTE_ORDER` to be either `LITTLE_ENDIAN` or `BIG_ENDIAN` to reflect the byte order of the CPU architecture.

6 Architecture specific functions

While the IP, ICMP and TCP protocol implementations in uIP are implemented in a single C function (the `uip_process()` function), they need the help of four support functions. The support functions implement 32-bit additions and checksum calculations. With uIP version 0.4, the support function implementations were split from the actual protocol implementations in order to make it easier to handcraft the support functions in assembler. Since the support functions called frequently, there is a substantial gain in making those functions run as fast as possible.

The four support functions are the two functions for calculating the IP and TCP checksums, `uip_ipchksum()`, `uip_tcpchksum()`, and the two functions for performing 32-bit additions of TCP sequence numbers, `uip_add_ack_nxt()`, and `uip_add_rcv_nxt()`. The uIP distribution contains sample C implementations of the support functions.

The `uip_ipchksum()` calculates and returns the Internet checksum [BBP88] of the IP header but without doing the bit-wise negation of the checksum. The IP header can be found in the first 20 bytes of the `uip_buf` array.

The `uip_tcpchksum()` function calculates the TCP checksum. The TCP checksum is the Internet checksum of the TCP header and the TCP data. This function is somewhat complicated by the fact that the TCP header and the TCP data may be located in different memory locations. The TCP header can be found 20 bytes from the start of the `uip_buf` array (i.e., at `&uip_buf[20]`) and the `uip_appdata` pointer points to start of the TCP data. The size of the TCP data can be calculated by subtracting the size of the IP and TCP headers from the size of the entire packet. The size of the packet is contained in the global `uip_len` variable. Since uIP does not support IP or TCP options in the data stream, the total size of the IP and TCP header is 40 bytes.

References

[BBP88] R. Braden, D. Borman, and C. Partridge. Computing the internet checksum. RFC 1071, Internet Engineering Task Force, September 1988.

[Pos81a] J. Postel. Internet control message protocol. RFC 792, Internet Engineering Task Force, September 1981.

[Pos81b] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[Pos81c] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[tea01] The GCC team. The gnu c compiler web page. Web page, September 2001.
URL: `http://www.gnu.org/software/gcc/`

[vB01] U. von Bassewitz. cc65 - a freeware c compiler for 6502 based systems. Web page, September 2001.
URL: `http://www.cc65.org/`