uIP - A Free Small TCP/IP Stack

Adam Dunkels
adam@dunkels.com

January 15, 2002

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 intended for embedded systems running low-end 8 or 16-bit microcontrollers. The code size and RAM requirements of uIP is an order of magnitude smaller than other generic TCP/IP stacks today.

The uIP stack uses an event based programming model to reduce code size and RAM usage. The callback based interface between the TCP/IP stack and the application program is described in this document. The interface between the lower layers of the system and uIP are also covered. Details about the actual protocol implementations are given. Finally, numerous examples of uIP appilication programs are included.

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 0.6.

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 keep 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 other 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 from ROM does not need to buffer the static content in a RAM buffer. Instead, the HTTP server can easily reproduce the data from ROM if a packet is lost. The data is simply read back from its 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 how to use uIP, both from the system's and the application's standpoints. Details about the protocol implementations are discussed in Section 3. Section 4 covers how uIP is configured, and Section 5 describes the architecture specific portions of uIP. Finally, Section 6 provides a few examples of application programs for uIP.

2 Interfacing uIP

uIP can be seen as a code library that provides certain functions to the system. Figure 1 shows the relations between uIP, the underlying system and the application program. uIP provides three functions to the underlying system, `uip_init()`, `uip_input()`, and `uip_periodic()`. The application must provide a callback function to uIP. The callback function is called when network or timer events occur. uIP provides the application with a number of functions for interacting with the stack.

Note that most of the functions provided by uIP is implemented as C macros for speed, code size efficiency, and stack usage reasons.

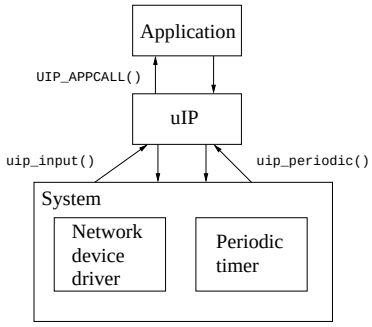

Figure 1: uIP seen as a library.

2.1 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 uses an event based programming model where the application is implemented as a C function that is called by uIP in response to certain events. 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. The application program provides only one callback function; it is up to the application to deal with mapping different network services to different ports and connections.

uIP is different from other TCP/IP stacks in that it requires help from the application when doing retransmissions. Other TCP/IP stacks 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 stack takes care of the retransmission without notifying the application. With 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 has to be made. In order to reduce memory usage, uIP utilizes the fact that the application may be able to regenerate sent data and lets the application take part in retransmissions.

2.1.1 uIP/application events

The application must be implemented as a C function, `UIP_APPCALL()`, that uIP calls whenever an event occurs. Table 1 lists the possible events and for each events show the corresponding test function. The test functions are used to distinguish between different events. The functions are implemented as C macros that will evaluate to either zero or non-zero. Note that certain events can happen in conjunction with each other (i.e., new data can arrive at the same time as data is acknowledged).

Table 1: uIP application events and the corresponding test functions.

A packet has arrived that acknowledges previously sent data.	`uip_acked()`
A packet has arrived with new data for the application.	`uip_newdata()`
A remote host has connected to a listening port.	`uip_connected()`
A connection has been successfully set up to a remote host.	`uip_connected()`
The retransmission timer expires.	`uip_rexmit()`
The periodic polling timer expires.	`uip_poll()`
The remote host has closed the connection.	`uip_closed()`
The connection has been aborted by the remote host.	`uip_aborted()`
The connection has been aborted due to too many retransmissions.	`uip_timedout()`

When the application is called, uIP sets the global variable `uip_conn` to point to the `uip_conn` structure (Figure 5) 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) to decide 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.

2.1.2 Receiving data

If the uIP test function `uip_newdata()` evaluates to 1, the remote host of the connection has sent new data. The `uip_appdata` pointer point to the actual data. The size of the data is obtained through the uIP function `uip_datalen()`. Since the data is not buffered, the application must act immediately upon it.

2.1.3 Sending data

The application sends data by using the uIP function `uip_send()`. The `uip_send()` function takes two arguments; a pointer to the data to be sent and the length of the data. If the application needs RAM space for producing the actual data that should be sent, the packet buffer (pointed to by the `uip_appdata` pointer) can be used for this purpose.

The application can send only one chunk of data at a time on a connection. It is therefore not possible to call `uip_send()` more than once per application invocation; only the data from the last call will be sent. Note that since the `uip_send()` call will change certain global variables, it should not be called until just before the application function returns.

2.1.4 Retransmitting data

If data has been lost in the network, the application will have to resend the data. uIP keeps track of if the data has been received or not, and will inform the application when the data is perceived to be lost. If the test function `uip_rexmit()` is true, the application should retransmit the last data it sent. Retransmission is done in the same way as ordinary transmissions, i.e., with `uip_send()`.

2.1.5 Closing connections

The application closes the current connection by calling the `uip_close()`. This will cause the connection to be cleanly closed. In order to indicate a fatal error, the application might want to abort the connection and does so by calling the `uip_abort()` function.

If the connection has been closed by the remote end, the test function `uip_closed()` is true. The application may then do any necessary cleanups.

2.1.6 Reporting errors

There are two fatal errors that can happen to a connection, either that the connection was aborted by the remote host, or that the connection retransmitted the last data too many times and has been aborted. uIP reports this by calling the application function. The application uses the two test functions `uip_aborted()` and `uip_timedout()` to test for those error conditions.

2.1.7 Polling

When a connection is idle, uIP polls the application every time the periodic timer fires. The application uses the test function `uip_poll()` to check if it is being polled by uIP.

2.1.8 Listening ports

uIP maintains a list of listening TCP ports. A new port is opened for listening with the `uip_listen()` function. When a connection request arrives on a listening port, uIP creates a new connection and calls the application function. The test function `uip_connected()` is true if the application was invoked because a new connection was created.

2.1.9 Opening connections

As of version 0.6 of uIP, new connections can be opened from within uIP by using the function `uip_connect()`. This function opens a new connection to a specified IP address and port and returns a pointer to the `uip_conn` structure for the new connection. If there are no free connection slots, the function returns NULL. For convenience, the function `uip_ipaddr()` may be used to pack an IP address into the two element 16-bit array used by uIP to represent IP addresses.

Two examples of usage are shown in Figures 2 and 3. The first example shows how to open a connection to TCP port 8080 of the remote end of the current connection. If there are not enough TCP connection slots to allow a new connection to be opened, the `uip_connect()` function returns NULL and the current connection is aborted by `uip_abort()`. The second example shows how to open a new connection to a specific IP address. No error checks are made in this example.

```
void connect_example1_app(void) {
   if(uip_connect(uip_conn->ripaddr, 8080) == NULL) {
      uip_abort();
   }
}
```

Figure 2: Opening a new connection to port 8080 at the remote end of the current connection.

```
void connect_example2(void) {
   u16_t ipaddr[2];

   uip_ipaddr(ipaddr, 192,168,0,1);
   uip_connect(ipaddr, 8080);
}
```

Figure 3: Opening a connection to port 8080 at host 192.168.0.1.

2.1.10 Flow control

uIP provides access to the TCP flow control through the functions `uip_stop()` and `uip_restart()`. Consider an application that downloads data from a server onto a slow device such as a disk drive. If the job queue for the disk drive is full,

the application is not prepared to accept any more data from the server until the queue has been drained. The function `uip_stop()` can be used to assert the flow control and stop the remote host from sending data. When the application is ready for more data, the function `uip_restart()` is used to tell the remote end to start sending data again. The function `uip_stopped()` can be used to check if the current connection is stopped.

2.2 The uIP/system interface

From the system's standpoint, uIP consists of three C functions, `uip_init()`, `uip_input()`, and `uip_periodic()`. The `uip_init()` function is used to initialize the uIP stack and is called during system startup. The function `uip_input()` is called when the network device driver has read an IP packet into the packet buffer, and `uip_periodic()` is called periodically, typically once per second. It is the responsibility of the system to call these uIP functions.

2.2.1 The uIP/device driver interface

When the device driver has placed an incoming IP packet in the packet buffer (the `uip_buf`), the system should call the `uip_input()` function. This function will process the packet and call the application if necessary. When the `uip_input()` function returns, an outgoing packet is placed in the packet buffer. The size of the outgoing packet is held in the global `uip_len` variable. If `uip_len` is zero, no packet is to be sent out.

2.2.2 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.

Similar to the `uip_input()` function, when the `uip_periodic()` function returns, an outbound IP packet may be placed in the packet buffer. Figure 4 shows an example code snipped for calling the `uip_periodic()` function and taking care of the outbound packet. 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.

```
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.

2.3 uIP function summary

Table 2 contains a summary of all functions that uIP provides.

Table 2: uIP function summary.

System interface	
`uip_init()`	Initialize uIP
`uip_input()`	Process an incoming packet
`uip_periodic()`	Process a periodic timer event
Application interface	
`uip_listen()`	Start listening on a port
`uip_connect()`	Connect to a remote host
`uip_send()`	Send data on the current connection
`uip_datalen()`	The size of the incoming data
`uip_close()`	Close the current connection
`uip_abort()`	Abort the current connection
`uip_stop()`	Stop the current connection
`uip_stopped()`	Find out if connection is stopped
`uip_restart()`	Restart the current connection
Test functions	
`uip_newdata()`	Remote host has sent new data
`uip_acked()`	Sent data has been acknowledged
`uip_connected()`	The current connection has just been connected
`uip_closed()`	The current connection has just been closed
`uip_aborted()`	The current connection has just been aborted
`uip_timedout()`	The current connection has just timed out
`uip_rexmit()`	Data should be retransmitted
`uip_poll()`	Application is being polled
Misc. functions	
`uip_mss()`	Obtain the MSS for the current connection
`uip_ipaddr()`	Pack IP address structure
`htons()`, `ntohs()`	Convert between host and network byte order

3 Protocol implementations

uIP implements four of the basic protocols in the TCP/IP protocol suite; ARP [Plu82], IP [Pos81b], ICMP [Pos81a] and TCP [Pos81c]. Link layer protocols such as PPP can be implemented as a device driver under uIP. Application layer protocols such as HTTP, FTP or SMTP can be implemented as an application running on top of uIP.

3.1 Address Resolution Protocol — ARP

The ARP protocol maps between IP addresses and Ethernet MAC addresses and is needed for TCP/IP operation on an Ethernet. The ARP implementation in uIP maintains a table of IP to MAC address mappings. When an IP packet is to be sent out on the Ethernet, the ARP table is consulted in order to find the MAC address to which the packet should be sent. If the IP address cannot be found in the table, an ARP request packet is sent. The request packet is broadcasted over the network and requests the MAC address for a given IP address. The host that has been given the requested IP address responds by sending an ARP reply. When uIP gets an ARP reply, the ARP table is updated.

To save memory, ARP requests for an IP address overwrites the outgoing IP packet for which the request is sent. It is assumed that the upper layers will retransmit the data that has been overwritten.

Every ten seconds, the table is refreshed and old entries are discarded. The default lifetime for an ARP table entry is 20 minutes.

3.2 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. The IP layer in uIP is greatly simplified by the fact that it does not implement IP fragmentation and reassembly.

3.3 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 **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.

3.4 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.

3.4.1 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, the value of the retransmission timer, the number of retransmissions for the last segment, and the MSS (maximum segment size) for the connection. In addition to this, each connection also may hold some application state. 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 5.

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 4).

```
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 mss;           /* The maximum segment size for the
                         connection. */
  u8_t appstate[UIP_APPSTATE_SIZE];
};
```

Figure 5: The `uip_conn` structure.

3.4.2 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 matched the incoming packet, the packet is dropped if it is not a connection request for a listing port. If the packet is a connection request for a closed port, uIP sends an RST packet in response.

If a listening port is found, the array of `uip_conn` structures is scanned for any inactive connections. If one is found, it is filled in with the port numbers and IP addresses of the new connection. If the connection request carries an TCP MSS (maximum segment size) option, it is parsed and checked against the currently configured MSS to determine the MSS of the connection. The smaller of the two is chooses. Finally, a response packet is sent to acknowledge the opened connection.

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 5). 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 state and the incoming packet acknowledged the previously sent SYNACK packet, the connection will enter the ESTABLISHED state, and the application function is invoked to inform that the connection has been fully connected. For connections in the ESTABLISHED state,

the application function is invoked if there is new data sent by the remote host, or if the remote host has acknowledged previously sent data.

When the application function returns, TCP checks if the application has any data to send. If so, a TCP/IP packet is constructed in the packet buffer.

3.4.3 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.

3.4.4 Retransmissions

Retransmissions are handled when uIP is invoked by the periodic timer (see Section 2.2.2). 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 5). For those connections, the `timer` variable is decreased. When the timer reaches zero, the last segment should be retransmitted and the application function is invoked to do the actual retransmission.

If the number of retransmissions of a particular segment exceeds a configurable threshold, the connection is dropped and an RST segment is sent to the remote end of the connection. The application function is invoked to inform it that the connection has timed out.

3.4.5 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 behavior might be revised in future versions of uIP.

4 Configuring uIP

The configuration for uIP is kept in a single `.h`-file called `uipopt.h`. This file contains not only configuration options that are project specific (such as IP address of the uIP node and the maximum number of simultaneous connections) but also architecture and C compiler specific options. The file is selfcontained and documented through comments.

5 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.

6 Example applications

This section presents a number of simple examples of uIP applications.

6.1 A very simple application

This first example shows a very simple application. The application listens for incoming connections on port 1234. When a connection has been established, the application replies to all data sent to it by saying "ok".

Figure 6 shows the implementation of this application. The application is initialized with the function called `example1_init()` and the uIP callback function is called `example1_app()`. For this application, the configuration variable `UIP_APPCALL` should be defined to be `example1_app`.

The initialization function calls the uIP function `uip_listen()` to register a listening port. The actual application function `example1_app()` uses the test functions `uip_newdata()` and `uip_rexmit()` to determine why it was called. If the application was called because the remote end has sent it data, it responds with an "ok". If the application function was called because data was lost in the network and has to be retransmitted, it also sends an "ok".

Note that this example shows a complete uIP application. It is not required for an application to deal with all types of events such as `uip_connected()` or `uip_timedout()`.

```
void example1_init(void) {
   uip_listen(1234);
}

void example1_app(void) {
   if(uip_newdata() || uip_rexmit()) {
      uip_send("ok\n", 3);
   }
}
```

Figure 6: A very simple application.

6.2 A more advanced application

This second example is slightly more advanced than the previous one, and shows how the application state field in the `uip_conn` structure is used.

This application is similar to the first application in that it listens to a port for incoming connections and responds to data sent to it with a single "ok". The big difference is that this application prints out a welcoming "Welcome!" message when the connection has been established.

This seemingly small change of operation makes a big difference in how the application is implemented. The reason for the increase in complexity is that if data should be lost in the network, the application must know what data to retransmit. If the "Welcome!" message was lost, the application must retransmit the welcome and if one of the "ok" messages is lost, the application must send a new "ok".

The application knows that as long as the "Welcome!" message has not been acknowledged by the remote host, it might have been dropped in the network. But once the remote host has sent an acknowledgment back, the application can be sure that the welcome has been received and knows that any lost data must be an "ok" message. Thus the application can be in either of two states: either in the WELCOME-SENT state where the "Welcome!" has been sent but not acknowledged, or in the WELCOME-ACKED state where the "Welcome!" has been acknowledged.

When a remote host connects to the application, the application sends the "Welcome!" message and sets it's state to WELCOME-SENT. When the welcome message is acknowledged, the application moves to the WELCOME-ACKED state. If the application receives any new data from the remote host, it responds by sending an "ok" back.

If the application is requested to retransmit the last message, it looks at in which state the application is. If the application is in the WELCOME-SENT state, it sends a "Welcome!" message since it knows that the previous welcome message hasn't been acknowledged. If the application is in the WELCOME-ACKED state, it knows that the last message was an "ok" message and sends such a message.

The implementation of this application can be seen in Figure 7. Figure 8 shows the configuration settings for this application.

```
struct example2_state {
   enum {WELCOME_SENT, WELCOME_ACKED} state;
};

void example2_init(void) {
   uip_listen(2345);
}

void example2_app(void) {
   struct example2_state *s;

   s = (struct example2_state *)uip_conn->appstate;

   if(uip_connected()) {
      s->state = WELCOME_SENT;
      uip_send("Welcome!\n", 9);
      return;
   }

   if(uip_acked() && s->state == WELCOME_SENT) {
      s->state = WELCOME_ACKED;
   }

   if(uip_newdata()) {
      uip_send("ok\n", 3);
   }

   if(uip_rexmit()) {
      switch(s->state) {
      case WELCOME_SENT:
         uip_send("Welcome!\n", 9);
         break;
      case WELCOME_ACKED:
         uip_send("ok\n", 3);
         break;
      }
   }
}
```

Figure 7: A more advanced application.

```
#define UIP_APPCALL       example2_app
#define UIP_APPSTATE_SIZE sizeof(struct example2_state)
```

Figure 8: Configuration settings for the application.

6.3 Differentiating between applications

If the system should run multiple applications, one technique to differentiate between them is to use the TCP port number of either the remote end or the local end of the connection. Figure 9 shows how the two examples above can be combined into one application.

```
void example3_init(void) {
   example1_init();
   example2_init();
}

void example3_app(void) {
   switch(uip_conn->lport) {
   case htons(1234):
      example1_app();
      break;
   case htons(2345):
      example2_app();
      break;
   }
}
```

Figure 9: Two applications combined and differentiated using local port numbers.

6.4 Receiving large amounts of data

This example shows a simple application that connects to a host, sends an HTTP request for a file and downloads it to a slow device such a disk drive. This shows how to use the flow control functions of uIP. The application is shown in Figure 10.

When the connection has been established, an HTTP request is sent to the server. Since this is the only data that is sent, the application knows that if it needs to retransmit any data, it is that request that should be retransmitted. It is therefore possible to combine these two events as is done in the example.

When the application receives new data from the remote host, it sends this data to the device by using the function `device_enqueue()`. It is important to note that this example assumes that this function copies the data into its own buffers. The data in the `uip_appdata` buffer will be overwritten by the next incoming packet.

If the device's queue is full, the application stops the data from the remote host by calling the uIP function `uip_stop()`. The application can then be sure that it will not receive any new data until `uip_restart()` is called. The application polling event is used to check if the device's queue is no longer full and if so, the data flow is restarted with `uip_restart()`.

```
void example4_init(void) {
   u16_t ipaddr[2];
   uip_ipaddr(ipaddr, 192,168,0,1);
   uip_conn(ipaddr, 80);
}

void example4_app(void) {
   if(uip_connected() || uip_rexmit()) {
      uip_send("GET /file HTTP/1.0\r\nServer:192.186.0.1\r\n\r\n",
               48);
      return;
   }

   if(uip_newdata()) {
      device_enqueue(uip_appdata, uip_datalen());
      if(device_queue_full()) {
         uip_stop();
      }
   }

   if(uip_poll() && uip_stopped()) {
      if(!device_queue_full()) {
         uip_restart();
      }
   }
}
```

Figure 10: An application which receives large amounts of data

6.5 A simple web server

This example shows a very simple file server application that listens to two ports and uses the port number to determine which file to send. If the files are properly formatted, this simple application can be used as a web server with static pages. Figure 11 shows the implementation.

The application state consists of a pointer to the data that should be sent and the size of the data that is left to send. When a remote host connects to the application, the local port number is used to determine which file to send. The first chunk of data is sent using `uip_send()`. Care is taken so that at most MSS bytes of data is sent.

The application is driven by incoming acknowledgments. When data has been acknowledged, new data can be sent. If there is no more data to send, the connection is closed using `uip_close()`.

References

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

[Plu82] D. C. Plummer. An ethernet address resolution protocol. RFC 826, Internet Engineering Task Force, November 1982.

[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.

```
struct example5_state {
   char *dataptr;
   unsigned int dataleft;
};

void example5_init(void) {
   uip_listen(80);
   uip_listen(81);
}

void example5_app(void) {
   struct example5_state *s;
   s = (struct example5_state)uip_conn->appstate;

   if(uip_connected()) {
      switch(uip_conn->lport) {
      case htons(80):
         s->dataptr = data_port_80;
         s->dataleft = datalen_port_80;
         break;
      case htons(81):
         s->dataptr = data_port_81;
         s->dataleft = datalen_port_81;
         break;
      }
      uip_send(s->dataptr,
               uip_mss() < s->dataleft? uip_mss(): s->dataleft);
      return;
   }

   if(uip_acked()) {
      if(s->dataleft < uip_mss()) {
         uip_close();
         return;
      }
      s->dataptr += uip_mss();
      s->dataleft -= uip_mss();
      uip_send(s->dataptr,
               uip_mss() < s->dataleft? uip_mss(): s->dataleft);
   }
}
```

Figure 11: A simple file server.