uIP 0.9 Reference Manual

Generated by Doxygen 1.3.3

Tue Oct 7 15:51:00 2003

Contents

Chapter 1

The uIP TCP/IP stack

Author:
Adam Dunkels <adam@dunkels.com>

The uIP TCP/IP stack is intended to make it possible to communicate using the TCP/IP protocol suite even on small 8-bit micro-controllers. Despite being small and simple, uIP do not require their peers to have complex, full-size stacks, but can communicate with peers running a similarly light-weight stack. The code size is on the order of a few kilobytes and RAM usage can be configured to be as low as a few hundred bytes.

1.1 uIP introduction

With the success of the Internet, the TCP/IP protocol suite has become a global standard for communication. TCP/IP is the underlying protocol used for web page transfers, e-mail transmissions, file transfers, and peer-to-peer networking over the Internet. For embedded systems, being able to run native TCP/IP makes it possible to connect the system directly to an intranet or even the global Internet. Embedded devices with full TCP/IP support will be first-class network citizens, thus being able to fully communicate with other hosts in the network.

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. 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 a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and contains only a rudimentary UDP implementation, but focuses on the IP, ICMP and TCP protocols. uIP is written in the C programming language.

Many other TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with another equally limited device, e.g., when running distributed peer-to-peer services and protocols. uIP is designed to be RFC compliant in order to let the embedded devices to act as first-class network citizens. The uIP TCP/IP implementation that is not tailored for any specific application.

1.2 TCP/IP communication

The full TCP/IP suite 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 SMTP that is used to transfer e-mail. The uIP is mostly concerned with the TCP and IP protocols and upper layer protocols will be referred to as ”the application”. Lower layer protocols are often implemented in hardware or firmware and will be referred to as ”the network device” that are controlled by the network device driver.

TCP provides a reliable byte stream to the upper layer protocols. It 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 destination is not on the physically connected network, the IP packet is forwarded onto another network by a router that is situated between the two networks. 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 segments are 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.

The formal requirements for the protocols in the TCP/IP stack is specified in a number of RFC documents published by the Internet Engineering Task Force, IETF. Each of the protocols in the stack is defined in one more RFC documents and RFC1122 collects all requirements and updates the previous RFCs.

The RFC1122 requirements can be divided into two categories; those that deal with the host to host communication and those that deal with communication between the application and the networking stack. An example of the first kind is ”A TCP MUST be able to receive a TCP option in any segment” and an example of the second kind is ”There MUST be a mechanism for reporting soft TCP error conditions to the application.” A TCP/IP implementation that violates requirements of the first kind may not be able to communicate with other TCP/IP implementations and may even lead to network failures. Violation of the second kind of requirements will only affect the communication within the system and will not affect host-to-host communication.

In uIP, all RFC requirements that affect host-to-host communication are implemented. However, in order to reduce code size, we have removed certain mechanisms in the interface between the application and the stack, such as the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections. Since there are only very few applications that make use of those features they can be removed without loss of generality.

1.3 Memory management

In the architectures for which uIP is intended, RAM is the most scarce resource. With only a few kilobytes of RAM available for the TCP/IP stack to use, mechanisms used in traditional TCP/IP cannot be directly applied.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

In uIP, the same global packet buffer that is used for incoming packets is also used for the TCP/IP headers of outgoing data. If the application sends dynamic data, it may use the parts of the global packet buffer that are not used for headers as a temporary storage buffer. To send the data, the application passes a pointer to the data as well as the length of the data to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and the application data out on the network. The data is not queued for retransmissions. Instead, the application will have to reproduce the data if a retransmission is necessary.

The total amount of memory usage for uIP depends heavily on the applications of the particular device in which the implementations are to be run. The memory configuration determines both the amount of traffic the system should be able to handle and the maximum amount of simultaneous connections. A device that will be sending large e-mails while at the same time running a web server with highly dynamic web pages and multiple simultaneous clients, will require more RAM than a simple Telnet server. It is possible to run the uIP implementation with as little as 200 bytes of RAM, but such a configuration will provide extremely low throughput and will only allow a small number of simultaneous connections.

1.4 Application program interface (API)

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. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in the intended uIP target architectures, the BSD socket interface is not suitable for our purposes.

Instead, uIP uses an event driven interface where the application is invoked in response to certain events. An application running on top of uIP 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. Because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

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. uIP does not keep track of packet contents after they have been sent by the device driver, and uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be retransmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

1.4.1 Application events

The application must be implemented as a C function, UIP_APPCALL(), that uIP calls whenever an event occurs. Each event has a corresponding test function that is 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).

1.4.2 The connection pointer

When the application is called by uIP, the global variable uip_conn is set to point to the uip_conn structure for the connection that currently is handled, and is called the ”current connection”. The fields in the uip_conn structure for the current connection can be used, e.g., to distinguish between different services, or to check to which IP address the connection is connected. One 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.

1.4.3 Receiving data

If the uIP test function uip_newdata() is non-zero, 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(). The data is not buffered by uIP, but will be overwritten after the application function returns, and the application will therefor have to either act directly on the incoming data, or by itself copy the incoming data into a buffer for later processing.

1.4.4 Sending data

When sending data, uIP adjusts the length of the data sent by the application according to the available buffer space and the current TCP window advertised by the receiver. The amount of buffer space is dictated by the memory configuration. It is therefore possible that all data sent from the application does not arrive at the receiver, and the application may use the uip_mss() function to see how much data that actually will be sent by the stack.

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 and it is not possible to call uip_send() more than once per application invocation; only the data from the last call will be sent.

1.4.5 Retransmitting data

Retransmissions are driven by the periodic TCP timer. Every time the periodic timer is invoked, the retransmission timer for each connection is decremented. If the timer reaches zero, a retransmission should be made. As uIP does not keep track of packet contents after they have been sent by the device driver, uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be retransmitted, the application function is called with the uip_rexmit() flag set, indicating that a retransmission is required.

The application must check the uip_rexmit() flag and produce the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefor, the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

1.4.6 Closing connections

The application closes the current connection by calling the uip_close() during an application call. 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.

1.4.7 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 can use the two test functions uip_aborted() and uip_timedout() to test for those error conditions.

1.4.8 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.

The polling event has two purposes. The first is to let the application periodically know that a connection is idle, which allows the application to close connections that have been idle for too long. The other purpose is to let the application send new data that has been produced. The application can only send data when invoked by uIP, and therefore the poll event is the only way to send data on an otherwise idle connection.

1.4.9 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.

The application can check the lport field in the uip_conn structure to check to which port the new connection was connected.

1.4.10 Opening connections

New connections can be opened from within uIP by the function uip_connect(). This function allocates a new connection and sets a flag in the connection state which will open a TCP connection to the specified IP address and port the next time the connection is polled by uIP. The uip_connect() function returns a pointer to the uip_conn structure for the new connection. If there are no free connection slots, the function returns NULL.

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 below. 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().

```
void connect_example1_app(void) {
   if(uip_connect(uip_conn->ripaddr, HTONS(8080)) == NULL) {
      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_example2(void) {
   u16_t ipaddr[2];

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

1.5 uIP device drivers

From the network device driver's standpoint, uIP consists of two C functions: uip_input() and uip_periodic(). The uip_input() function should be called by the device driver when an IP packet has been received and put into the uip_buf packet buffer. The uip_input() function will process the packet, and when it returns an outbound packet may have been placed in the same uip_buf packet buffer (indicated by the uip_len variable being non-zero). The device driver should then send out this packet onto the network.

The uip_periodic() function should be invoked periodically once per connection by the device driver, typically one per second. This function is used by uIP to drive protocol timers and retransmissions, and when it returns it may have placed an outbound packet in the uip_buf buffer.

1.6 Architecture specific functions

uIP requires a few functions to be implemented specifically for the architecture on which uIP is intended to run. These functions should be hand-tuned for the particular architecture, but generic C implementations are given as part of the uIP distribution.

1.6.1 Checksum calculation

The TCP and IP protocols implement a checksum that covers the data and header portions of the TCP and IP packets. Since the calculation of this checksum is made over all bytes in every packet being sent and received it is important that the function that calculates the checksum is efficient. Most often, this means that the checksum calculation must be fine-tuned for the particular architecture on which the uIP stack runs.

Because of this, uIP does not implement a generic checksum function, but leaves this to the architecture specific files which must implement the two functions uip_ipchksum() and uip_tcpchksum(). The checksum calculations in those functions can be written in highly optimized assembler rather than generic C code.

An example C implementation of the checksum function is provided in the uIP distribution.

1.6.2 32-bit arithmetic

The TCP protocol uses 32-bit sequence numbers, and a TCP implementation will have to do a number of 32-bit additions as part of the normal protocol processing. Since 32-bit arithmetic is not natively available on many of the platforms for which uIP is intended, uIP leaves the 32-bit additions to be implemented by the architecture specific module and does not make use of any 32-bit arithmetic in the main code base.

The architecture specific code must implement a function uip_add32() which does a 32-bit addition and stores the result in a global variable uip_acc32.

1.7 Examples

This section presents a number of very simple uIP applications. The uIP code distribution contains several more complex applications.

1.7.1 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”

The implementation of this application is shown below. 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().

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

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

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 actually 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().

1.7.2 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 is seen below. This configuration settings for the application is follows after its implementation.

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

void example2_init(void) {
   uip_listen(HTONS(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;
      }
   }
}
```

The configuration for the application:

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

1.7.3 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. The example below 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;
   }
}
```

1.7.4 Utilizing TCP flow control

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.

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

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().

1.7.5 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. The implementation follows.

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

void example5_init(void) {
   uip_listen(HTONS(80));
   uip_listen(HTONS(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, s->dataleft);
      return;
   }

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

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(). uIP makes sure that no more than MSS bytes of data is actually sent, even though s->dataleft may be larger than the MSS.

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().

1.7.6 Structured application program design

When writing larger programs using uIP it is useful to be able to utilize the uIP API in a structured way. The following example provides a structured design that has showed itself to be useful for writing larger protocol implementations than the previous examples showed here. The program is divided into an uIP event handler function that calls seven application handler functions that process new data, act on acknowledged data, send new data, deal with connection establishment or closure events and handle errors. The functions are called newdata(), acked(), senddata(), connected(), closed(), aborted(), and timedout(), and needs to be written specifically for the protocol that is being implemented.

The uIP event handler function is shown below.

```
void example6_app(void) {
  if(uip_aborted()) {
    aborted();
  }
  if(uip_timedout()) {
    timedout();
  }
  if(uip_closed()) {
    closed();
  }
  if(uip_connected()) {
    connected();
  }
  if(uip_acked()) {
    acked();
  }
  if(uip_newdata()) {
    newdata();
  }
  if(uip_rexmit() ||
     uip_newdata() ||
     uip_acked() ||
     uip_connected() ||
     uip_poll()) {
    senddata();
  }
}
```

The function starts with dealing with any error conditions that might have happened by checking if uip_aborted() or uip_timedout() are true. If so, the appropriate error function is called. Also, if the connection has been closed, the closed() function is called to the it deal with the event.

Next, the function checks if the connection has just been established by checking if uip_connected() is true. The connected() function is called and is supposed to do whatever needs to be done when the connection is established, such as intializing the application state for the connection. Since it may be the case that data should be sent out, the senddata() function is called to deal with the outgoing data.

The following very simple application serves as an example of how the application handler functions might look. This application simply waits for any data to arrive on the connection, and responds to the data by sending out the message ”Hello world!”. To illustrate how to develop an application state machine, this message is sent in two parts, first the ”Hello” part and then the ”world!” part.

```
#define STATE_WAITING 0
#define STATE_HELLO   1
#define STATE_WORLD   2

struct example6_state {
  u8_t state;
  char *textptr;
  int  textlen;
};

static void aborted(void) {}
static void timedout(void) {}
static void closed(void) {}

static void connected(void) {
  struct example6_state *s = (struct example6_state *)uip_conn->appstate;

  s->state   = STATE_WAITING;
  s->textlen = 0;
}

static void newdata(void) {
  struct example6_state *s = (struct example6_state *)uip_conn->appstate;

  if(s->state == STATE_WAITING) {
    s->state   = STATE_HELLO;
    s->textptr = "Hello ";
    s->textlen = 6;
  }
}

static void acked(void) {
  struct example6_state *s = (struct example6_state *)uip_conn->appstate;

  s->textlen -= uip_conn->len;
  s->textptr += uip_conn->len;
  if(s->textlen == 0) {
    switch(s->state) {
    case STATE_HELLO:
      s->state   = STATE_WORLD;
      s->textptr = "world!\n";
      s->textlen = 7;
      break;
    case STATE_WORLD:
      uip_close();
      break;
    }
  }
}

static void senddata(void) {
  struct example6_state *s = (struct example6_state *)uip_conn->appstate;

  if(s->textlen > 0) {
    uip_send(s->textptr, s->textlen);
  }
}
```

The application state consists of a ”state” variable, a ”textptr” pointer to a text message and the ”textlen” length of the text message. The ”state” variable can be either ”STATE_WAITING”, meaning that the application is waiting for data to arrive from the network, ”STATE_HELLO”, in which the application is sending the ”Hello” part of the message, or ”STATE_WORLD”, in which the application is sending the ”world!” message.

The application does not handle errors or connection closing events, and therefore the aborted(), timedout() and closed() functions are implemented as empty functions.

The connected() function will be called when a connection has been established, and in this case sets the ”state” variable to be ”STATE_WAITING” and the ”textlen” variable to be zero, indicating that there is no message to be sent out.

When new data arrives from the network, the newdata() function will be called by the event handler function. The newdata() function will check if the connection is in the ”STATE_WAITING” state, and if so switches to the ”STATE_HELLO” state and registers a 6 byte long ”Hello ” message with the connection. This message will later be sent out by the senddata() function.

The acked() function is called whenever data that previously was sent has been acknowleged by the receiving host. This acked() function first reduces the amount of data that is left to send, by subtracting the length of the previously sent data (obtained from ”uip_conn → len”) from the ”textlen” variable, and also adjusts the ”textptr” pointer accordingly. It then checks if the ”textlen” variable now is zero, which indicates that all data now has been successfully received, and if so changes application state. If the application was in the ”STATE_HELLO” state, it switches state to ”STATE_WORLD” and sets up a 7 byte ”world!\n” message to be sent. If the application was in the ”STATE_WORLD” state, it closes the connection.

Finally, the senddata() function takes care of actually sending the data that is to be sent. It is called by the event handler function when new data has been received, when data has been acknowledged, when a new connection has been established, when the connection is polled because of inactivity, or when a retransmission should be made. The purpose of the senddata() function is to optionally format the data that is to be sent, and to call the uip_send() function to actually send out the data. In this particular example, the function simply calls uip_send() with the appropriate arguments if data is to be sent, after checking if data should be sent out or not as indicated by the ”textlen” variable.

It is important to note that the senddata() function never should affect the application state; this should only be done in the acked() and newdata() functions.

Chapter 2

uIP 0.9 Module Index

2.1 uIP 0.9 Modules

Here is a list of all modules:

Chapter 3

uIP 0.9 Data Structure Index

3.1 uIP 0.9 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

uIP 0.9 File Index

4.1 uIP 0.9 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

uIP 0.9 Module Documentation

5.1 The uIP TCP/IP stack

Files

- file uip.c

  The uIP TCP/IP stack code.

- file uip.h

  Header file for the uIP TCP/IP stack.

Modules

- uIP configuration functions
- uIP initialization functions
- uIP device driver functions
- uIP application functions
- uIP conversion functions
- uIP Address Resolution Protocol
- Serial Line IP (SLIP) protocol
- uIP hostname resolver functions
- Architecture specific uIP functions

Data Structures

- struct uip_conn

  Representation of a uIP TCP connection.

- struct uip_stats

  The structure holding the TCP/IP statistics that are gathered if UIP_STATISTICS is set to 1.

- struct uip_udp_conn

  Representation of a uIP UDP connection.

Functions

- void uip_init (void)

  uIP initialization function.

- uip_udp_conn ∗ uip_udp_new (u16_t ∗ripaddr, u16_t rport)

  Set up a new UDP connection.

- void uip_unlisten (u16_t port)

  Stop listening to the specified port.

- void uip_listen (u16_t port)

  Start listening to the specified port.

- u16_t htons (u16_t val)

  Convert 16-bit quantity from host byte order to network byte order.

Variables

- volatile u8_t ∗ uip_appdata

  Pointer to the application data in the packet buffer.

- uip_stats uip_stat

  The uIP TCP/IP statistics.

- u8_t uip_buf [UIP_BUFSIZE+2]

  The uIP packet buffer.

- volatile u8_t uip_acc32 [4]

  4-byte array used for the 32-bit sequence number calculations.

5.1.1 Function Documentation

5.1.1.1 u16_t htons (u16_t *val*)

Convert 16-bit quantity from host byte order to network byte order.

This function is primarily used for converting variables from host byte order to network byte order. For converting constants to network byte order, use the HTONS() macro instead.

5.1.1.2 void uip_init (void)

uIP initialization function.

This function should be called at boot up to initilize the uIP TCP/IP stack.

5.1.1.3 void uip_listen (u16_t *port*)

Start listening to the specified port.

Note:

Since this function expects the port number in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_listen(HTONS(80));
```

Parameters:

port A 16-bit port number in network byte order.

5.1.1.4 struct uip_udp_conn* uip_udp_new (u16_t * *ripaddr*, u16_t *rport*)

Set up a new UDP connection.

Parameters:

ripaddr A pointer to a 4-byte structure representing the IP address of the remote host.

rport The remote port number in network byte order.

Returns:

The uip_udp_conn structure for the new connection or NULL if no connection could be allocated.

5.1.1.5 void uip_unlisten (u16_t *port*)

Stop listening to the specified port.

Note:

Since this function expects the port number in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_unlisten(HTONS(80));
```

Parameters:

port A 16-bit port number in network byte order.

5.1.2 Variable Documentation

5.1.2.1 volatile u8_t* uip_appdata

Pointer to the application data in the packet buffer.

This pointer points to the application data when the application is called. If the application wishes to send data, the application may use this space to write the data into before calling uip_send().

5.1.2.2 u8_t uip_buf[UIP_BUFSIZE+2]

The uIP packet buffer.

The uip_buf array is used to hold incoming and outgoing packets. The device driver should place incoming data into this buffer. When sending data, the device driver should read the link level headers and the TCP/IP headers from this buffer. The size of the link level headers is configured by the UIP_LLH_LEN define.

Note:

The application data need not be placed in this buffer, so the device driver must read it from the place pointed to by the uip_appdata pointer as illustrated by the following example:

```
void
devicedriver_send(void)
{
   hwsend(&uip_buf[0], UIP_LLH_LEN);
   hwsend(&uip_buf[UIP_LLH_LEN], 40);
   hwsend(uip_appdata, uip_len - 40 - UIP_LLH_LEN);
}
```

5.1.2.3 struct uip_stats uip_stat

The uIP TCP/IP statistics.

This is the variable in which the uIP TCP/IP statistics are gathered.

5.2 Example applications

5.2.1 Detailed Description

The uIP distribution contains a number of example applications that can be either used directory or studied when learning to develop applications for uIP.

Files

- file memb.c

  Memory block allocation routines.

- file memb.h

  Memory block allocation routines.

Modules

- Web client
- SMTP E-mail sender
- Telnet server
- Web server

Defines

- #define MEMB(name, size, num)

  Declare a memory block.

Functions

- void memb_init (struct memb_blocks ∗m)

  Initialize a memory block that was declared with MEMB().

- char ∗ memb_alloc (struct memb_blocks ∗m)

  Allocate a memory block from a block of memory declared with MEMB().

- char memb_ref (struct memb_blocks ∗m, char ∗ptr)

  Increase the reference count for a memory chunk.

- char memb_free (struct memb_blocks ∗m, char ∗ptr)

  Deallocate a memory block from a memory block previously declared with MEMB().

5.2.2 Define Documentation

5.2.2.1 #define MEMB(name, size, num)

Value:

```
static char memb_mem[(size + 1) * num]; \
        static struct memb_blocks name = {size, num, memb_mem}
```

Declare a memory block.

Parameters:

name The name of the memory block (later used with memb_init(), memb_alloc() and memb_free()).

size The size of each memory chunk, in bytes.

num The total number of memory chunks in the block.

5.2.3 Function Documentation

5.2.3.1 char∗ memb_alloc (struct memb_blocks ∗ *m*)

Allocate a memory block from a block of memory declared with MEMB().

Parameters:

m A memory block previosly declared with MEMB().

5.2.3.2 char memb_free (struct memb_blocks ∗ *m*, char ∗ *ptr*)

Deallocate a memory block from a memory block previously declared with MEMB().

Parameters:

m m A memory block previosly declared with MEMB().

ptr A pointer to the memory block that is to be deallocated.

Returns:

The new reference count for the memory block (should be 0 if successfully deallocated) or -1 if the pointer ”ptr” did not point to a legal memory block.

5.2.3.3 void memb_init (struct memb_blocks ∗ *m*)

Initialize a memory block that was declared with MEMB().

Parameters:

m A memory block previosly declared with MEMB().

5.2.3.4 char memb_ref (struct memb_blocks ∗ *m*, char ∗ *ptr*)

Increase the reference count for a memory chunk.

Note:

No sanity checks are currently made.

Parameters:

m m A memory block previosly declared with MEMB().

ptr A pointer to the memory chunk for which the reference count should be increased.

Returns:

The new reference count.

5.3 uIP configuration functions

5.3.1 Detailed Description

The uIP configuration functions are used for setting run-time parameters in uIP such as IP addresses.

Defines

- #define uip_sethostaddr(addr)

  Set the IP address of this host.

- #define uip_gethostaddr(addr)

  Get the IP address of this host.

- #define uip_setdraddr(addr)

  Set the default router's IP address.

- #define uip_setnetmask(addr)

  Set the netmask.

- #define uip_getdraddr(addr)

  Get the default router's IP address.

- #define uip_getnetmask(addr)

  Get the netmask.

- #define uip_setethaddr(eaddr)

  Specifiy the Ethernet MAC address.

5.3.2 Define Documentation

5.3.2.1 #define uip_getdraddr(addr)

Get the default router's IP address.

Parameters:

addr A pointer to a 4-byte array that will be filled in with the IP address of the default router.

5.3.2.2 #define uip_gethostaddr(addr)

Get the IP address of this host.

The IP address is represented as a 4-byte array where the first octet of the IP address is put in the first member of the 4-byte array.

Parameters:

addr A pointer to a 4-byte array that will be filled in with the currently configured IP address.

5.3.2.3 #define uip_getnetmask(addr)

Get the netmask.

Parameters:

addr A pointer to a 4-byte array that will be filled in with the value of the netmask.

5.3.2.4 #define uip_setdraddr(addr)

Set the default router's IP address.

Parameters:

addr A pointer to a 4-byte array containing the IP address of the default router.

5.3.2.5 #define uip_setethaddr(eaddr)

Specifiy the Ethernet MAC address.

The ARP code needs to know the MAC address of the Ethernet card in order to be able to respond to ARP queries and to generate working Ethernet headers.

Note:

This macro only specifies the Ethernet MAC address to the ARP code. It cannot be used to change the MAC address of the Ethernet card.

Parameters:

eaddr A pointer to a struct uip_eth_addr containing the Ethernet MAC address of the Ethernet card.

5.3.2.6 #define uip_sethostaddr(addr)

Set the IP address of this host.

The IP address is represented as a 4-byte array where the first octet of the IP address is put in the first member of the 4-byte array.

Parameters:

addr A pointer to a 4-byte representation of the IP address.

5.3.2.7 #define uip_setnetmask(addr)

Set the netmask.

Parameters:

addr A pointer to a 4-byte array containing the IP address of the netmask.

5.4 uIP initialization functions

5.4.1 Detailed Description

The uIP initialization functions are used for booting uIP.

Functions

- void uip_init (void)
  uIP initialization function.

5.4.2 Function Documentation

5.4.2.1 void uip_init (void)

uIP initialization function.

This function should be called at boot up to initilize the uIP TCP/IP stack.

5.5 uIP device driver functions

5.5.1 Detailed Description

These functions are used by a network device driver for interacting with uIP.

Defines

- #define uip_input()

  Process an incoming packet.

- #define uip_periodic(conn)

  Periodic processing for a connection identified by its number.

- #define uip_periodic_conn(conn)

  Periodic processing for a connection identified by a pointer to its structure.

- #define uip_udp_periodic(conn)

  Periodic processing for a UDP connection identified by its number.

- #define uip_udp_periodic_conn(conn)

  Periodic processing for a UDP connection identified by a pointer to its structure.

Variables

- u8_t uip_buf [UIP_BUFSIZE+2]

  The uIP packet buffer.

5.5.2 Define Documentation

5.5.2.1 #define uip_input()

Process an incoming packet.

This function should be called when the device driver has received a packet from the network. The packet from the device driver must be present in the uip_buf buffer, and the length of the packet should be placed in the uip_len variable.

When the function returns, there may be an outbound packet placed in the uip_buf packet buffer. If so, the uip_len variable is set to the length of the packet. If no packet is to be sent out, the uip_len variable is set to 0.

The usual way of calling the function is presented by the source code below.

```
uip_len = devicedriver_poll();
if(uip_len > 0) {
  uip_input();
  if(uip_len > 0) {
    devicedriver_send();
  }
}
```

Note:

If you are writing a uIP device driver that needs ARP (Address Resolution Protocol), e.g., when running uIP over Ethernet, you will need to call the uIP ARP code before calling this function:

```
#define BUF ((struct uip_eth_hdr *)&uip_buf[0])
uip_len = ethernet_devicedrver_poll();
if(uip_len > 0) {
  if(BUF->type == HTONS(UIP_ETHTYPE_IP)) {
    uip_arp_ipin();
    uip_input();
    if(uip_len > 0) {
      uip_arp_out();
      ethernet_devicedriver_send();
    }
  } else if(BUF->type == HTONS(UIP_ETHTYPE_ARP)) {
    uip_arp_arpin();
    if(uip_len > 0) {
      ethernet_devicedriver_send();
    }
  }
```

5.5.2.2 #define uip_periodic(conn)

Periodic processing for a connection identified by its number.

This function does the necessary periodic processing (timers, polling) for a uIP TCP conneciton, and should be called when the periodic uIP timer goes off. It should be called for every connection, regardless of whether they are open of closed.

When the function returns, it may have an outbound packet waiting for service in the uIP packet buffer, and if so the uip_len variable is set to a value larger than zero. The device driver should be called to send out the packet.

The ususal way of calling the function is through a for() loop like this:

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

Note:

If you are writing a uIP device driver that needs ARP (Address Resolution Protocol), e.g., when running uIP over Ethernet, you will need to call the uip_arp_out() function before calling the device driver:

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

Parameters:

conn The number of the connection which is to be periodically polled.

5.5.2.3 #define uip_periodic_conn(conn)

Periodic processing for a connection identified by a pointer to its structure.

Same as uip_periodic() but takes a pointer to the actual uip_conn struct instead of an integer as its argument. This function can be used to force periodic processing of a specific connection.

Parameters:

conn A pointer to the uip_conn struct for the connection to be processed.

5.5.2.4 #define uip_udp_periodic(conn)

Periodic processing for a UDP connection identified by its number.

This function is essentially the same as uip_prerioic(), but for UDP connections. It is called in a similar fashion as the uip_periodic() function:

```
for(i = 0; i < UIP_UDP_CONNS; i++) {
  uip_udp_periodic(i);
  if(uip_len > 0) {
    devicedriver_send();
  }
}
```

Note:

As for the uip_periodic() function, special care has to be taken when using uIP together with ARP and Ethernet:

```
for(i = 0; i < UIP_UDP_CONNS; i++) {
  uip_udp_periodic(i);
  if(uip_len > 0) {
    uip_arp_out();
    ethernet_devicedriver_send();
  }
}
```

Parameters:

conn The number of the UDP connection to be processed.

5.5.2.5 #define uip_udp_periodic_conn(conn)

Periodic processing for a UDP connection identified by a pointer to its structure.

Same as uip_udp_periodic() but takes a pointer to the actual uip_conn struct instead of an integer as its argument. This function can be used to force periodic processing of a specific connection.

Parameters:

conn A pointer to the uip_udp_conn struct for the connection to be processed.

5.5.3 Variable Documentation

5.5.3.1 u8_t uip_buf[UIP_BUFSIZE+2]

The uIP packet buffer.

The uip_buf array is used to hold incoming and outgoing packets. The device driver should place incoming data into this buffer. When sending data, the device driver should read the link level headers and the TCP/IP headers from this buffer. The size of the link level headers is configured by the UIP_LLH_LEN define.

Note:

The application data need not be placed in this buffer, so the device driver must read it from the place pointed to by the uip_appdata pointer as illustrated by the following example:

```
void
devicedriver_send(void)
{
   hwsend(&uip_buf[0], UIP_LLH_LEN);
   hwsend(&uip_buf[UIP_LLH_LEN], 40);
   hwsend(uip_appdata, uip_len - 40 - UIP_LLH_LEN);
}
```

5.6 uIP application functions

5.6.1 Detailed Description

Functions used by an application running of top of uIP.

Defines

- #define uip_send(data, len)

  Send data on the current connection.

- #define uip_datalen()

  The length of any incoming data that is currently avaliable (if avaliable) in the uip_appdata buffer.

- #define uip_urgdatalen()

  The length of any out-of-band data (urgent data) that has arrived on the connection.

- #define uip_close()

  Close the current connection.

- #define uip_abort()

  Abort the current connection.

- #define uip_stop()

  Tell the sending host to stop sending data.

- #define uip_stopped(conn)

  Find out if the current connection has been previously stopped with uip_stop().

- #define uip_restart()

  Restart the current connection, if is has previously been stopped with uip_stop().

- #define uip_newdata()

  Is new incoming data available?

- #define uip_acked()

  Has previously sent data been acknowledged?

- #define uip_connected()

  Has the connection just been connected?

- #define uip_closed()

  Has the connection been closed by the other end?

- #define uip_aborted()

  Has the connection been aborted by the other end?

- #define uip_timedout()

  Has the connection timed out?

- #define uip_rexmit()

  Do we need to retransmit previously data?

- #define uip_poll()

  Is the connection being polled by uIP?

- #define uip_initialmss()

  Get the initial maxium segment size (MSS) of the current connection.

- #define uip_mss()

  Get the current maxium segment size that can be sent on the current connection.

- #define uip_udp_remove(conn)

  Removed a UDP connection.

- #define uip_udp_send(len)

  Send a UDP datagram of length len on the current connection.

Functions

- void uip_listen (u16_t port)

  Start listening to the specified port.

- void uip_unlisten (u16_t port)

  Stop listening to the specified port.

- uip_conn ∗ uip_connect (u16_t ∗ripaddr, u16_t port)

  Connect to a remote host using TCP.

- uip_udp_conn ∗ uip_udp_new (u16_t ∗ripaddr, u16_t rport)

  Set up a new UDP connection.

5.6.2 Define Documentation

5.6.2.1 #define uip_abort()

Abort the current connection.

This function will abort (reset) the current connection, and is usually used when an error has occured that prevents using the uip_close() function.

5.6.2.2 #define uip_aborted()

Has the connection been aborted by the other end?

Non-zero if the current connection has been aborted (reset) by the remote host.

5.6.2.3 #define uip_acked()

Has previously sent data been acknowledged?

Will reduce to non-zero if the previously sent data has been acknowledged by the remote host. This means that the application can send new data.

5.6.2.4 #define uip_close()

Close the current connection.

This function will close the current connection in a nice way.

5.6.2.5 #define uip_closed()

Has the connection been closed by the other end?

Is non-zero if the connection has been closed by the remote host. The application may then do the necessary clean-ups.

5.6.2.6 #define uip_connected()

Has the connection just been connected?

Reduces to non-zero if the current connection has been connected to a remote host. This will happen both if the connection has been actively opened (with uip_connect()) or passively opened (with uip_listen()).

5.6.2.7 #define uip_datalen()

The length of any incoming data that is currently avaliable (if avaliable) in the uip_appdata buffer.

The test function uip_data() must first be used to check if there is any data available at all.

5.6.2.8 #define uip_mss()

Get the current maxium segment size that can be sent on the current connection.

The current maxiumum segment size that can be sent on the connection is computed from the receiver's window and the MSS of the connection (which also is available by calling uip_initialmss()).

5.6.2.9 #define uip_newdata()

Is new incoming data available?

Will reduce to non-zero if there is new data for the application present at the uip_appdata pointer. The size of the data is avaliable through the uip_len variable.

5.6.2.10 #define uip_poll()

Is the connection being polled by uIP?

Is non-zero if the reason the application is invoked is that the current connection has been idle for a while and should be polled.

The polling event can be used for sending data without having to wait for the remote host to send data.

5.6.2.11 #define uip_restart()

Restart the current connection, if is has previously been stopped with uip_stop().

This function will open the receiver's window again so that we start receiving data for the current connection.

5.6.2.12 #define uip_rexmit()

Do we need to retransmit previously data?

Reduces to non-zero if the previously sent data has been lost in the network, and the application should retransmit it. The application should send the exact same data as it did the last time, using the uip_send() function.

5.6.2.13 #define uip_send(data, len)

Send data on the current connection.

This function is used to send out a single segment of TCP data. Only applications that have been invoked by uIP for event processing can send data.

The amount of data that actually is sent out after a call to this funcion is determined by the maximum amount of data TCP allows. uIP will automatically crop the data so that only the appropriate amount of data is sent. The function uip_mss() can be used to query uIP for the amount of data that actually will be sent.

Note:

This function does not guarantee that the sent data will arrive at the destination. If the data is lost in the network, the application will be invoked with the uip_rexmit() event being set. The application will then have to resend the data using this function.

Parameters:

data A pointer to the data which is to be sent.

len The maximum amount of data bytes to be sent.

5.6.2.14 #define uip_stop()

Tell the sending host to stop sending data.

This function will close our receiver's window so that we stop receiving data for the current connection.

5.6.2.15 #define uip_timedout()

Has the connection timed out?

Non-zero if the current connection has been aborted due to too many retransmissions.

5.6.2.16 #define uip_udp_remove(conn)

Removed a UDP connection.

Parameters:

conn A pointer to the uip_udp_conn structure for the connection.

5.6.2.17 #define uip_udp_send(len)

Send a UDP datagram of length len on the current connection.

This function can only be called in response to a UDP event (poll or newdata). The data must be present in the uip_buf buffer, at the place pointed to by the uip_appdata pointer.

Parameters:

len The length of the data in the uip_buf buffer.

5.6.2.18 #define uip_urgdatalen()

The length of any out-of-band data (urgent data) that has arrived on the connection.

Note:

The configuration parameter UIP_URGDATA must be set for this function to be enabled.

5.6.3 Function Documentation

5.6.3.1 struct uip_conn* uip_connect (u16_t * *ripaddr*, u16_t *port*)

Connect to a remote host using TCP.

This function is used to start a new connection to the specified port on the specied host. It allocates a new connection identifier, sets the connection to the SYN_SENT state and sets the retransmission timer to 0. This will cause a TCP SYN segment to be sent out the next time this connection is periodically processed, which usually is done within 0.5 seconds after the call to uip_connect().

Note:

This function is avaliable only if support for active open has been configured by defining UIP_ACTIVE_OPEN to 1 in uipopt.h.

Since this function requires the port number to be in network byte order, a convertion using HTONS() or htons() is necessary.

```
u16_t ipaddr[2];

uip_ipaddr(ipaddr, 192,168,1,2);
uip_connect(ipaddr, HTONS(80));
```

Parameters:

ripaddr A pointer to a 4-byte array representing the IP address of the remote hot.

port A 16-bit port number in network byte order.

Returns:

A pointer to the uIP connection identifier for the new connection, or NULL if no connection could be allocated.

5.6.3.2 void uip_listen (u16_t *port*)

Start listening to the specified port.

Note:

Since this function expects the port number in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_listen(HTONS(80));
```

Parameters:

port A 16-bit port number in network byte order.

5.6.3.3 struct uip_udp_conn∗ uip_udp_new (u16_t ∗ *ripaddr*, u16_t *rport*)

Set up a new UDP connection.

Parameters:

ripaddr A pointer to a 4-byte structure representing the IP address of the remote host.

rport The remote port number in network byte order.

Returns:

The uip_udp_conn structure for the new connection or NULL if no connection could be allocated.

5.6.3.4 void uip_unlisten (u16_t *port*)

Stop listening to the specified port.

Note:

Since this function expects the port number in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_unlisten(HTONS(80));
```

Parameters:

port A 16-bit port number in network byte order.

5.7 uIP conversion functions

5.7.1 Detailed Description

These functions can be used for converting between different data formats used by uIP.

Defines

- #define uip_ipaddr(addr, addr0, addr1, addr2, addr3)

  Pack an IP address into a 4-byte array which is used by uIP to represent IP addresses.

- #define HTONS(n)

  Convert 16-bit quantity from host byte order to network byte order.

Functions

- u16_t htons (u16_t val)

  Convert 16-bit quantity from host byte order to network byte order.

5.7.2 Define Documentation

5.7.2.1 #define HTONS(n)

Convert 16-bit quantity from host byte order to network byte order.

This macro is primarily used for converting constants from host byte order to network byte order. For converting variables to network byte order, use the htons() function instead.

5.7.2.2 #define uip_ipaddr(addr, addr0, addr1, addr2, addr3)

Pack an IP address into a 4-byte array which is used by uIP to represent IP addresses.

Example:

```
u16_t ipaddr[2];

uip_ipaddr(&ipaddr, 192,168,1,2);
```

Parameters:

addr A pointer to a 4-byte array that will be filled in with the IP addres.

addr0 The first octet of the IP address.

addr1 The second octet of the IP address.

addr2 The third octet of the IP address.

addr3 The forth octet of the IP address.

5.7.3 Function Documentation

5.7.3.1 u16_t htons (u16_t *val*)

Convert 16-bit quantity from host byte order to network byte order.

This function is primarily used for converting variables from host byte order to network byte order. For converting constants to network byte order, use the HTONS() macro instead.

5.8 Architecture specific uIP functions

5.8.1 Detailed Description

The functions in the architecture specific module implement the IP check sum and 32-bit additions.

The IP checksum calculation is the most computationally expensive operation in the TCP/IP stack and it therefore pays off to implement this in efficient assembler. The purpose of the uip-arch module is to let the checksum functions to be implemented in architecture specific assembler.

Files

- file uip_arch.h

  Declarations of architecture specific functions.

Functions

- void uip_add32 (u8_t ∗op32, u16_t op16)

  Carry out a 32-bit addition.

- u16_t uip_chksum (u16_t ∗buf, u16_t len)

  Calculate the Internet checksum over a buffer.

- u16_t uip_ipchksum (void)

  Calculate the IP header checksum of the packet header in uip_buf.

- u16_t uip_tcpchksum (void)

  Calculate the TCP checksum of the packet in uip_buf and uip_appdata.

Variables

- volatile u8_t uip_acc32 [4]

  4-byte array used for the 32-bit sequence number calculations.

5.8.2 Function Documentation

5.8.2.1 void uip_add32 (u8_t ∗ *op32*, u16_t *op16*)

Carry out a 32-bit addition.

Because not all architectures for which uIP is intended has native 32-bit arithmetic, uIP uses an external C function for doing the required 32-bit additions in the TCP protocol processing. This function should add the two arguments and place the result in the global variable uip_acc32.

Note:

The 32-bit integer pointed to by the op32 parameter and the result in the uip_acc32 variable are in network byte order (big endian).

Parameters:

op32 A pointer to a 4-byte array representing a 32-bit integer in network byte order (big endian).

op16 A 16-bit integer in host byte order.

5.8.2.2 u16_t uip_chksum (u16_t ∗ *buf*, u16_t *len*)

Calculate the Internet checksum over a buffer.

The Internet checksum is the one's complement of the one's complement sum of all 16-bit words in the buffer.

See RFC1071.

Note:

This function is not called in the current version of uIP, but future versions might make use of it.

Parameters:

buf A pointer to the buffer over which the checksum is to be computed.

len The length of the buffer over which the checksum is to be computed.

Returns:

The Internet checksum of the buffer.

5.8.2.3 u16_t uip_ipchksum (void)

Calculate the IP header checksum of the packet header in uip_buf.

The IP header checksum is the Internet checksum of the 20 bytes of the IP header.

Returns:

The IP header checksum of the IP header in the uip_buf buffer.

5.8.2.4 u16_t uip_tcpchksum (void)

Calculate the TCP checksum of the packet in uip_buf and uip_appdata.

The TCP checksum is the Internet checksum of data contents of the TCP segment, and a pseudo-header as defined in RFC793.

Note:

The uip_appdata pointer that points to the packet data may point anywhere in memory, so it is not possible to simply calculate the Internet checksum of the contents of the uip_buf buffer.

Returns:

The TCP checksum of the TCP segment in uip_buf and pointed to by uip_appdata.

5.9 uIP Address Resolution Protocol

5.9.1 Detailed Description

The Address Resolution Protocol ARP is used for mapping between IP addresses and link level addresses such as the Ethernet MAC addresses. ARP uses broadcast queries to ask for the link level address of a known IP address and the host which is configured with the IP address for which the query was meant, will respond with its link level address.

Note:

This ARP implementation only supports Ethernet.

Files

- file uip_arp.c

  Implementation of the ARP Address Resolution Protocol.

- file uip_arp.h

  Macros and definitions for the ARP module.

Data Structures

- struct uip_eth_addr

  Representation of a 48-bit Ethernet address.

- struct uip_eth_hdr

  The Ethernet header.

Functions

- void uip_arp_init (void)

  Initialize the ARP module.

- void uip_arp_ipin (void)

  ARP processing for incoming IP packets.

- void uip_arp_arpin (void)

  ARP processing for incoming ARP packets.

- void uip_arp_out (void)

  Prepend Ethernet header to an outbound IP packet and see if we need to send out an ARP request.

- void uip_arp_timer (void)

  Periodic ARP processing function.

5.9.2 Function Documentation

5.9.2.1 void uip_arp_arpin (void)

ARP processing for incoming ARP packets.

This function should be called by the device driver when an ARP packet has been received. The function will act differently depending on the ARP packet type: if it is a reply for a request that we previously sent out, the ARP cache will be filled in with the values from the ARP reply. If the incoming ARP packet is an ARP request for our IP address, an ARP reply packet is created and put into the uip_buf[] buffer.

When the function returns, the value of the global variable uip_len indicates whether the device driver should send out a packet or not. If uip_len is zero, no packet should be sent. If uip_len is non-zero, it contains the length of the outbound packet that is present in the uip_buf[] buffer.

This function expects an ARP packet with a prepended Ethernet header in the uip_buf[] buffer, and the length of the packet in the global variable uip_len.

5.9.2.2 void uip_arp_ipin (void)

ARP processing for incoming IP packets.

This function should be called by the device driver when an IP packet has been received. The function will check if the address is in the ARP cache, and if so the ARP cache entry will be refreshed. If no ARP cache entry was found, a new one is created.

This function expects an IP packet with a prepended Ethernet header in the uip_buf[] buffer, and the length of the packet in the global variable uip_len.

5.9.2.3 void uip_arp_out (void)

Prepend Ethernet header to an outbound IP packet and see if we need to send out an ARP request.

This function should be called before sending out an IP packet. The function checks the destination IP address of the IP packet to see what Ethernet MAC address that should be used as a destination MAC address on the Ethernet.

If the destination IP address is in the local network (determined by logical ANDing of netmask and our IP address), the function checks the ARP cache to see if an entry for the destination IP address is found. If so, an Ethernet header is prepended and the function returns. If no ARP cache entry is found for the destination IP address, the packet in the uip_buf[] is replaced by an ARP request packet for the IP address. The IP packet is dropped and it is assumed that they higher level protocols (e.g., TCP) eventually will retransmit the dropped packet.

If the destination IP address is not on the local network, the IP address of the default router is used instead.

When the function returns, a packet is present in the uip_buf[] buffer, and the length of the packet is in the global variable uip_len.

5.9.2.4 void uip_arp_timer (void)

Periodic ARP processing function.

This function performs periodic timer processing in the ARP module and should be called at regular intervals. The recommended interval is 10 seconds between the calls.

5.10 Serial Line IP (SLIP) protocol

5.10.1 Detailed Description

The SLIP protocol is a very simple way to transmit IP packets over a serial line. It does not provide any framing or error control, and is therefore not very widely used today.

This SLIP implementation requires two functions for accessing the serial device: slipdev_char_poll() and slipdev_char_put(). These must be implemented specifically for the system on which the SLIP protocol is to be run.

Files

- file slipdev.c

  SLIP protocol implementation.

- file slipdev.h

  SLIP header file.

Functions

- void slipdev_char_put (u8_t c)

  Put a character on the serial device.

- u8_t slipdev_char_poll (u8_t ∗c)

  Poll the serial device for a character.

- void slipdev_init (void)

  Initialize the SLIP module.

- void slipdev_send (void)

  Send the packet in the uip_buf and uip_appdata buffers using the SLIP protocol.

- u16_t slipdev_poll (void)

  Poll the SLIP device for an available packet.

5.10.2 Function Documentation

5.10.2.1 u8_t slipdev_char_poll (u8_t ∗ *c*)

Poll the serial device for a character.

This function is used by the SLIP implementation to poll the serial device for a character. It must be implemented specifically for the system on which the SLIP implementation is to be run.

The function should return immediately regardless if a character is available or not. If a character is available it should be placed at the memory location pointed to by the pointer supplied by the arguement c.

Parameters:

c A pointer to a byte that is filled in by the function with the received character, if available.

Return values:

0 If no character is available.

Non-zero If a character is available.

5.10.2.2 void slipdev_char_put (u8_t *c*)

Put a character on the serial device.

This function is used by the SLIP implementation to put a character on the serial device. It must be implemented specifically for the system on which the SLIP implementation is to be run.

Parameters:

c The character to be put on the serial device.

5.10.2.3 void slipdev_init (void)

Initialize the SLIP module.

This function does not initialize the underlying RS232 device, but only the SLIP part.

5.10.2.4 u16_t slipdev_poll (void)

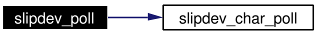
Poll the SLIP device for an available packet.

This function will poll the SLIP device to see if a packet is available. It uses a buffer in which all avaliable bytes from the RS232 interface are read into. When a full packet has been read into the buffer, the packet is copied into the uip_buf buffer and the length of the packet is returned.

Returns:

The length of the packet placed in the uip_buf buffer, or zero if no packet is available.

Here is the call graph for this function:

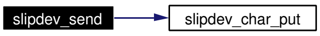


5.10.2.5 void slipdev_send (void)

Send the packet in the uip_buf and uip_appdata buffers using the SLIP protocol.

The first 40 bytes of the packet (the IP and TCP headers) are read from the uip_buf buffer, and the following bytes (the application data) are read from the uip_appdata buffer.

Here is the call graph for this function:



5.11 Configuration options for uIP

5.11.1 Detailed Description

uIP is configured using the per-project configuration file ”uipopt.h”. This file contains all compile-time options for uIP and should be tweaked to match each specific project. The uIP distribution contains a documented example ”uipopt.h” that can be copied and modified for each project.

Files

- file uipopt.h

  Configuration options for uIP.

Modules

- uIP type definitions
- Static configuration options
- IP configuration options
- UDP configuration options
- TCP configuration options
- ARP configuration options
- General configuration options
- CPU architecture configuration
- Appication specific configurations

5.12 uIP type definitions

Typedefs

- typedef unsigned char u8_t
  The 8-bit unsigned data type.
- typedef unsigned short u16_t
  The 16-bit unsigned data type.
- typedef unsigned short uip_stats_t
  The statistics data type.

5.12.1 Typedef Documentation

5.12.1.1 typedef unsigned short u16_t

The 16-bit unsigned data type.

This may have to be tweaked for your particular compiler. ”unsigned short” works for most compilers.

5.12.1.2 typedef unsigned char u8_t

The 8-bit unsigned data type.

This may have to be tweaked for your particular compiler. ”unsigned char” works for most compilers.

5.12.1.3 typedef unsigned short uip_stats_t

The statistics data type.

This datatype determines how high the statistics counters are able to count.

5.13 Static configuration options

5.13.1 Detailed Description

These configuration options can be used for setting the IP address settings statically, but only if UIP_FIXEDADDR is set to 1. The configuration options for a specific node includes IP address, netmask and default router as well as the Ethernet address. The netmask, default router and Ethernet address are appliciable only if uIP should be run over Ethernet.

All of these should be changed to suit your project.

Defines

- #define UIP_FIXEDADDR

  Determines if uIP should use a fixed IP address or not.

- #define UIP_PINGADDRCONF

  Ping IP address asignment.

- #define UIP_IPADDR0

  The first octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_IPADDR1

  The second octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_IPADDR2

  The third octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_IPADDR3

  The fourth octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK0

  The first octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK1

  The second octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK2

  The third octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK3

  The fourth octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR0

  The first octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR1

  The second octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR2

The third octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR3

  The fourth octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_FIXEDETHADDR

  Specifies if the uIP ARP module should be compiled with a fixed Ethernet MAC address or not.

- #define UIP_ETHADDR0

  The first octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR1

  The second octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR2

  The third octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR3

  The fourth octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR4

  The fifth octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR5

  The sixth octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

5.13.2 Define Documentation

5.13.2.1 #define UIP_FIXEDADDR

Determines if uIP should use a fixed IP address or not.

If uIP should use a fixed IP address, the settings are set in the uipopt.h file. If not, the macros uip_sethostaddr(), uip_setdraddr() and uip_setnetmask() should be used instead.

5.13.2.2 #define UIP_FIXEDETHADDR

Specifies if the uIP ARP module should be compiled with a fixed Ethernet MAC address or not.

If this configuration option is 0, the macro uip_setethaddr() can be used to specify the Ethernet address at run-time.

5.13.2.3 #define UIP_PINGADDRCONF

Ping IP address asignment.

uIP uses a ”ping” packets for setting its own IP address if this option is set. If so, uIP will start with an empty IP address and the destination IP address of the first incoming ”ping” (ICMP echo) packet will be used for setting the hosts IP address.

Note:

This works only if UIP_FIXEDADDR is 0.

5.14 IP configuration options

Defines

- #define UIP_TTL 255

  The IP TTL (time to live) of IP packets sent by uIP.
- #define UIP_REASSEMBLY

  Turn on support for IP packet reassembly.
- #define UIP_REASS_MAXAGE 40

  The maximum time an IP fragment should wait in the reassembly buffer before it is dropped.

5.14.1 Define Documentation

5.14.1.1 #define UIP_REASSEMBLY

Turn on support for IP packet reassembly.

uIP supports reassembly of fragmented IP packets. This features requires an additonal amount of RAM to hold the reassembly buffer and the reassembly code size is approximately 700 bytes. The reassembly buffer is of the same size as the uip_buf buffer (configured by UIP_BUFSIZE).

Note:

IP packet reassembly is not heavily tested.

5.14.1.2 #define UIP_TTL 255

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

This should normally not be changed.

5.15 UDP configuration options

5.15.1 Detailed Description

Note:

The UDP support in uIP is still not entirely complete; there is no support for sending or receiving broadcast or multicast packets, but it works well enough to support a number of vital applications such as DNS queries, though

Defines

- #define UIP_UDP

  Toggles wether UDP support should be compiled in or not.

- #define UIP_UDP_CHECKSUMS

  Toggles if UDP checksums should be used or not.

- #define UIP_UDP_CONNS

  The maximum amount of concurrent UDP connections.

- #define UIP_UDP_APPCALL

  The name of the function that should be called when UDP datagrams arrive.

5.15.2 Define Documentation

5.15.2.1 #define UIP_UDP_CHECKSUMS

Toggles if UDP checksums should be used or not.

Note:

Support for UDP checksums is currently not included in uIP, so this option has no function.

5.16 TCP configuration options

Defines

- #define UIP_ACTIVE_OPEN

  Determines if support for opening connections from uIP should be compiled in.

- #define UIP_CONNS

  The maximum number of simultaneously open TCP connections.

- #define UIP_LISTENPORTS

  The maximum number of simultaneously listening TCP ports.

- #define UIP_RECEIVE_WINDOW

  The size of the advertised receiver's window.

- #define UIP_URGDATA

  Determines if support for TCP urgent data notification should be compiled in.

- #define UIP_RTO 3

  The initial retransmission timeout counted in timer pulses.

- #define UIP_MAXRTX 8

  The maximum number of times a segment should be retransmitted before the connection should be aborted.

- #define UIP_MAXSYNRTX 3

  The maximum number of times a SYN segment should be retransmitted before a connection request should be deemed to have been unsuccessful.

- #define UIP_TCP_MSS (UIP_BUFSIZE - UIP_LLH_LEN - 40)

  The TCP maximum segment size.

- #define UIP_TIME_WAIT_TIMEOUT 120

  How long a connection should stay in the TIME_WAIT state.

5.16.1 Define Documentation

5.16.1.1 #define UIP_ACTIVE_OPEN

Determines if support for opening connections from uIP should be compiled in.

If the applications that are running on top of uIP for this project do not need to open outgoing TCP connections, this configration option can be turned off to reduce the code size of uIP.

5.16.1.2 #define UIP_CONNS

The maximum number of simultaneously open TCP connections.

Since the TCP connections are statically allocated, turning this configuration knob down results in less RAM used. Each TCP connection requires approximatly 30 bytes of memory.

5.16.1.3 #define UIP_LISTENPORTS

The maximum number of simultaneously listening TCP ports.

Each listening TCP port requires 2 bytes of memory.

5.16.1.4 #define UIP_MAXRTX 8

The maximum number of times a segment should be retransmitted before the connection should be aborted.

This should not be changed.

5.16.1.5 #define UIP_MAXSYNRTX 3

The maximum number of times a SYN segment should be retransmitted before a connection request should be deemed to have been unsuccessful.

This should not need to be changed.

5.16.1.6 #define UIP_RECEIVE_WINDOW

The size of the advertised receiver's window.

Should be set low (i.e., to the size of the uip_buf buffer) is the application is slow to process incoming data, or high (32768 bytes) if the application processes data quickly.

5.16.1.7 #define UIP_RTO 3

The initial retransmission timeout counted in timer pulses.

This should not be changed.

5.16.1.8 #define UIP_TCP_MSS (UIP_BUFSIZE - UIP_LLH_LEN - 40)

The TCP maximum segment size.

This is should not be to set to more than UIP_BUFSIZE - UIP_LLH_LEN - 40.

5.16.1.9 #define UIP_TIME_WAIT_TIMEOUT 120

How long a connection should stay in the TIME_WAIT state.

This configiration option has no real implication, and it should be left untouched.

5.16.1.10 #define UIP_URGDATA

Determines if support for TCP urgent data notification should be compiled in.

Urgent data (out-of-band data) is a rarely used TCP feature that very seldom would be required.

5.17 ARP configuration options

Defines

- #define UIP_ARPTAB_SIZE

  The size of the ARP table.

- #define UIP_ARP_MAXAGE 120

  The maxium age of ARP table entries measured in 10ths of seconds.

5.17.1 Define Documentation

5.17.1.1 #define UIP_ARP_MAXAGE 120

The maxium age of ARP table entries measured in 10ths of seconds.

An UIP_ARP_MAXAGE of 120 corresponds to 20 minutes (BSD default).

5.17.1.2 #define UIP_ARPTAB_SIZE

The size of the ARP table.

This option should be set to a larger value if this uIP node will have many connections from the local network.

5.18 General configuration options

Defines

- #define UIP_BUFSIZE

  The size of the uIP packet buffer.

- #define UIP_STATISTICS

  Determines if statistics support should be compiled in.

- #define UIP_LOGGING

  Determines if logging of certain events should be compiled in.

- #define UIP_LLH_LEN

  The link level header length.

Functions

- void uip_log (char ∗msg)

  Print out a uIP log message.

5.18.1 Define Documentation

5.18.1.1 #define UIP_BUFSIZE

The size of the uIP packet buffer.

The uIP packet buffer should not be smaller than 60 bytes, and does not need to be larger than 1500 bytes. Lower size results in lower TCP throughput, larger size results in higher TCP throughput.

5.18.1.2 #define 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. For SLIP, this should be set to 0.

5.18.1.3 #define UIP_LOGGING

Determines if logging of certain events should be compiled in.

This is useful mostly for debugging. The function uip_log() must be implemented to suit the architecture of the project, if logging is turned on.

5.18.1.4 #define UIP_STATISTICS

Determines if statistics support should be compiled in.

The statistics is useful for debugging and to show the user.

5.18.2 Function Documentation

5.18.2.1 void uip_log (char * *msg*)

Print out a uIP log message.

This function must be implemented by the module that uses uIP, and is called by uIP whenever a log message is generated.

5.19 CPU architecture configuration

5.19.1 Detailed Description

The CPU architecture configuration is where the endianess of the CPU on which uIP is to be run is specified. Most CPUs today are little endian, and the most notable exception are the Motorolas which are big endian. The BYTE_ORDER macro should be changed to reflect the CPU architecture on which uIP is to be run.

Defines

- #define BYTE_ORDER

  The byte order of the CPU architecture on which uIP is to be run.

5.19.2 Define Documentation

5.19.2.1 #define BYTE_ORDER

The byte order of the CPU architecture on which uIP is to be run.

This option can be either BIG_ENDIAN (Motorola byte order) or LITTLE_ENDIAN (Intel byte order).

5.20 Appication specific configurations

5.20.1 Detailed Description

An uIP application is implemented using a single application function that is called by uIP whenever a TCP/IP event occurs. The name of this function must be registered with uIP at compile time using the UIP_APPCALL definition.

uIP applications can store the application state within the uip_conn structure by specifying the size of the application structure with the UIP_APPSTATE_SIZE macro.

The file containing the definitions must be included in the uipopt.h file.

The following example illustrates how this can look.

```
void httpd_appcall(void);
#define UIP_APPCALL     httpd_appcall

struct httpd_state {
  u8_t state;
  u16_t count;
  char *dataptr;
  char *script;
};
#define UIP_APPSTATE_SIZE (sizeof(struct httpd_state))
```

Defines

- #define UIP_APPCALL smtp_appcall

  The name of the application function that uIP should call in response to TCP/IP events.

- #define UIP_APPSTATE_SIZE (sizeof(struct smtp_state))

  The size of the application state that is to be stored in the uip_conn structure.

5.21 Web client

5.21.1 Detailed Description

This example shows a HTTP client that is able to download web pages and files from web servers. It requires a number of callback functions to be implemented by the module that utilizes the code: webclient_datahandler(), webclient_connected(), webclient_timedout(), webclient_aborted(), webclient_closed().

Files

- file webclient.c

  Implementation of the HTTP client.

- file webclient.h

  Header file for the HTTP client.

Functions

- void webclient_datahandler (char ∗data, u16_t len)

  Callback function that is called from the webclient code when HTTP data has been received.

- void webclient_connected (void)

  Callback function that is called from the webclient code when the HTTP connection has been connected to the web server.

- void webclient_timedout (void)

  Callback function that is called from the webclient code if the HTTP connection to the web server has timed out.

- void webclient_aborted (void)

  Callback function that is called from the webclient code if the HTTP connection to the web server has been aborted by the web server.

- void webclient_closed (void)

  Callback function that is called from the webclient code when the HTTP connection to the web server has been closed.

- void webclient_init (void)

  Initialize the webclient module.

- unsigned char webclient_get (char ∗host, u16_t port, char ∗file)

  Open an HTTP connection to a web server and ask for a file using the GET method.

- void webclient_close (void)

  Close the currently open HTTP connection.

- char ∗ webclient_mimetype (void)

  Obtain the MIME type of the current HTTP data stream.

- char ∗ webclient_filename (void)

  Obtain the filename of the current HTTP data stream.

- char ∗ webclient_hostname (void)

  Obtain the hostname of the current HTTP data stream.

- unsigned short webclient_port (void)

  Obtain the port number of the current HTTP data stream.

5.21.2 Function Documentation

5.21.2.1 void webclient_aborted (void)

Callback function that is called from the webclient code if the HTTP connection to the web server has been aborted by the web server.

This function must be implemented by the module that uses the webclient code.

5.21.2.2 void webclient_closed (void)

Callback function that is called from the webclient code when the HTTP connection to the web server has been closed.

This function must be implemented by the module that uses the webclient code.

5.21.2.3 void webclient_connected (void)

Callback function that is called from the webclient code when the HTTP connection has been connected to the web server.

This function must be implemented by the module that uses the webclient code.

5.21.2.4 void webclient_datahandler (char ∗ *data*, u16_t *len*)

Callback function that is called from the webclient code when HTTP data has been received.

This function must be implemented by the module that uses the webclient code. The function is called from the webclient module when HTTP data has been received. The function is not called when HTTP headers are received, only for the actual data.

Note:

> This function is called many times, repetedly, when data is being received, and not once when all data has been received.

Parameters:

data	A pointer to the data that has been received.
len	The length of the data that has been received.

5.21.2.5 char∗ webclient_filename (void)

Obtain the filename of the current HTTP data stream.

The filename of an HTTP request may be changed by the web server, and may therefore not be the same as when the original GET request was made with webclient_get(). This function is used for obtaining the current filename.

Returns:

A pointer to the current filename.

5.21.2.6 unsigned char webclient_get (char ∗ *host*, u16_t *port*, char ∗ *file*)

Open an HTTP connection to a web server and ask for a file using the GET method.

This function opens an HTTP connection to the specified web server and requests the specified file using the GET method. When the HTTP connection has been connected, the webclient_connected() callback function is called and when the HTTP data arrives the webclient_datahandler() callback function is called.

The callback function webclient_timedout() is called if the web server could not be contacted, and the webclient_aborted() callback function is called if the HTTP connection is aborted by the web server.

When the HTTP request has been completed and the HTTP connection is closed, the webclient_closed() callback function will be called.

Note:

If the function is passed a host name, it must already be in the resolver cache in order for the function to connect to the web server. It is therefore up to the calling module to implement the resolver calls and the signal handler used for reporting a resolv query answer.

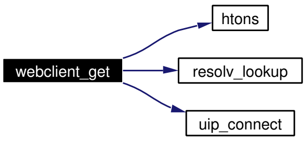
Parameters:

host	A pointer to a string containing either a host name or a numerical IP address in dotted decimal notation (e.g., 192.168.23.1).
port	The port number to which to connect, in host byte order.
file	A pointer to the name of the file to get.

Return values:

0	if the host name could not be found in the cache, or if a TCP connection could not be created.
1	if the connection was initiated.

Here is the call graph for this function:



5.21.2.7 char∗ webclient_hostname (void)

Obtain the hostname of the current HTTP data stream.

The hostname of the web server of an HTTP request may be changed by the web server, and may therefore not be the same as when the original GET request was made with webclient_get(). This function is used for obtaining the current hostname.

Returns:
A pointer to the current hostname.

5.21.2.8 char∗ webclient_mimetype (void)

Obtain the MIME type of the current HTTP data stream.

Returns:
A pointer to a string contaning the MIME type. The string may be empty if no MIME type was reported by the web server.

5.21.2.9 unsigned short webclient_port (void)

Obtain the port number of the current HTTP data stream.

The port number of an HTTP request may be changed by the web server, and may therefore not be the same as when the original GET request was made with webclient_get(). This function is used for obtaining the current port number.

Returns:
The port number of the current HTTP data stream, in host byte order.

5.21.2.10 void webclient_timedout (void)

Callback function that is called from the webclient code if the HTTP connection to the web server has timed out.

This function must be implemented by the module that uses the webclient code.

5.22 SMTP E-mail sender

5.22.1 Detailed Description

The Simple Mail Transfer Protocol (SMTP) as defined by RFC821 is the standard way of sending and transfering e-mail on the Internet. This simple example implementation is intended as an example of how to implement protocols in uIP, and is able to send out e-mail but has not been extensively tested.

Files

- file smtp.c

  SMTP example implementation.

- file smtp.h

  SMTP header file.

Defines

- #define SMTP_ERR_OK 0

  Error number that signifies a non-error condition.

Functions

- void smtp_done (unsigned char error)

  Callback function that is called when an e-mail transmission is done.

- void smtp_configure (char ∗localhostname, u16_t ∗smtpserver)

  Specificy an SMTP server and hostname.

- unsigned char smtp_send (char ∗to, char ∗from, char ∗subject, char ∗msg, u16_t msglen)

  Send an e-mail.

5.22.2 Function Documentation

5.22.2.1 void smtp_configure (char ∗ *lhostname*, u16_t ∗ *server*)

Specificy an SMTP server and hostname.

This function is used to configure the SMTP module with an SMTP server and the hostname of the host.

Parameters:

lhostname The hostname of the uIP host.

server A pointer to a 4-byte array representing the IP address of the SMTP server to be configured.

5.22.2.2 void smtp_done (unsigned char *error*)

Callback function that is called when an e-mail transmission is done.

This function must be implemented by the module that uses the SMTP module.

Parameters:

error The number of the error if an error occured, or SMTP_ERR_OK.

5.22.2.3 unsigned char smtp_send (char ∗ *to*, char ∗ *from*, char ∗ *subject*, char ∗ *msg*, u16_t *msglen*)

Send an e-mail.

Parameters:

to The e-mail address of the receiver of the e-mail.

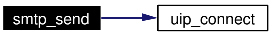
from The e-mail address of the sender of the e-mail.

subject The subject of the e-mail.

msg The actual e-mail message.

msglen The length of the e-mail message.

Here is the call graph for this function:



5.23 Telnet server

5.23.1 Detailed Description

The uIP telnet server provides a command based interface to uIP. It allows using the ”telnet” application to access uIP, and implements the required telnet option negotiation.

The code is structured in a way which makes it possible to add commands without having to rewrite the main telnet code. The main telnet code calls two callback functions, telnetd_connected() and telnetd_input(), when a telnet connection has been established and when a line of text arrives on a telnet connection. These two functions can be implemented in a way which suits the particular application or environment in which the uIP system is intended to be run.

The uIP distribution contains an example telnet shell implementation that provides a basic set of commands.

Files

- file telnetd-shell.c

  An example telnet server shell.

- file telnetd.c

  Implementation of the Telnet server.

- file telnetd.h

  Header file for the telnet server.

Data Structures

- struct telnetd_state

  A telnet connection structure.

Defines

- #define TELNETD_LINELEN

  The maximum length of a telnet line.

- #define TELNETD_NUMLINES

  The number of output lines being buffered for all telnet connections.

Functions

- void telnetd_connected (struct telnetd_state ∗s)

  Callback function that is called when a telnet connection has been established.

- void telnetd_input (struct telnetd_state ∗s, char ∗cmd)

  Callback function that is called when a line of text has arrived on a telnet connection.

- void telnetd_close (struct telnetd_state ∗s)

  Close a telnet session.

- void telnetd_output (struct telnetd_state ∗s, char ∗s1, char ∗s2)

  Print out a string on a telnet connection.

- void telnetd_prompt (struct telnetd_state ∗s, char ∗str)

  Print a prompt on a telnet connection.

- void telnetd_init (void)

  Initialize the telnet server.

5.23.2 Function Documentation

5.23.2.1 void telnetd_close (struct telnetd_state ∗ *s*)

Close a telnet session.

This function can be called from a telnet command in order to close the connection.

Parameters:

s The connection which is to be closed.

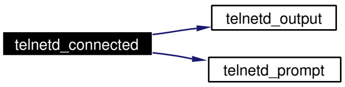
5.23.2.2 void telnetd_connected (struct telnetd_state ∗ *s*)

Callback function that is called when a telnet connection has been established.

Parameters:

s The telnet connection.

Here is the call graph for this function:

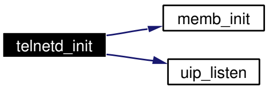


5.23.2.3 void telnetd_init (void)

Initialize the telnet server.

This function will perform the necessary initializations and start listening on TCP port 23.

Here is the call graph for this function:



5.23.2.4 void telnetd_input (struct telnetd_state ∗ *s*, char ∗ *cmd*)

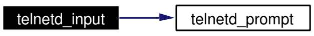
Callback function that is called when a line of text has arrived on a telnet connection.

Parameters:

s The telnet connection.

cmd The line of text.

Here is the call graph for this function:



5.23.2.5 void telnetd_output (struct telnetd_state ∗ *s*, char ∗ *str1*, char ∗ *str2*)

Print out a string on a telnet connection.

This function can be called from a telnet command parser in order to print out a string of text on the connection. The two strings given as arguments to the function will be concatenated, a carrige return and a new line character will be added, and the line is sent.

Parameters:

s The telnet connection.

str1 The first string.

str2 The second string.

5.23.2.6 void telnetd_prompt (struct telnetd_state ∗ *s*, char ∗ *str*)

Print a prompt on a telnet connection.

This function can be called by the telnet command shell in order to print out a command prompt.

Parameters:

s A telnet connection.

str The command prompt.

5.24 Web server

5.24.1 Detailed Description

The uIP web server is a very simplistic implementation of an HTTP server. It can serve web pages and files from a read-only ROM filesystem, and provides a very small scripting language.

The script language is very simple and works as follows. Each script line starts with a command character, either ”i”, ”t”, ”c”, ”#” or ”.”. The ”i” command tells the script interpreter to ”include” a file from the virtual file system and output it to the web browser. The ”t” command should be followed by a line of text that is to be output to the browser. The ”c” command is used to call one of the C functions from the httpd-cgi.c file. A line that starts with a ”#” is ignored (i.e., the ”#” denotes a comment), and the ”.” denotes the last script line.

The script that produces the file statistics page looks somewhat like this:

```
i /header.html
t <h1>File statistics</h1><br><table width="100%">
t <tr><td><a href="/index.html">/index.html</a></td><td>
c a /index.html
t </td></tr> <tr><td><a href="/cgi/files">/cgi/files</a></td><td>
c a /cgi/files
t </td></tr> <tr><td><a href="/cgi/tcp">/cgi/tcp</a></td><td>
c a /cgi/tcp
t </td></tr> <tr><td><a href="/404.html">/404.html</a></td><td>
c a /404.html
t </td></tr></table>
i /footer.plain
.
```

Files

- file cgi.c

  HTTP server script language C functions file.

- file cgi.h

  HTTP script language header file.

- file fs.c

  HTTP server read-only file system code.

- file fs.h

  HTTP server read-only file system header file.

- file httpd.c

  HTTP server.

- file httpd.h

  HTTP server header file.

Data Structures

- struct fs_file

An open file in the read-only file system.

Functions

- void httpd_init (void)

  Initialize the web server.

- int fs_open (const char ∗name, struct fs_file ∗file)

  Open a file in the read-only file system.

- void fs_init (void)

  Initialize the read-only file system.

Variables

- cgifunction cgitab [ ]

  A table containing pointers to C functions that can be called from a web server script.

5.24.2 Function Documentation

5.24.2.1 int fs_open (const char ∗ *name*, struct fs_file ∗ *file*)

Open a file in the read-only file system.

Parameters:

name The name of the file.

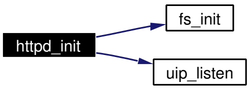
file The file pointer, which must be allocated by caller and will be filled in by the function.

5.24.2.2 void httpd_init (void)

Initialize the web server.

Starts to listen for incoming connection requests on TCP port 80.

Here is the call graph for this function:



5.25 uIP hostname resolver functions

5.25.1 Detailed Description

The uIP DNS resolver functions are used to lookup a hostname and map it to a numerical IP address. It maintains a list of resolved hostnames that can be queried with the resolv_lookup() function. New hostnames can be resolved using the resolv_query() function.

When a hostname has been resolved (or found to be non-existant), the resolver code calls a callback function called resolv_found() that must be implemented by the module that uses the resolver.

Files

- file resolv.c

  DNS host name to IP address resolver.

- file resolv.h

  DNS resolver code header file.

Functions

- void resolv_found (char ∗name, u16_t ∗ipaddr)

  Callback function which is called when a hostname is found.

- void resolv_conf (u16_t ∗dnsserver)

  Configure which DNS server to use for queries.

- u16_t ∗ resolv_getserver (void)

  Obtain the currently configured DNS server.

- void resolv_init (void)

  Initalize the resolver.

- u16_t ∗ resolv_lookup (char ∗name)

  Look up a hostname in the array of known hostnames.

- void resolv_query (char ∗name)

  Queues a name so that a question for the name will be sent out.

5.25.2 Function Documentation

5.25.2.1 void resolv_conf (u16_t ∗ *dnsserver*)

Configure which DNS server to use for queries.

Parameters:

dnsserver A pointer to a 4-byte representation of the IP address of the DNS server to be configured.

Here is the call graph for this function:

resolv_conf
uip_udp_new

5.25.2.2 void resolv_found (char ∗ *name*, u16_t ∗ *ipaddr*)

Callback function which is called when a hostname is found.

This function must be implemented by the module that uses the DNS resolver. It is called when a hostname is found, or when a hostname was not found.

Parameters:

name A pointer to the name that was looked up.

ipaddr A pointer to a 4-byte array containing the IP address of the hostname, or NULL if the hostname could not be found.

5.25.2.3 u16_t∗ resolv_getserver (void)

Obtain the currently configured DNS server.

Returns:

A pointer to a 4-byte representation of the IP address of the currently configured DNS server or NULL if no DNS server has been configured.

5.25.2.4 u16_t∗ resolv_lookup (char ∗ *name*)

Look up a hostname in the array of known hostnames.

Note:

This function only looks in the internal array of known hostnames, it does not send out a query for the hostname if none was found. The function resolv_query() can be used to send a query for a hostname.

Returns:

A pointer to a 4-byte representation of the hostname's IP address, or NULL if the hostname was not found in the array of hostnames.

5.25.2.5 void resolv_query (char ∗ *name*)

Queues a name so that a question for the name will be sent out.

Parameters:

name The hostname that is to be queried.

Chapter 6

uIP 0.9 Data Structure Documentation

6.1 fs_file Struct Reference

`#include <fs.h>`

6.1.1 Detailed Description

An open file in the read-only file system.

Data Fields

- char * data

  The actual file data.

- int len

  The length of the file data.

6.2 telnetd_state Struct Reference

`#include <telnetd.h>`

6.2.1 Detailed Description

A telnet connection structure.

6.3 uip_conn Struct Reference

`#include <uip.h>`

6.3.1 Detailed Description

Representation of a uIP TCP connection.

The uip_conn structure is used for identifying a connection. All but one field in the structure are to be considered read-only by an application. The only exception is the appstate field whos purpose is to let the application store application-specific state (e.g., file pointers) for the connection. The size of this field is configured in the ”uipopt.h” header file.

Data Fields

- u16_t ripaddr [2]

  The IP address of the remote host.

- u16_t lport

  The local TCP port, in network byte order.

- u16_t rport

  The local remote TCP port, in network byte order.

- 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.

- u16_t len

  Length of the data that was previously sent.

- u16_t mss

  Current maximum segment size for the connection.

- u16_t initialmss

  Initial maximum segment size for the connection.

- u8_t sa

  Retransmission time-out calculation state variable.

- u8_t sv

  Retransmission time-out calculation state variable.

- u8_t rto

  Retransmission time-out.

- u8_t tcpstateflags

TCP state and flags.

- u8_t timer

  The retransmission timer.

- u8_t nrtx

  The number of retransmissions for the last segment sent.

- u8_t appstate [UIP_APPSTATE_SIZE]

  The application state.

6.4 uip_eth_addr Struct Reference

`#include <uip_arp.h>`

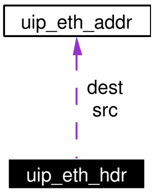
6.4.1 Detailed Description

Representation of a 48-bit Ethernet address.

6.5 uip_eth_hdr Struct Reference

`#include <uip_arp.h>`

Collaboration diagram for uip_eth_hdr:



6.5.1 Detailed Description

The Ethernet header.

6.6 uip_stats Struct Reference

`#include <uip.h>`

6.6.1 Detailed Description

The structure holding the TCP/IP statistics that are gathered if UIP_STATISTICS is set to 1.

Data Fields

- struct {
  uip_stats_t drop
  uip_stats_t recv
  uip_stats_t sent
  uip_stats_t vhlerr
  uip_stats_t hblenerr
  uip_stats_t lblenerr
  uip_stats_t fragerr
  uip_stats_t chkerr
  uip_stats_t protoerr
  } ip

  IP statistics.

- struct {
  uip_stats_t drop
  uip_stats_t recv
  uip_stats_t sent
  uip_stats_t typeerr
  } icmp

  ICMP statistics.

- struct {
  uip_stats_t drop
  uip_stats_t recv
  uip_stats_t sent
  uip_stats_t chkerr
  uip_stats_t ackerr
  uip_stats_t rst
  uip_stats_t rexmit
  uip_stats_t syndrop
  uip_stats_t synrst
  } tcp

  TCP statistics.

6.6.2 Field Documentation

6.6.2.1 uip_stats_t uip_stats::ackerr

Number of TCP segments with a bad ACK number.

6.6.2.2 uip_stats_t uip_stats::chkerr

Number of TCP segments with a bad checksum.

6.6.2.3 uip_stats_t uip_stats::drop

Number of dropped TCP segments.

6.6.2.4 uip_stats_t uip_stats::fragerr

Number of packets dropped since they were IP fragments.

6.6.2.5 uip_stats_t uip_stats::hblenerr

Number of packets dropped due to wrong IP length, high byte.

6.6.2.6 uip_stats_t uip_stats::lblenerr

Number of packets dropped due to wrong IP length, low byte.

6.6.2.7 uip_stats_t uip_stats::protoerr

Number of packets dropped since they were neither ICMP, UDP nor TCP.

6.6.2.8 uip_stats_t uip_stats::recv

Number of recived TCP segments.

6.6.2.9 uip_stats_t uip_stats::rexmit

Number of retransmitted TCP segments.

6.6.2.10 uip_stats_t uip_stats::rst

Number of recevied TCP RST (reset) segments.

6.6.2.11 uip_stats_t uip_stats::sent

Number of sent TCP segments.

6.6.2.12 uip_stats_t uip_stats::syndrop

Number of dropped SYNs due to too few connections was avaliable.

6.6.2.13 uip_stats_t uip_stats::synrst

Number of SYNs for closed ports, triggering a RST.

6.6.2.14 uip_stats_t uip_stats::typeerr

Number of ICMP packets with a wrong type.

6.6.2.15 uip_stats_t uip_stats::vhlerr

Number of packets dropped due to wrong IP version or header length.

6.7 uip_udp_conn Struct Reference

`#include <uip.h>`

6.7.1 Detailed Description

Representation of a uIP UDP connection.

Data Fields

- u16_t ripaddr [2]

  The IP address of the remote peer.

- u16_t lport

  The local port number in network byte order.

- u16_t rport

  The remote port number in network byte order.

Chapter 7

uIP 0.9 File Documentation

7.1 apps/httpd/cgi.c File Reference

7.1.1 Detailed Description

HTTP server script language C functions file.

Author:

Adam Dunkels <adam@dunkels.com>

This file contains functions that are called by the web server scripts. The functions takes one argument, and the return value is interpreted as follows. A zero means that the function did not complete and should be invoked for the next packet as well. A non-zero value indicates that the function has completed and that the web server should move along to the next script line.

`#include "uip.h"`

`#include "cgi.h"`

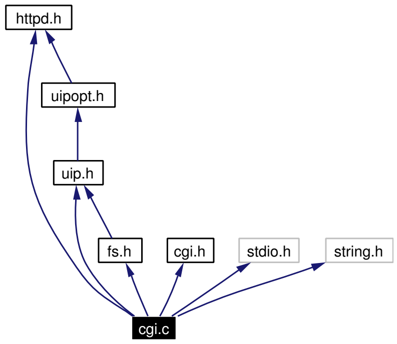
`#include "httpd.h"`

`#include "fs.h"`

`#include <stdio.h>`

`#include <string.h>`

Include dependency graph for cgi.c:



Variables

- cgifunction cgitab [ ]

  A table containing pointers to C functions that can be called from a web server script.

7.2 apps/httpd/cgi.h File Reference

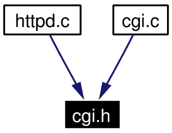
7.2.1 Detailed Description

HTTP script language header file.

Author:

Adam Dunkels <adam@dunkels.com>

This graph shows which files directly or indirectly include this file:



Variables

- cgifunction cgitab [ ]

  A table containing pointers to C functions that can be called from a web server script.

7.3 apps/httpd/fs.c File Reference

7.3.1 Detailed Description

HTTP server read-only file system code.

Author:
Adam Dunkels <adam@dunkels.com>

A simple read-only filesystem.

`#include "uip.h"`

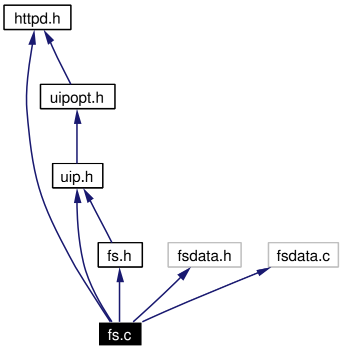
`#include "httpd.h"`

`#include "fs.h"`

`#include "fsdata.h"`

`#include "fsdata.c"`

Include dependency graph for fs.c:



Functions

- int fs_open (const char ∗name, struct fs_file ∗file)

  Open a file in the read-only file system.

- void fs_init (void)

  Initialize the read-only file system.

7.4 apps/httpd/fs.h File Reference

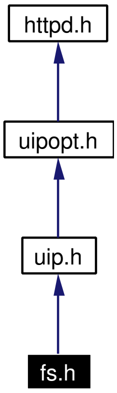
7.4.1 Detailed Description

HTTP server read-only file system header file.

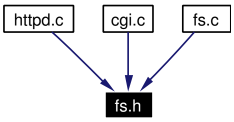
Author:
Adam Dunkels <`adam@dunkels.com`>

`#include "uip.h"`

Include dependency graph for fs.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct fs_file

  An open file in the read-only file system.

Functions

- int fs_open (const char ∗name, struct fs_file ∗file)

  Open a file in the read-only file system.

- void fs_init (void)

  Initialize the read-only file system.

7.5 apps/httpd/httpd.c File Reference

7.5.1 Detailed Description

HTTP server.

Author:

Adam Dunkels <`adam@dunkels.com`>

`#include "uip.h"`

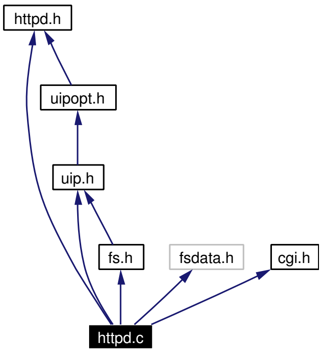
`#include "httpd.h"`

`#include "fs.h"`

`#include "fsdata.h"`

`#include "cgi.h"`

Include dependency graph for httpd.c:



Functions

- void httpd_init (void)

  Initialize the web server.

7.6 apps/httpd/httpd.h File Reference

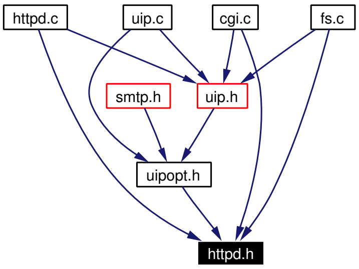
7.6.1 Detailed Description

HTTP server header file.

Author:

Adam Dunkels <`adam@dunkels.com`>

This graph shows which files directly or indirectly include this file:



Functions

- void httpd_init (void)

  Initialize the web server.

7.7 apps/resolv/resolv.c File Reference

7.7.1 Detailed Description

DNS host name to IP address resolver.

Author:

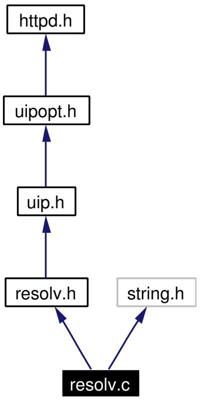
Adam Dunkels <`adam@dunkels.com`>

This file implements a DNS host name to IP address resolver.

`#include "resolv.h"`

`#include <string.h>`

Include dependency graph for resolv.c:



Functions

- void resolv_query (char ∗name)

  Queues a name so that a question for the name will be sent out.

- u16_t ∗ resolv_lookup (char ∗name)

  Look up a hostname in the array of known hostnames.

- u16_t ∗ resolv_getserver (void)

  Obtain the currently configured DNS server.

- void resolv_conf (u16_t ∗dnsserver)

  Configure which DNS server to use for queries.

- void resolv_init (void)

  Initalize the resolver.

7.8 apps/resolv/resolv.h File Reference

7.8.1 Detailed Description

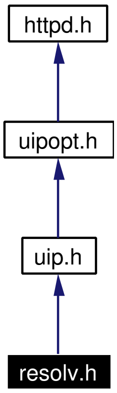
DNS resolver code header file.

Author:

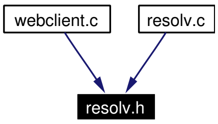
Adam Dunkels <`adam@dunkels.com`>

`#include "uip.h"`

Include dependency graph for resolv.h:



This graph shows which files directly or indirectly include this file:



Functions

- void resolv_found (char ∗name, u16_t ∗ipaddr)

  Callback function which is called when a hostname is found.

- void resolv_conf (u16_t ∗dnsserver)

  Configure which DNS server to use for queries.

- u16_t ∗ resolv_getserver (void)

  Obtain the currently configured DNS server.

- void resolv_init (void)

  Initalize the resolver.

- u16_t ∗ resolv_lookup (char ∗name)

  Look up a hostname in the array of known hostnames.

- void resolv_query (char ∗name)

  Queues a name so that a question for the name will be sent out.

7.9 apps/smtp/smtp.c File Reference

7.9.1 Detailed Description

SMTP example implementation.

Author:

Adam Dunkels <adam@dunkels.com>

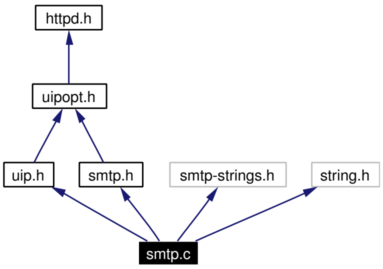
`#include "uip.h"`

`#include "smtp.h"`

`#include "smtp-strings.h"`

`#include <string.h>`

Include dependency graph for smtp.c:



Functions

- unsigned char smtp_send (char ∗to, char ∗from, char ∗subject, char ∗msg, u16_t msglen)

  Send an e-mail.

- void smtp_configure (char ∗lhostname, u16_t ∗server)

  Specificy an SMTP server and hostname.

7.10 apps/smtp/smtp.h File Reference

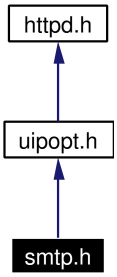
7.10.1 Detailed Description

SMTP header file.

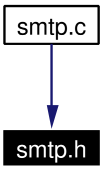
Author:
Adam Dunkels <`adam@dunkels.com`>

```
#include "uipopt.h"
```

Include dependency graph for smtp.h:



This graph shows which files directly or indirectly include this file:



Defines

- #define SMTP_ERR_OK 0

  Error number that signifies a non-error condition.

- #define UIP_APPCALL smtp_appcall

  The name of the application function that uIP should call in response to TCP/IP events.

- #define UIP_APPSTATE_SIZE (sizeof(struct smtp_state))

  The size of the application state that is to be stored in the uip_conn structure.

Functions

- void smtp_done (unsigned char error)

  Callback function that is called when an e-mail transmission is done.

- void smtp_configure (char ∗localhostname, u16_t ∗smtpserver)

  Specificy an SMTP server and hostname.

- unsigned char smtp_send (char ∗to, char ∗from, char ∗subject, char ∗msg, u16_t msglen)

  Send an e-mail.

7.11 apps/telnetd/memb.c File Reference

7.11.1 Detailed Description

Memory block allocation routines.

Author:

Adam Dunkels <adam@sics.se>

The memory block allocation routines provide a simple yet powerful set of functions for managing a set of memory blocks of fixed size. A set of memory blocks is statically declared with the MEMB() macro. Memory blocks are allocated from the declared memory by the memb_alloc() function, and are deallocated with the memb_free() function.

Note:

Because of namespace clashes only one MEMB() can be declared per C module, and the name scope of a MEMB() memory block is local to each C module.

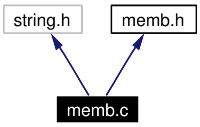
The following example shows how to declare and use a memory block called "cmem" which has 8 chunks of memory with each memory chunk being 20 bytes large.

```
MEMB(cmem, 20, 8);

int main(int argc, char *argv[]) {
   char *ptr;

   memb_init(&cmem);

   ptr = memb_alloc(&cmem);

   if(ptr != NULL) {
      do_something(ptr);
   } else {
      printf("Could not allocate memory.\n");
   }

   if(memb_free(ptr) == 0) {
      printf("Deallocation succeeded.\n");
   }
}
```

`#include <string.h>`

`#include "memb.h"`

Include dependency graph for memb.c:



Functions

- void memb_init (struct memb_blocks *m)

Initialize a memory block that was declared with MEMB().

- char ∗ memb_alloc (struct memb_blocks ∗m)

  Allocate a memory block from a block of memory declared with MEMB().

- char memb_free (struct memb_blocks ∗m, char ∗ptr)

  Deallocate a memory block from a memory block previously declared with MEMB().

- char memb_ref (struct memb_blocks ∗m, char ∗ptr)

  Increase the reference count for a memory chunk.

7.12 apps/telnetd/memb.h File Reference

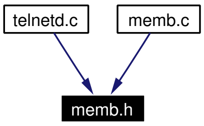
7.12.1 Detailed Description

Memory block allocation routines.

Author:

Adam Dunkels <adam@sics.se>

This graph shows which files directly or indirectly include this file:



Defines

- #define MEMB(name, size, num)

  Declare a memory block.

Functions

- void memb_init (struct memb_blocks ∗m)

  Initialize a memory block that was declared with MEMB().

- char ∗ memb_alloc (struct memb_blocks ∗m)

  Allocate a memory block from a block of memory declared with MEMB().

- char memb_ref (struct memb_blocks ∗m, char ∗ptr)

  Increase the reference count for a memory chunk.

- char memb_free (struct memb_blocks ∗m, char ∗ptr)

  Deallocate a memory block from a memory block previously declared with MEMB().

7.13 apps/telnetd/telnetd-shell.c File Reference

7.13.1 Detailed Description

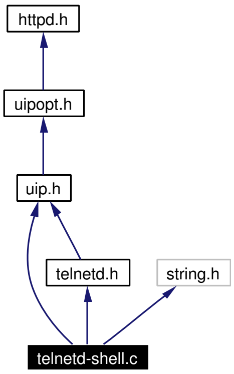
An example telnet server shell.

Author:

Adam Dunkels <adam@dunkels.com>

```
#include "uip.h"
#include "telnetd.h"
#include <string.h>
```

Include dependency graph for telnetd-shell.c:



Functions

- void telnetd_connected (struct telnetd_state ∗s)

  Callback function that is called when a telnet connection has been established.

- void telnetd_input (struct telnetd_state ∗s, char ∗cmd)

  Callback function that is called when a line of text has arrived on a telnet connection.

7.14 apps/telnetd/telnetd.c File Reference

7.14.1 Detailed Description

Implementation of the Telnet server.

Author:

Adam Dunkels <`adam@dunkels.com`>

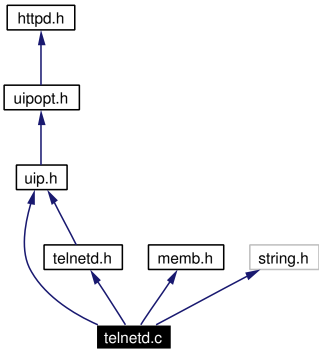
`#include "uip.h"`

`#include "memb.h"`

`#include "telnetd.h"`

`#include <string.h>`

Include dependency graph for telnetd.c:



Functions

- void telnetd_close (struct telnetd_state ∗s)

  Close a telnet session.

- void telnetd_prompt (struct telnetd_state ∗s, char ∗str)

  Print a prompt on a telnet connection.

- void telnetd_output (struct telnetd_state ∗s, char ∗str1, char ∗str2)

  Print out a string on a telnet connection.

- void telnetd_init (void)

  Initialize the telnet server.

7.15 apps/telnetd/telnetd.h File Reference

7.15.1 Detailed Description

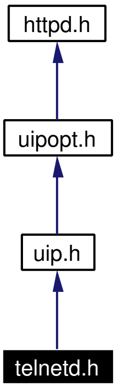
Header file for the telnet server.

Author:

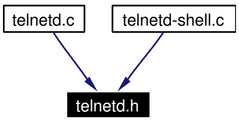
Adam Dunkels <`adam@dunkels.com`>

`#include "uip.h"`

Include dependency graph for telnetd.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct telnetd_state

  A telnet connection structure.

Defines

- #define TELNETD_LINELEN

  The maximum length of a telnet line.

- #define TELNETD_NUMLINES

  The number of output lines being buffered for all telnet connections.

Functions

- void telnetd_connected (struct telnetd_state ∗s)

  Callback function that is called when a telnet connection has been established.

- void telnetd_input (struct telnetd_state ∗s, char ∗cmd)

  Callback function that is called when a line of text has arrived on a telnet connection.

- void telnetd_close (struct telnetd_state ∗s)

  Close a telnet session.

- void telnetd_output (struct telnetd_state ∗s, char ∗s1, char ∗s2)

  Print out a string on a telnet connection.

- void telnetd_prompt (struct telnetd_state ∗s, char ∗str)

  Print a prompt on a telnet connection.

- void telnetd_init (void)

  Initialize the telnet server.

7.16 apps/webclient/webclient.c File Reference

7.16.1 Detailed Description

Implementation of the HTTP client.

Author:

Adam Dunkels <`adam@dunkels.com`>

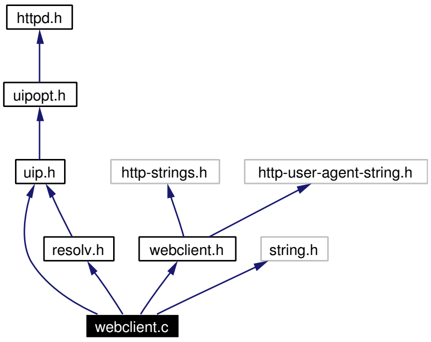
`#include "uip.h"`

`#include "webclient.h"`

`#include "resolv.h"`

`#include <string.h>`

Include dependency graph for webclient.c:



Functions

- char ∗ webclient_mimetype (void)

  Obtain the MIME type of the current HTTP data stream.

- char ∗ webclient_filename (void)

  Obtain the filename of the current HTTP data stream.

- char ∗ webclient_hostname (void)

  Obtain the hostname of the current HTTP data stream.

- unsigned short webclient_port (void)

  Obtain the port number of the current HTTP data stream.

- void webclient_init (void)

  Initialize the webclient module.

- void webclient_close (void)

  Close the currently open HTTP connection.

- unsigned char webclient_get (char ∗host, u16_t port, char ∗file)

Open an HTTP connection to a web server and ask for a file using the GET method.

7.17 apps/webclient/webclient.h File Reference

7.17.1 Detailed Description

Header file for the HTTP client.

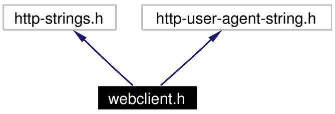
Author:

Adam Dunkels <`adam@dunkels.com`>

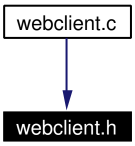
`#include "http-strings.h"`

`#include "http-user-agent-string.h"`

Include dependency graph for webclient.h:



This graph shows which files directly or indirectly include this file:



Functions

- void webclient_datahandler (char ∗data, u16_t len)

  Callback function that is called from the webclient code when HTTP data has been received.

- void webclient_connected (void)

  Callback function that is called from the webclient code when the HTTP connection has been connected to the web server.

- void webclient_timedout (void)

  Callback function that is called from the webclient code if the HTTP connection to the web server has timed out.

- void webclient_aborted (void)

  Callback function that is called from the webclient code if the HTTP connection to the web server has been aborted by the web server.

- void webclient_closed (void)

  Callback function that is called from the webclient code when the HTTP connection to the web server has been closed.

- void webclient_init (void)

  Initialize the webclient module.

- unsigned char webclient_get (char ∗host, u16_t port, char ∗file)

  Open an HTTP connection to a web server and ask for a file using the GET method.

- void webclient_close (void)

  Close the currently open HTTP connection.

- char ∗ webclient_mimetype (void)

  Obtain the MIME type of the current HTTP data stream.

- char ∗ webclient_filename (void)

  Obtain the filename of the current HTTP data stream.

- char ∗ webclient_hostname (void)

  Obtain the hostname of the current HTTP data stream.

- unsigned short webclient_port (void)

  Obtain the port number of the current HTTP data stream.

7.18 uip/slipdev.c File Reference

7.18.1 Detailed Description

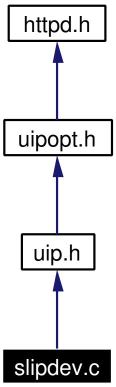
SLIP protocol implementation.

Author:

Adam Dunkels <`adam@dunkels.com`>

`#include "uip.h"`

Include dependency graph for slipdev.c:



Functions

- void slipdev_send (void)

  Send the packet in the uip_buf and uip_appdata buffers using the SLIP protocol.

- u16_t slipdev_poll (void)

  Poll the SLIP device for an available packet.

- void slipdev_init (void)

  Initialize the SLIP module.

7.19 uip/slipdev.h File Reference

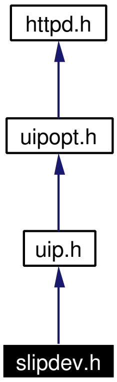
7.19.1 Detailed Description

SLIP header file.

Author:
Adam Dunkels <`adam@dunkels.com`>

`#include "uip.h"`

Include dependency graph for slipdev.h:



Functions

- void slipdev_char_put (u8_t c)

  Put a character on the serial device.

- u8_t slipdev_char_poll (u8_t ∗c)

  Poll the serial device for a character.

- void slipdev_init (void)

  Initialize the SLIP module.

- void slipdev_send (void)

  Send the packet in the uip_buf and uip_appdata buffers using the SLIP protocol.

- u16_t slipdev_poll (void)

  Poll the SLIP device for an available packet.

7.20 uip/uip.c File Reference

7.20.1 Detailed Description

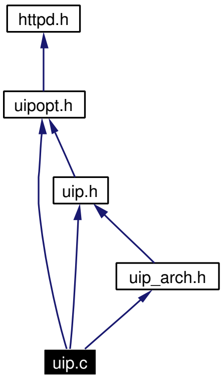
The uIP TCP/IP stack code.

Author:

Adam Dunkels <`adam@dunkels.com`>

```
#include "uip.h"
#include "uipopt.h"
#include "uip_arch.h"
```

Include dependency graph for uip.c:



Functions

- void uip_init (void)

  uIP initialization function.

- uip_udp_conn ∗ uip_udp_new (u16_t ∗ripaddr, u16_t rport)

  Set up a new UDP connection.

- void uip_unlisten (u16_t port)

  Stop listening to the specified port.

- void uip_listen (u16_t port)

  Start listening to the specified port.

- u16_t htons (u16_t val)

  Convert 16-bit quantity from host byte order to network byte order.

Variables

- u8_t uip_buf [UIP_BUFSIZE+2]

The uIP packet buffer.

- volatile u8_t ∗ uip_appdata

  Pointer to the application data in the packet buffer.

- volatile u8_t uip_acc32 [4]

  4-byte array used for the 32-bit sequence number calculations.

7.21 uip/uip.h File Reference

7.21.1 Detailed Description

Header file for the uIP TCP/IP stack.

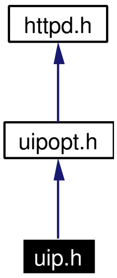
Author:

Adam Dunkels <`adam@dunkels.com`>

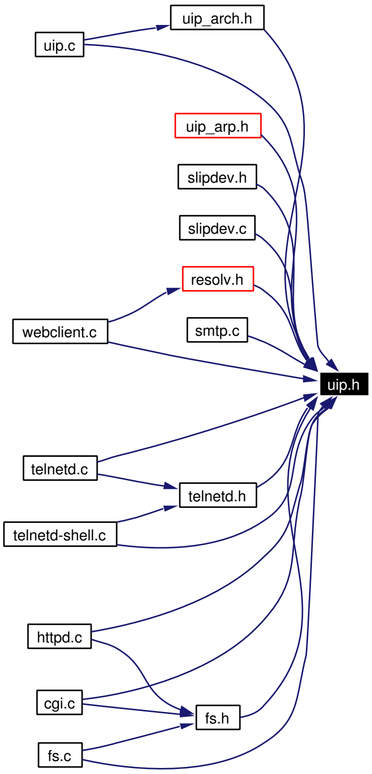
The uIP TCP/IP stack header file contains definitions for a number of C macros that are used by uIP programs as well as internal uIP structures, TCP/IP header structures and function declarations.

```
#include "uipopt.h"
```

Include dependency graph for uip.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct uip_conn

  Representation of a uIP TCP connection.

- struct uip_stats

  The structure holding the TCP/IP statistics that are gathered if UIP_STATISTICS is set to 1.

- struct uip_udp_conn

  Representation of a uIP UDP connection.

Defines

- #define uip_sethostaddr(addr)

  Set the IP address of this host.

- #define uip_gethostaddr(addr)

  Get the IP address of this host.

- #define uip_input()

  Process an incoming packet.

- #define uip_periodic(conn)

Periodic processing for a connection identified by its number.

- #define uip_periodic_conn(conn)

  Periodic processing for a connection identified by a pointer to its structure.

- #define uip_udp_periodic(conn)

  Periodic processing for a UDP connection identified by its number.

- #define uip_udp_periodic_conn(conn)

  Periodic processing for a UDP connection identified by a pointer to its structure.

- #define uip_send(data, len)

  Send data on the current connection.

- #define uip_datalen()

  The length of any incoming data that is currently avaliable (if avaliable) in the uip_appdata buffer.

- #define uip_urgdatalen()

  The length of any out-of-band data (urgent data) that has arrived on the connection.

- #define uip_close()

  Close the current connection.

- #define uip_abort()

  Abort the current connection.

- #define uip_stop()

  Tell the sending host to stop sending data.

- #define uip_stopped(conn)

  Find out if the current connection has been previously stopped with uip_stop().

- #define uip_restart()

  Restart the current connection, if is has previously been stopped with uip_stop().

- #define uip_newdata()

  Is new incoming data available?

- #define uip_acked()

  Has previously sent data been acknowledged?

- #define uip_connected()

  Has the connection just been connected?

- #define uip_closed()

  Has the connection been closed by the other end?

- #define uip_aborted()

  Has the connection been aborted by the other end?

- #define uip_timedout()

  Has the connection timed out?

- #define uip_rexmit()

  Do we need to retransmit previously data?

- #define uip_poll()

  Is the connection being polled by uIP?

- #define uip_initialmss()

  Get the initial maxium segment size (MSS) of the current connection.

- #define uip_mss()

  Get the current maxium segment size that can be sent on the current connection.

- #define uip_udp_remove(conn)

  Removed a UDP connection.

- #define uip_udp_send(len)

  Send a UDP datagram of length len on the current connection.

- #define uip_ipaddr(addr, addr0, addr1, addr2, addr3)

  Pack an IP address into a 4-byte array which is used by uIP to represent IP addresses.

- #define HTONS(n)

  Convert 16-bit quantity from host byte order to network byte order.

Functions

- void uip_init (void)

  uIP initialization function.

- void uip_listen (u16_t port)

  Start listening to the specified port.

- void uip_unlisten (u16_t port)

  Stop listening to the specified port.

- uip_conn ∗ uip_connect (u16_t ∗ripaddr, u16_t port)

  Connect to a remote host using TCP.

- uip_udp_conn ∗ uip_udp_new (u16_t ∗ripaddr, u16_t rport)

  Set up a new UDP connection.

- u16_t htons (u16_t val)

  Convert 16-bit quantity from host byte order to network byte order.

Variables

- u8_t uip_buf [UIP_BUFSIZE+2]

  The uIP packet buffer.

- volatile u8_t ∗ uip_appdata

  Pointer to the application data in the packet buffer.

- volatile u8_t uip_acc32 [4]

  4-byte array used for the 32-bit sequence number calculations.

- uip_stats uip_stat

  The uIP TCP/IP statistics.

7.22 uip/uip_arch.h File Reference

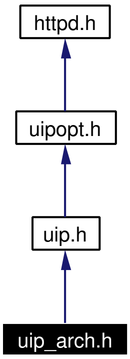
7.22.1 Detailed Description

Declarations of architecture specific functions.

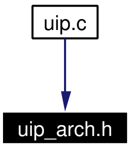
Author:
Adam Dunkels <`adam@dunkels.com`>

```
#include "uip.h"
```

Include dependency graph for uip_arch.h:



This graph shows which files directly or indirectly include this file:



Functions

- void uip_add32 (u8_t ∗op32, u16_t op16)
  Carry out a 32-bit addition.
- u16_t uip_chksum (u16_t ∗buf, u16_t len)
  Calculate the Internet checksum over a buffer.
- u16_t uip_ipchksum (void)
  Calculate the IP header checksum of the packet header in uip_buf.
- u16_t uip_tcpchksum (void)
  Calculate the TCP checksum of the packet in uip_buf and uip_appdata.

7.23 uip/uip_arp.c File Reference

7.23.1 Detailed Description

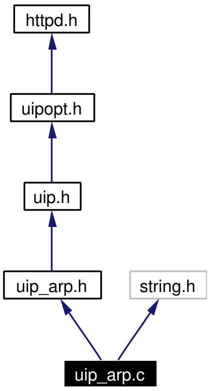
Implementation of the ARP Address Resolution Protocol.

Author:

Adam Dunkels <`adam@dunkels.com`>

```
#include "uip_arp.h"
#include <string.h>
```

Include dependency graph for uip_arp.c:



Functions

- void uip_arp_init (void)

  Initialize the ARP module.

- void uip_arp_timer (void)

  Periodic ARP processing function.

- void uip_arp_ipin (void)

  ARP processing for incoming IP packets.

- void uip_arp_arpin (void)

  ARP processing for incoming ARP packets.

- void uip_arp_out (void)

  Prepend Ethernet header to an outbound IP packet and see if we need to send out an ARP request.

7.24 uip/uip_arp.h File Reference

7.24.1 Detailed Description

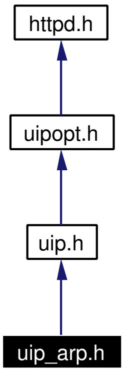
Macros and definitions for the ARP module.

Author:

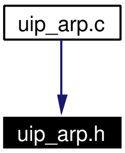
Adam Dunkels <`adam@dunkels.com`>

```
#include "uip.h"
```

Include dependency graph for uip_arp.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct uip_eth_addr

  Representation of a 48-bit Ethernet address.

- struct uip_eth_hdr

  The Ethernet header.

Defines

- #define uip_setdraddr(addr)

  Set the default router's IP address.

- #define uip_setnetmask(addr)

  Set the netmask.

- #define uip_getdraddr(addr)

  Get the default router's IP address.

- #define uip_getnetmask(addr)

  Get the netmask.

- #define uip_setethaddr(eaddr)

  Specifiy the Ethernet MAC address.

Functions

- void uip_arp_init (void)

  Initialize the ARP module.

- void uip_arp_ipin (void)

  ARP processing for incoming IP packets.

- void uip_arp_arpin (void)

  ARP processing for incoming ARP packets.

- void uip_arp_out (void)

  Prepend Ethernet header to an outbound IP packet and see if we need to send out an ARP request.

- void uip_arp_timer (void)

  Periodic ARP processing function.

7.25 unix/uipopt.h File Reference

7.25.1 Detailed Description

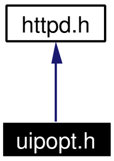
Configuration options for uIP.

Author:
Adam Dunkels <`adam@dunkels.com`>

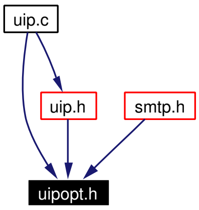
This file is used for tweaking various configuration options for uIP. You should make a copy of this file into one of your project's directories instead of editing this example "uipopt.h" file that comes with the uIP distribution.

`#include "httpd.h"`

Include dependency graph for uipopt.h:



This graph shows which files directly or indirectly include this file:



Defines

- #define UIP_FIXEDADDR

  Determines if uIP should use a fixed IP address or not.

- #define UIP_PINGADDRCONF

  Ping IP address asignment.

- #define UIP_IPADDR0

  The first octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_IPADDR1

  The second octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_IPADDR2

The third octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_IPADDR3

  The fourth octet of the IP address of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK0

  The first octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK1

  The second octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK2

  The third octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_NETMASK3

  The fourth octet of the netmask of this uIP node, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR0

  The first octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR1

  The second octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR2

  The third octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_DRIPADDR3

  The fourth octet of the IP address of the default router, if UIP_FIXEDADDR is 1.

- #define UIP_FIXEDETHADDR

  Specifies if the uIP ARP module should be compiled with a fixed Ethernet MAC address or not.

- #define UIP_ETHADDR0

  The first octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR1

  The second octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR2

  The third octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR3

  The fourth octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR4

  The fifth octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_ETHADDR5

  The sixth octet of the Ethernet address if UIP_FIXEDETHADDR is 1.

- #define UIP_TTL 255

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

- #define UIP_REASSEMBLY

  Turn on support for IP packet reassembly.

- #define UIP_REASS_MAXAGE 40

  The maximum time an IP fragment should wait in the reassembly buffer before it is dropped.

- #define UIP_UDP

  Toggles wether UDP support should be compiled in or not.

- #define UIP_UDP_CHECKSUMS

  Toggles if UDP checksums should be used or not.

- #define UIP_UDP_CONNS

  The maximum amount of concurrent UDP connections.

- #define UIP_UDP_APPCALL

  The name of the function that should be called when UDP datagrams arrive.

- #define UIP_ACTIVE_OPEN

  Determines if support for opening connections from uIP should be compiled in.

- #define UIP_CONNS

  The maximum number of simultaneously open TCP connections.

- #define UIP_LISTENPORTS

  The maximum number of simultaneously listening TCP ports.

- #define UIP_RECEIVE_WINDOW

  The size of the advertised receiver's window.

- #define UIP_URGDATA

  Determines if support for TCP urgent data notification should be compiled in.

- #define UIP_RTO 3

  The initial retransmission timeout counted in timer pulses.

- #define UIP_MAXRTX 8

  The maximum number of times a segment should be retransmitted before the connection should be aborted.

- #define UIP_MAXSYNRTX 3

  The maximum number of times a SYN segment should be retransmitted before a connection request should be deemed to have been unsuccessful.

- #define UIP_TCP_MSS (UIP_BUFSIZE - UIP_LLH_LEN - 40)

  The TCP maximum segment size.

- #define UIP_TIME_WAIT_TIMEOUT 120

How long a connection should stay in the TIME_WAIT state.

- #define UIP_ARPTAB_SIZE

  The size of the ARP table.

- #define UIP_ARP_MAXAGE 120

  The maxium age of ARP table entries measured in 10ths of seconds.

- #define UIP_BUFSIZE

  The size of the uIP packet buffer.

- #define UIP_STATISTICS

  Determines if statistics support should be compiled in.

- #define UIP_LOGGING

  Determines if logging of certain events should be compiled in.

- #define UIP_LLH_LEN

  The link level header length.

- #define BYTE_ORDER

  The byte order of the CPU architecture on which uIP is to be run.

Typedefs

- typedef unsigned char u8_t

  The 8-bit unsigned data type.

- typedef unsigned short u16_t

  The 16-bit unsigned data type.

- typedef unsigned short uip_stats_t

  The statistics data type.

Functions

- void uip_log (char ∗msg)

  Print out a uIP log message.

Index