

The uIP Embedded TCP/IP Stack

The uIP 1.0 Reference Manual

June 2006



Adam Dunkels
adam@sics.se

Swedish Institute of Computer Science

Contents

1	The uIP TCP/IP stack	1
1.1	Introduction	1
1.2	TCP/IP Communication	2
1.3	Main Control Loop	2
1.4	Architecture Specific Functions	3
1.5	Memory Management	3
1.6	Application Program Interface (API)	4
1.7	Examples	7
1.8	Protocol Implementations	14
1.9	Performance	16
2	uIP 1.0 Module Index	19
2.1	uIP 1.0 Modules	19
3	uIP 1.0 Hierarchical Index	21
3.1	uIP 1.0 Class Hierarchy	21
4	uIP 1.0 Data Structure Index	23
4.1	uIP 1.0 Data Structures	23
5	uIP 1.0 File Index	25
5.1	uIP 1.0 File List	25
6	uIP 1.0 Module Documentation	27
6.1	Protothreads	27
6.2	Applications	36
6.3	uIP configuration functions	37
6.4	uIP initialization functions	40
6.5	uIP device driver functions	41
6.6	uIP application functions	46

6.7	uIP conversion functions	55
6.8	Variables used in uIP device drivers	61
6.9	The uIP TCP/IP stack	62
6.10	Architecture specific uIP functions	74
6.11	uIP Address Resolution Protocol	76
6.12	Configuration options for uIP	79
6.13	uIP TCP throughput booster hack	89
6.14	Local continuations	90
6.15	Timer library	91
6.16	Clock interface	94
6.17	Protosockets library	95
6.18	Memory block management functions	102
6.19	DNS resolver	105
6.20	SMTP E-mail sender	108
6.21	Telnet server	110
6.22	Hello, world	113
6.23	Web client	114
6.24	Web server	120
7	uIP 1.0 Data Structure Documentation	123
7.1	dhcpc_state Struct Reference	123
7.2	hello_world_state Struct Reference	124
7.3	httpd_cgi_call Struct Reference	125
7.4	httpd_state Struct Reference	126
7.5	memb_blocks Struct Reference	127
7.6	psock Struct Reference	128
7.7	psock_buf Struct Reference	129
7.8	pt Struct Reference	130
7.9	smtp_state Struct Reference	131
7.10	telnetd_state Struct Reference	132
7.11	timer Struct Reference	133
7.12	uip_conn Struct Reference	134
7.13	uip_eth_addr Struct Reference	136
7.14	uip_eth_hdr Struct Reference	137
7.15	uip_icmpip_hdr Struct Reference	138
7.16	uip_neighbor_addr Struct Reference	139
7.17	uip_stats Struct Reference	140

7.18	uip_tcpip_hdr Struct Reference	142
7.19	uip_udp_conn Struct Reference	143
7.20	uip_udpip_hdr Struct Reference	144
7.21	webclient_state Struct Reference	145
8	uIP 1.0 File Documentation	147
8.1	apps/hello-world/hello-world.c File Reference	147
8.2	apps/hello-world/hello-world.h File Reference	148
8.3	apps/resolv/resolv.c File Reference	149
8.4	apps/resolv/resolv.h File Reference	151
8.5	apps/smtp/smtp.c File Reference	153
8.6	apps/smtp/smtp.h File Reference	154
8.7	apps/telnetd/shell.c File Reference	156
8.8	apps/telnetd/shell.h File Reference	157
8.9	apps/telnetd/telnetd.c File Reference	158
8.10	apps/telnetd/telnetd.h File Reference	159
8.11	apps/webclient/webclient.c File Reference	160
8.12	apps/webclient/webclient.h File Reference	162
8.13	apps/webserver/httpd-cgi.c File Reference	164
8.14	apps/webserver/httpd-cgi.h File Reference	165
8.15	apps/webserver/httpd.c File Reference	166
8.16	lib/memb.c File Reference	167
8.17	lib/memb.h File Reference	168
8.18	uip/lc-addrlabels.h File Reference	169
8.19	uip/lc-switch.h File Reference	170
8.20	uip/lc.h File Reference	171
8.21	uip/psock.h File Reference	172
8.22	uip/pt.h File Reference	174
8.23	uip/timer.c File Reference	176
8.24	uip/timer.h File Reference	177
8.25	uip/uip-neighbor.c File Reference	178
8.26	uip/uip-neighbor.h File Reference	179
8.27	uip/uip-split.h File Reference	180
8.28	uip/uip.c File Reference	181
8.29	uip/uip.h File Reference	184
8.30	uip/uip_arch.h File Reference	190
8.31	uip/uip_arp.c File Reference	191

8.32	uip/uip_arp.h File Reference	192
8.33	uip/uiptopt.h File Reference	193
8.34	unix/uip-conf.h File Reference	196
9	uIP 1.0 Example Documentation	197
9.1	dhcpc.c	197
9.2	dhcpc.h	203
9.3	example-mainloop-with-arp.c	205
9.4	example-mainloop-without-arp.c	207
9.5	hello-world.c	208
9.6	hello-world.h	210
9.7	resolv.c	211
9.8	resolv.h	218
9.9	smtp.c	220
9.10	smtp.h	224
9.11	telnetd.c	226
9.12	telnetd.h	232
9.13	uip-code-style.c	234
9.14	uip-conf.h	236
9.15	webclient.c	239
9.16	webclient.h	246

Chapter 1

The uIP TCP/IP stack

Author:

Adam Dunkels, <http://www.sics.se/~adam/>

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.

uIP can be found at the uIP web page: <http://www.sics.se/~adam/uip/>

See also:

[Application programs](#)

[Compile-time configuration options](#)

[Run-time configuration functions](#)

[Initialization functions](#)

[Device driver interface](#) and [variables used by device drivers](#)

[uIP functions called from application programs](#) (see below) and the [protosockets API](#) and their underlying [protothreads](#)

1.1 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 the IP, ICMP, UDP 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 Main Control Loop

The uIP stack can be run either as a task in a multitasking system, or as the main program in a singletasking system. In both cases, the main control loop does two things repeatedly:

- Check if a packet has arrived from the network.
- Check if a periodic timeout has occurred.

If a packet has arrived, the input handler function, `uip_input()`, should be invoked by the main control loop. The input handler function will never block, but will return at once. When it returns, the stack or the application for which the incoming packet was intended may have produced one or more reply packets which should be sent out. If so, the network device driver should be called to send out these packets.

Periodic timeouts are used to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. When the main control loop infers that the periodic timer should fire, it should invoke the timer handler function `uip_periodic()`. Because the TCP/IP stack may perform retransmissions when dealing with a timer event, the network device driver should be called to send out the packets that may have been produced.

1.4 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.

1.4.1 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.

While uIP includes a generic checksum function, it also leaves it open for an architecture specific implementation of 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.

1.4.2 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.

While uIP implements a generic 32-bit addition, there is support for having an architecture specific implementation of the `uip_add32()` function.

1.5 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.

1.6 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.

uIP provides two APIs to programmers: protosockets, a BSD socket-like API without the overhead of full multi-threading, and a "raw" event-based API that is more low-level than protosockets but uses less memory.

See also:

[Protosockets library](#)

[Protothreads](#)

1.6.1 The uIP raw API

The "raw" uIP API 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 re-transmissions. 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.

1.6.1.1 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).

1.6.1.2 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.

1.6.1.3 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 therefore have to either act directly on the incoming data, or by itself copy the incoming data into a buffer for later processing.

1.6.1.4 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.

1.6.1.5 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. 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.

1.6.1.6 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.

1.6.1.7 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.

1.6.1.8 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.

1.6.1.9 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.

1.6.1.10 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) {
    uip_addr_t ipaddr;

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

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 its 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) {
    uip_ipaddr_t ipaddr;
    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(uiplib_connected()) {
        connected();
    }
    if(uiplib_acked()) {
        acked();
    }
    if(uiplib_newdata()) {
        newdata();
    }
    if(uiplib_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 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 initializing 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()`, `timeout()` 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 acknowledged 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.

1.8 Protocol Implementations

The protocols in the TCP/IP protocol suite are designed in a layered fashion where each protocol performs a specific function and the interactions between the protocol layers are strictly defined. While the layered approach is a good way to design protocols, it is not always the best way to implement them. In uIP, the protocol implementations are tightly coupled in order to save code space.

This section gives detailed information on the specific protocol implementations in uIP.

1.8.1 IP — Internet Protocol

When incoming packets are processed by uIP, the IP layer is the first protocol that examines the packet. The IP layer does a few simple checks such as if the destination IP address of the incoming packet matches any of the local IP address and verifies the IP header checksum. Since there are no IP options that are strictly required and because they are very uncommon, any IP options in received packets are dropped.

1.8.1.1 IP Fragment Reassembly

IP fragment reassembly is implemented using a separate buffer that holds the packet to be reassembled. An incoming fragment is copied into the right place in the buffer and a bit map is used to keep track of which fragments have been received. Because the first byte of an IP fragment is aligned on an 8-byte boundary, the bit map requires a small amount of memory. When all fragments have been reassembled, the resulting IP packet is passed to the transport layer. If all fragments have not been received within a specified time frame, the packet is dropped.

The current implementation only has a single buffer for holding packets to be reassembled, and therefore does not support simultaneous reassembly of more than one packet. Since fragmented packets are uncommon, this ought to be a reasonable decision. Extending the implementation to support multiple buffers would be straightforward, however.

1.8.1.2 Broadcasts and Multicasts

IP has the ability to broadcast and multicast packets on the local network. Such packets are addressed to special broadcast and multicast addresses. Broadcast is used heavily in many UDP based protocols such as the Microsoft Windows file-sharing SMB protocol. Multicast is primarily used in protocols used for multimedia distribution such as RTP. TCP is a point-to-point protocol and does not use broadcast or multicast packets. uIP current supports broadcast packets as well as sending multicast packets. Joining multicast groups (IGMP) and receiving non-local multicast packets is not currently supported.

1.8.2 ICMP — Internet Control Message Protocol

The ICMP protocol is used for reporting soft error conditions and for querying host parameters. Its main use is, however, the echo mechanism which is used by the "ping" program.

The ICMP implementation in uIP is very simple as it is restricted to only implement ICMP echo messages. Replies to echo messages are constructed by simply swapping the source and destination IP addresses of incoming echo requests and rewriting the ICMP header with the Echo-Reply message type. The ICMP checksum is adjusted using standard techniques (see RFC1624).

Since only the ICMP echo message is implemented, there is no support for Path MTU discovery or ICMP redirect messages. Neither of these is strictly required for interoperability; they are performance enhancement mechanisms.

1.8.3 TCP — Transmission Control Protocol

The TCP implementation in uIP is driven by incoming packets and timer events. Incoming packets are parsed by TCP and if the packet contains data that is to be delivered to the application, the application is invoked by the means of the application function call. If the incoming packet acknowledges previously sent data, the connection state is updated and the application is informed, allowing it to send out new data.

1.8.3.1 Listening Connections

TCP allows a connection to listen for incoming connection requests. In uIP, a listening connection is identified by the 16-bit port number and incoming connection requests are checked against the list of listening connections. This list of listening connections is dynamic and can be altered by the applications in the system.

1.8.3.2 Sliding Window

Most TCP implementations use a sliding window mechanism for sending data. Multiple data segments are sent in succession without waiting for an acknowledgment for each segment.

The sliding window algorithm uses a lot of 32-bit operations and because 32-bit arithmetic is fairly expensive on most 8-bit CPUs, uIP does not implement it. Also, uIP does not buffer sent packets and a sliding window implementation that does not buffer sent packets will have to be supported by a complex application layer. Instead, uIP allows only a single TCP segment per connection to be unacknowledged at any given time.

It is important to note that even though most TCP implementations use the sliding window algorithm, it is not required by the TCP specifications. Removing the sliding window mechanism does not affect interoperability in any way.

1.8.3.3 Round-Trip Time Estimation

TCP continuously estimates the current Round-Trip Time (RTT) of every active connection in order to find a suitable value for the retransmission time-out.

The RTT estimation in uIP is implemented using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network. When an acknowledgment is received, the current value of the counter is used as a sample of the RTT. The sample is used together with Van Jacobson's standard TCP RTT estimation function to calculate an estimate of the RTT. Karn's algorithm is used to ensure that retransmissions do not skew the estimates.

1.8.3.4 Retransmissions

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, 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.

1.8.3.5 Flow Control

The purpose of TCP's flow control mechanisms is to allow communication between hosts with wildly varying memory dimensions. In each TCP segment, the sender of the segment indicates its available buffer space. A TCP sender must not send more data than the buffer space indicated by the receiver.

In uIP, the application cannot send more data than the receiving host can buffer. And application cannot send more data than the amount of bytes it is allowed to send by the receiving host. If the remote host cannot accept any data at all, the stack initiates the zero window probing mechanism.

1.8.3.6 Congestion Control

The congestion control mechanisms limit the number of simultaneous TCP segments in the network. The algorithms used for congestion control are designed to be simple to implement and require only a few lines of code.

Since uIP only handles one in-flight TCP segment per connection, the amount of simultaneous segments cannot be further limited, thus the congestion control mechanisms are not needed.

1.8.3.7 Urgent Data

TCP's urgent data mechanism provides an application-to-application notification mechanism, which can be used by an application to mark parts of the data stream as being more urgent than the normal stream. It is up to the receiving application to interpret the meaning of the urgent data.

In many TCP implementations, including the BSD implementation, the urgent data feature increases the complexity of the implementation because it requires an asynchronous notification mechanism in an otherwise synchronous API. As uIP already use an asynchronous event based API, the implementation of the urgent data feature does not lead to increased complexity.

1.9 Performance

In TCP/IP implementations for high-end systems, processing time is dominated by the checksum calculation loop, the operation of copying packet data and context switching. Operating systems for high-end systems often have multiple protection domains for protecting kernel data from user processes and user processes from each other. Because the TCP/IP stack is run in the kernel, data has to be copied between the kernel space and the address space of the user processes and a context switch has to be performed once the data has been copied. Performance can be enhanced by combining the copy operation with the checksum calculation. Because high-end systems usually have numerous active connections, packet demultiplexing is also an expensive operation.

A small embedded device does not have the necessary processing power to have multiple protection domains and the power to run a multitasking operating system. Therefore there is no need to copy data between the TCP/IP stack and the application program. With an event based API there is no context switch between the TCP/IP stack and the applications.

In such limited systems, the TCP/IP processing overhead is dominated by the copying of packet data from the network device to host memory, and checksum calculation. Apart from the checksum calculation

and copying, the TCP processing done for an incoming packet involves only updating a few counters and flags before handing the data over to the application. Thus an estimate of the CPU overhead of our TCP/IP implementations can be obtained by calculating the amount of CPU cycles needed for the checksum calculation and copying of a maximum sized packet.

1.9.1 The Impact of Delayed Acknowledgments

Most TCP receivers implement the delayed acknowledgment algorithm for reducing the number of pure acknowledgment packets sent. A TCP receiver using this algorithm will only send acknowledgments for every other received segment. If no segment is received within a specific time-frame, an acknowledgment is sent. The time-frame can be as high as 500 ms but typically is 200 ms.

A TCP sender such as uIP that only handles a single outstanding TCP segment will interact poorly with the delayed acknowledgment algorithm. Because the receiver only receives a single segment at a time, it will wait as much as 500 ms before an acknowledgment is sent. This means that the maximum possible throughput is severely limited by the 500 ms idle time.

Thus the maximum throughput equation when sending data from uIP will be $p = s / (t + t_d)$ where s is the segment size and t_d is the delayed acknowledgment timeout, which typically is between 200 and 500 ms. With a segment size of 1000 bytes, a round-trip time of 40 ms and a delayed acknowledgment timeout of 200 ms, the maximum throughput will be 4166 bytes per second. With the delayed acknowledgment algorithm disabled at the receiver, the maximum throughput would be 25000 bytes per second.

It should be noted, however, that since small systems running uIP are not very likely to have large amounts of data to send, the delayed acknowledgment throughput degradation of uIP need not be very severe. Small amounts of data sent by such a system will not span more than a single TCP segment, and would therefore not be affected by the throughput degradation anyway.

The maximum throughput when uIP acts as a receiver is not affected by the delayed acknowledgment throughput degradation.

Note:

The [uIP TCP throughput booster hack](#) module implements a hack that overcomes the problems with the delayed acknowledgment throughput degradation.

Chapter 2

uIP 1.0 Module Index

2.1 uIP 1.0 Modules

Here is a list of all modules:

Protothreads	27
Local continuations	90
Applications	36
DNS resolver	105
SMTP E-mail sender	108
Telnet server	110
Hello, world	113
Web client	114
Web server	120
The uIP TCP/IP stack	62
uIP configuration functions	37
uIP initialization functions	40
uIP device driver functions	41
uIP application functions	46
uIP conversion functions	55
Variables used in uIP device drivers	61
uIP Address Resolution Protocol	76
uIP TCP throughput booster hack	89
Architecture specific uIP functions	74
Configuration options for uIP	79
Timer library	91
Clock interface	94
Protosockets library	95
Memory block management functions	102

Chapter 3

uIP 1.0 Hierarchical Index

3.1 uIP 1.0 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

dhcpc_state	123
hello_world_state	124
httpd_cgi_call	125
httpd_state	126
memb_blocks	127
psock	128
psock_buf	129
pt	130
smtp_state	131
telnetd_state	132
timer	133
uip_conn	134
uip_eth_addr	136
uip_eth_hdr	137
uip_icmpip_hdr	138
uip_neighbor_addr	139
uip_stats	140
uip_tcpip_hdr	142
uip_udp_conn	143
uip_udpip_hdr	144
webclient_state	145

Chapter 4

uIP 1.0 Data Structure Index

4.1 uIP 1.0 Data Structures

Here are the data structures with brief descriptions:

dhcpc_state	123
hello_world_state	124
httpd_cgi_call	125
httpd_state	126
memb_blocks	127
psock (The representation of a protosocket)	128
psock_buf	129
pt	130
smtp_state	131
telnetd_state	132
timer (A timer)	133
uip_conn (Representation of a uIP TCP connection)	134
uip_eth_addr (Representation of a 48-bit Ethernet address)	136
uip_eth_hdr (The Ethernet header)	137
uip_icmpip_hdr	138
uip_neighbor_addr	139
uip_stats (The structure holding the TCP/IP statistics that are gathered if UIP_STATISTICS is set to 1)	140
uip_tcpip_hdr	142
uip_udp_conn (Representation of a uIP UDP connection)	143
uip_udpip_hdr	144
webclient_state	145

Chapter 5

uIP 1.0 File Index

5.1 uIP 1.0 File List

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

apps/dhcpc/dhcpc.c	??
apps/dhcpc/dhcpc.h	??
apps/hello-world/hello-world.c (An example of how to write uIP applications with protosockets)	147
apps/hello-world/hello-world.h (Header file for an example of how to write uIP applications with protosockets)	148
apps/resolv/resolv.c (DNS host name to IP address resolver)	149
apps/resolv/resolv.h (DNS resolver code header file)	151
apps/smtp/smtp.c (SMTP example implementation)	153
apps/smtp/smtp.h (SMTP header file)	154
apps/telnetd/shell.c (Simple shell)	156
apps/telnetd/shell.h (Simple shell, header file)	157
apps/telnetd/telnetd.c (Shell server)	158
apps/telnetd/telnetd.h (Shell server)	159
apps/webclient/webclient.c (Implementation of the HTTP client)	160
apps/webclient/webclient.h (Header file for the HTTP client)	162
apps/webserver/httpd-cgi.c (Web server script interface)	164
apps/webserver/httpd-cgi.h (Web server script interface header file)	165
apps/webserver/httpd.c (Web server)	166
apps/webserver/httpd.h	??
lib/memb.c (Memory block allocation routines)	167
lib/memb.h (Memory block allocation routines)	168
uip/clock.h	??
uip/lc-addrlabels.h (Implementation of local continuations based on the "Labels as values" feature of gcc)	169
uip/lc-switch.h (Implementation of local continuations based on switch() statement)	170
uip/lc.h (Local continuations)	171
uip/psock.c	??
uip/psock.h (Protosocket library header file)	172
uip/pt.h (Protothreads implementation)	174
uip/timer.c (Timer library implementation)	176
uip/timer.h (Timer library header file)	177
uip/uip-neighbor.c (Database of link-local neighbors, used by IPv6 code and to be used by a future ARP code rewrite)	178

uip/uip-neighbor.h (Header file for database of link-local neighbors, used by IPv6 code and to be used by future ARP code)	179
uip/uip-split.c	??
uip/uip-split.h (Module for splitting outbound TCP segments in two to avoid the delayed ACK throughput degradation)	180
uip/uip.c (The uIP TCP/IP stack code)	181
uip/uip.h (Header file for the uIP TCP/IP stack)	184
uip/uip_arch.h (Declarations of architecture specific functions)	190
uip/uip_arp.c (Implementation of the ARP Address Resolution Protocol)	191
uip/uip_arp.h (Macros and definitions for the ARP module)	192
uip/uipopt.h (Configuration options for uIP)	193
unix/uip-conf.h (An example uIP configuration file)	196

Chapter 6

uIP 1.0 Module Documentation

6.1 Protothreads

6.1.1 Detailed Description

Protothreads are a type of lightweight stackless threads designed for severely memory constrained systems such as deeply embedded systems or sensor network nodes.

Protothreads provides linear code execution for event-driven systems implemented in C. Protothreads can be used with or without an RTOS.

Protothreads are an extremely lightweight, stackless type of threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading. Protothreads provides conditional blocking inside C functions.

The advantage of protothreads over a purely event-driven approach is that protothreads provides a sequential code structure that allows for blocking functions. In purely event-driven systems, blocking must be implemented by manually breaking the function into two pieces - one for the piece of code before the blocking call and one for the code after the blocking call. This makes it hard to use control structures such as `if()` conditionals and `while()` loops.

The advantage of protothreads over ordinary threads is that a protothread does not require a separate stack. In memory constrained systems, the overhead of allocating multiple stacks can consume large amounts of the available memory. In contrast, each protothread only requires between two and twelve bytes of state, depending on the architecture.

Note:

Because protothreads do not save the stack context across a blocking call, **local variables are not preserved when the protothread blocks**. This means that local variables should be used with utmost care - **if in doubt, do not use local variables inside a protothread!**

Main features:

- No machine specific code - the protothreads library is pure C
- Does not use error-prone functions such as `longjmp()`
- Very small RAM overhead - only two bytes per protothread

- Can be used with or without an OS
- Provides blocking wait without full multi-threading or stack-switching

Examples applications:

- Memory constrained systems
- Event-driven protocol stacks
- Deeply embedded systems
- Sensor network nodes

The protothreads API consists of four basic operations: initialization: [PT_INIT\(\)](#), execution: [PT_BEGIN\(\)](#), conditional blocking: [PT_WAIT_UNTIL\(\)](#) and exit: [PT_END\(\)](#). On top of these, two convenience functions are built: reversed condition blocking: [PT_WAIT_WHILE\(\)](#) and protothread blocking: [PT_WAIT_THREAD\(\)](#).

See also:

[Protothreads API documentation](#)

The protothreads library is released under a BSD-style license that allows for both non-commercial and commercial usage. The only requirement is that credit is given.

6.1.2 Authors

The protothreads library was written by Adam Dunkels <adam@sics.se> with support from Oliver Schmidt <ol.sc@web.de>.

6.1.3 Protothreads

Protothreads are a extremely lightweight, stackless threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without using complex state machines or full multi-threading. Protothreads provides conditional blocking inside a C function.

In memory constrained systems, such as deeply embedded systems, traditional multi-threading may have a too large memory overhead. In traditional multi-threading, each thread requires its own stack, that typically is over-provisioned. The stacks may use large parts of the available memory.

The main advantage of protothreads over ordinary threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. A protothread only requires only two bytes of memory per protothread. Moreover, protothreads are implemented in pure C and do not require any machine-specific assembler code.

A protothread runs within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead made by spawning a separate protothread for each potentially blocking function. The advantage of

this approach is that blocking is explicit: the programmer knows exactly which functions that block that which functions the never blocks.

Protothreads are similar to asymmetric co-routines. The main difference is that co-routines uses a separate stack for each co-routine, whereas protothreads are stackless. The most similar mechanism to protothreads are Python generators. These are also stackless constructs, but have a different purpose. Protothreads provides blocking contexts inside a C function, whereas Python generators provide multiple exit points from a generator function.

6.1.4 Local variables

Note:

Because protothreads do not save the stack context across a blocking call, local variables are not preserved when the protothread blocks. This means that local variables should be used with utmost care - if in doubt, do not use local variables inside a protothread!

6.1.5 Scheduling

A protothread is driven by repeated calls to the function in which the protothread is running. Each time the function is called, the protothread will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

6.1.6 Implementation

Protothreads are implemented using [local continuations](#). A local continuation represents the current state of execution at a particular place in the program, but does not provide any call history or local variables. A local continuation can be set in a specific function to capture the state of the function. After a local continuation has been set can be resumed in order to restore the state of the function at the point where the local continuation was set.

Local continuations can be implemented in a variety of ways:

1. by using machine specific assembler code,
2. by using standard C constructs, or
3. by using compiler extensions.

The first way works by saving and restoring the processor state, except for stack pointers, and requires between 16 and 32 bytes of memory per protothread. The exact amount of memory required depends on the architecture.

The standard C implementation requires only two bytes of state per protothread and utilizes the C `switch()` statement in a non-obvious way that is similar to Duff's device. This implementation does, however, impose a slight restriction to the code that uses protothreads in that the code cannot use `switch()` statements itself.

Certain compilers has C extensions that can be used to implement protothreads. GCC supports label pointers that can be used for this purpose. With this implementation, protothreads require 4 bytes of RAM per protothread.

Files

- file [pt.h](#)

Protothreads implementation.

Modules

- [Local continuations](#)

Local continuations form the basis for implementing protothreads.

Data Structures

- struct [pt](#)

Initialization

- #define [PT_INIT\(pt\)](#)

Initialize a protothread.

Declaration and definition

- #define [PT_THREAD\(name_args\)](#)

Declaration of a protothread.

- #define [PT_BEGIN\(pt\)](#)

Declare the start of a protothread inside the C function implementing the protothread.

- #define [PT_END\(pt\)](#)

Declare the end of a protothread.

Blocked wait

- #define [PT_WAIT_UNTIL\(pt, condition\)](#)

Block and wait until condition is true.

- #define [PT_WAIT_WHILE\(pt, cond\)](#)

Block and wait while condition is true.

Hierarchical protothreads

- #define [PT_WAIT_THREAD\(pt, thread\)](#)

Block and wait until a child protothread completes.

- #define [PT_SPAWN\(pt, child, thread\)](#)

Spawn a child protothread and wait until it exits.

Exiting and restarting

- #define [PT_RESTART\(pt\)](#)
Restart the protothread.
- #define [PT_EXIT\(pt\)](#)
Exit the protothread.

Calling a protothread

- #define [PT_SCHEDULE\(f\)](#)
Schedule a protothread.

Yielding from a protothread

- #define [PT_YIELD\(pt\)](#)
Yield from the current protothread.
- #define [PT_YIELD_UNTIL\(pt, cond\)](#)
Yield from the protothread until a condition occurs.

Defines

- #define [PT_WAITING](#) 0
- #define [PT_EXITED](#) 1
- #define [PT_ENDED](#) 2
- #define [PT_YELDED](#) 3

6.1.7 Define Documentation

6.1.7.1 #define [PT_BEGIN\(pt\)](#)

Declare the start of a protothread inside the C function implementing the protothread.

This macro is used to declare the starting point of a protothread. It should be placed at the start of the function in which the protothread runs. All C statements above the [PT_BEGIN\(\)](#) invocation will be executed each time the protothread is scheduled.

Parameters:

pt A pointer to the protothread control structure.

Examples:

[dhcpc.c](#).

Definition at line 115 of file pt.h.

6.1.7.2 #define PT_END([pt](#))

Declare the end of a protothread.

This macro is used for declaring that a protothread ends. It must always be used together with a matching [PT_BEGIN\(\)](#) macro.

Parameters:

pt A pointer to the protothread control structure.

Examples:

[dhcpc.c](#).

Definition at line 127 of file [pt.h](#).

6.1.7.3 #define PT_EXIT([pt](#))

Exit the protothread.

This macro causes the protothread to exit. If the protothread was spawned by another protothread, the parent protothread will become unblocked and can continue to run.

Parameters:

pt A pointer to the protothread control structure.

Definition at line 246 of file [pt.h](#).

6.1.7.4 #define PT_INIT([pt](#))

Initialize a protothread.

Initializes a protothread. Initialization must be done prior to starting to execute the protothread.

Parameters:

pt A pointer to the protothread control structure.

See also:

[PT_SPAWN\(\)](#)

Examples:

[dhcpc.c](#).

Definition at line 80 of file [pt.h](#).

Referenced by [httpd_appcall\(\)](#).

6.1.7.5 #define PT_RESTART([pt](#))

Restart the protothread.

This macro will block and cause the running protothread to restart its execution at the place of the [PT_BEGIN\(\)](#) call.

Parameters:

pt A pointer to the protothread control structure.

Examples:

[dhcpc.c](#).

Definition at line 229 of file pt.h.

6.1.7.6 #define PT_SCHEDULE(f)

Schedule a protothread.

This function schedules a protothread. The return value of the function is non-zero if the protothread is running or zero if the protothread has exited.

Parameters:

f The call to the C function implementing the protothread to be scheduled

Definition at line 271 of file pt.h.

6.1.7.7 #define PT_SPAWN(*pt*, *child*, *thread*)

Spawn a child protothread and wait until it exits.

This macro spawns a child protothread and waits until it exits. The macro can only be used within a protothread.

Parameters:

pt A pointer to the protothread control structure.

child A pointer to the child protothread's control structure.

thread The child protothread with arguments

Definition at line 206 of file pt.h.

6.1.7.8 #define PT_THREAD(name_args)

Declaration of a protothread.

This macro is used to declare a protothread. All protothreads must be declared with this macro.

Parameters:

name_args The name and arguments of the C function implementing the protothread.

Examples:

[dhcpc.c](#), and [smtp.c](#).

Definition at line 100 of file pt.h.

6.1.7.9 #define PT_WAIT_THREAD(*pt*, *thread*)

Block and wait until a child protothread completes.

This macro schedules a child protothread. The current protothread will block until the child protothread completes.

Note:

The child protothread must be manually initialized with the [PT_INIT\(\)](#) function before this function is used.

Parameters:

pt A pointer to the protothread control structure.
thread The child protothread with arguments

See also:

[PT_SPAWN\(\)](#)

Definition at line 192 of file pt.h.

6.1.7.10 #define PT_WAIT_UNTIL(*pt*, *condition*)

Block and wait until condition is true.

This macro blocks the protothread until the specified condition is true.

Parameters:

pt A pointer to the protothread control structure.
condition The condition.

Examples:

[dhcpc.c](#).

Definition at line 148 of file pt.h.

6.1.7.11 #define PT_WAIT_WHILE(*pt*, *cond*)

Block and wait while condition is true.

This function blocks and waits while condition is true. See [PT_WAIT_UNTIL\(\)](#).

Parameters:

pt A pointer to the protothread control structure.
cond The condition.

Definition at line 167 of file pt.h.

6.1.7.12 #define PT_YIELD(*pt*)

Yield from the current protothread.

This function will yield the protothread, thereby allowing other processing to take place in the system.

Parameters:

pt A pointer to the protothread control structure.

Examples:

[dhcpc.c](#).

Definition at line 290 of file pt.h.

6.1.7.13 #define PT_YIELD_UNTIL([pt](#), cond)

Yield from the protothread until a condition occurs.

Parameters:

pt A pointer to the protothread control structure.

cond The condition.

This function will yield the protothread, until the specified condition evaluates to true.

Definition at line 310 of file pt.h.

6.2 Applications

6.2.1 Detailed Description

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

Modules

- [DNS resolver](#)

The uIP DNS resolver functions are used to lookup a hostname and map it to a numerical IP address.

- [SMTP E-mail sender](#)

The Simple Mail Transfer Protocol (SMTP) as defined by RFC821 is the standard way of sending and transferring e-mail on the Internet.

- [Telnet server](#)

The uIP telnet server.

- [Hello, world](#)

A small example showing how to write applications with [protosockets](#).

- [Web client](#)

This example shows a HTTP client that is able to download web pages and files from web servers.

- [Web server](#)

The uIP web server is a very simplistic implementation of an HTTP server.

Variables

- char [telnetd_state::buf](#) [TELNETD_CONF_LINELEN]
- char [telnetd_state::bufptr](#)
- [u8_t telnetd_state::numsent](#)
- [u8_t telnetd_state::state](#)

6.3 uIP configuration functions

6.3.1 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)`
Specify the Ethernet MAC address.

6.3.2 Define Documentation

6.3.2.1 `#define uip_getdraddr(addr)`

Get the default router's IP address.

Parameters:

addr A pointer to a `uip_ipaddr_t` variable that will be filled in with the IP address of the default router.

Definition at line 161 of file `uip.h`.

6.3.2.2 `#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.

Example:

```
uip_ipaddr_t hostaddr;  
  
uip_gethostaddr(&hostaddr);
```

Parameters:

addr A pointer to a `uip_ipaddr_t` variable that will be filled in with the currently configured IP address.

Definition at line 126 of file `uip.h`.

6.3.2.3 #define uip_getnetmask(addr)

Get the netmask.

Parameters:

addr A pointer to a `uip_ipaddr_t` variable that will be filled in with the value of the netmask.

Definition at line 171 of file `uip.h`.

6.3.2.4 #define uip_setdraddr(addr)

Set the default router's IP address.

Parameters:

addr A pointer to a `uip_ipaddr_t` variable containing the IP address of the default router.

See also:

[uip_ipaddr\(\)](#)

Definition at line 138 of file `uip.h`.

6.3.2.5 #define uip_setethaddr(eaddr)

Specify 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.

Definition at line 134 of file `uip_arp.h`.

6.3.2.6 #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.

Example:

```
uip_ipaddr_t addr;  
  
uip_ipaddr(&addr, 192, 168, 1, 2);  
uip_sethostaddr(&addr);
```

Parameters:

addr A pointer to an IP address of type uip_ipaddr_t;

See also:

[uip_ipaddr\(\)](#)

Examples:

[dhcpc.c](#), [example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 106 of file uip.h.

6.3.2.7 #define uip_setnetmask(addr)

Set the netmask.

Parameters:

addr A pointer to a uip_ipaddr_t variable containing the IP address of the netmask.

See also:

[uip_ipaddr\(\)](#)

Definition at line 150 of file uip.h.

6.4 uIP initialization functions

6.4.1 Detailed Description

The uIP initialization functions are used for booting uIP.

Functions

- void [uip_init](#) (void)
uIP initialization function.
- void [uip_setipid](#) ([u16_t](#) id)
uIP initialization function.

6.4.2 Function Documentation

6.4.2.1 void [uip_init](#) (void)

uIP initialization function.

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

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 379 of file [uip.c](#).

References [UIP_LISTENPORTS](#).

6.4.2.2 void [uip_setipid](#) ([u16_t](#) id)

uIP initialization function.

This function may be used at boot time to set the initial `ip_id`.

Definition at line 181 of file [uip.c](#).

6.5 uIP device driver functions

6.5.1 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_conn_active(conn) (uip_conns[conn].tcpstateflags != UIP_CLOSED)`
- `#define uip_periodic_conn(conn)`
Perform periodic processing for a connection identified by a pointer to its structure.
- `#define uip_poll_conn(conn)`
Reuqest that a particular connection should be polled.
- `#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.

6.5.2 Define Documentation

6.5.2.1 `#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_devicedriver_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();
        }
    }
}

```

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 257 of file uip.h.

6.5.2.2 #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 or 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 usual 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.

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 301 of file uip.h.

6.5.2.3 #define uip_periodic_conn(conn)

Perform 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.

Definition at line 323 of file uip.h.

6.5.2.4 #define uip_poll_conn(conn)

Reuquest that a particular connection should be polled.

Similar to [uip_periodic_conn\(\)](#) but does not perform any timer processing. The application is polled for new data.

Parameters:

conn A pointer to the [uip_conn](#) struct for the connection to be processed.

Definition at line 337 of file uip.h.

6.5.2.5 #define uip_udp_periodic(conn)

Periodic processing for a UDP connection identified by its number.

This function is essentially the same as [uip_periodic\(\)](#), 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.

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 373 of file uip.h.

6.5.2.6 #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.

Definition at line 390 of file uip.h.

6.5.3 Variable Documentation**6.5.3.1 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);
    if(uip_len <= UIP_LLH_LEN + UIP_TCPIP_HLEN) {
        hwsend(&uip_buf[UIP_LLH_LEN], uip_len - UIP_LLH_LEN);
    } else {
        hwsend(&uip_buf[UIP_LLH_LEN], UIP_TCPIP_HLEN);
        hwsend(uip_appdata, uip_len - UIP_TCPIP_HLEN - UIP_LLH_LEN);
    }
}
```

Definition at line 139 of file uip.c.

Referenced by uip_process().

6.6 uIP application functions

6.6.1 Detailed Description

Functions used by an application running on top of uIP.

Defines

- #define `uip_outstanding(conn)` $((conn) \rightarrow len)$
- #define `uip_datalen()`
The length of any incoming data that is currently available (if available) 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 it has previously been stopped with `uip_stop()`.
- #define `uip_udpconnection()`
Is the current connection a UDP connection?
- #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_bind(conn, port)`

Bind a UDP connection to a local port.

- #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 (uip_ipaddr_t *ripaddr, u16_t port)`

Connect to a remote host using TCP.

- void `uip_send (const void *data, int len)`

Send data on the current connection.

- `uip_udp_conn * uip_udp_new (uip_ipaddr_t *ripaddr, u16_t rport)`

Set up a new UDP connection.

6.6.2 Define Documentation

6.6.2.1 #define `uip_abort()`

Abort the current connection.

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

Examples:

[webclient.c](#).

Definition at line 581 of file uip.h.

Referenced by `httpd_appcall()`, and `webclient_appcall()`.

6.6.2.2 #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.

Examples:

[smtp.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 680 of file uip.h.

Referenced by `httpd_appcall()`, `smtp_appcall()`, and `webclient_appcall()`.

6.6.2.3 #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.

Examples:

[telnetd.c](#), and [webclient.c](#).

Definition at line 648 of file uip.h.

Referenced by `webclient_appcall()`.

6.6.2.4 #define uip_close()

Close the current connection.

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

Examples:

[telnetd.c](#).

Definition at line 570 of file uip.h.

6.6.2.5 #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.

Examples:

[smtp.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 670 of file uip.h.

Referenced by `httpd_appcall()`, `smtp_appcall()`, and `webclient_appcall()`.

6.6.2.6 #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\(\)](#)).

Examples:

[hello-world.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 660 of file uip.h.

Referenced by [hello_world_appcall\(\)](#), [httpd_appcall\(\)](#), [telnetd_appcall\(\)](#), and [webclient_appcall\(\)](#).

6.6.2.7 #define uip_datalen()

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

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

Examples:

[dhcpc.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 550 of file uip.h.

6.6.2.8 #define uip_mss()

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

The current maximum 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\(\)](#)).

Examples:

[telnetd.c](#), and [webclient.c](#).

Definition at line 737 of file uip.h.

6.6.2.9 #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 available through the uip_len variable.

Examples:

[dhcpc.c](#), [resolv.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 637 of file uip.h.

Referenced by [psock_newdata\(\)](#), [resolv_appcall\(\)](#), and [webclient_appcall\(\)](#).

6.6.2.10 #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.

Examples:

[resolv.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 716 of file uip.h.

Referenced by `httpd_appcall()`, `resolv_appcall()`, and `webclient_appcall()`.

6.6.2.11 `#define uip_restart()`

Restart the current connection, if it 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.

Definition at line 610 of file uip.h.

6.6.2.12 `#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.

Examples:

[telnetd.c](#), and [webclient.c](#).

Definition at line 702 of file uip.h.

Referenced by `webclient_appcall()`.

6.6.2.13 `#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.

Definition at line 591 of file uip.h.

6.6.2.14 `#define uip_timeout()`

Has the connection timed out?

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

Examples:

[smtp.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 690 of file uip.h.

Referenced by `httpd_appcall()`, `smtp_appcall()`, and `webclient_appcall()`.

6.6.2.15 #define uip_udp_bind(conn, port)

Bind a UDP connection to a local port.

Parameters:

conn A pointer to the [uip_udp_conn](#) structure for the connection.

port The local port number, in network byte order.

Examples:

[dhcpc.c](#).

Definition at line 787 of file uip.h.

6.6.2.16 #define uip_udp_remove(conn)

Removed a UDP connection.

Parameters:

conn A pointer to the [uip_udp_conn](#) structure for the connection.

Examples:

[resolv.c](#).

Definition at line 775 of file uip.h.

Referenced by [resolv_conf\(\)](#).

6.6.2.17 #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.

Examples:

[resolv.c](#).

Definition at line 800 of file uip.h.

6.6.2.18 #define uip_udpconnection()

Is the current connection a UDP connection?

This function checks whether the current connection is a UDP connection.

Definition at line 626 of file uip.h.

6.6.2.19 #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.

Definition at line 561 of file `uip.h`.

6.6.3 Function Documentation

6.6.3.1 struct `uip_conn*` `uip_connect (uip_ipaddr_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 specified 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 available 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 conversion using `HTONS()` or `htons()` is necessary.

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

Parameters:

ripaddr The IP address of the remote host.

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.

Examples:

`smtp.c`, and `webclient.c`.

Definition at line 407 of file `uip.c`.

References `htons()`, `uip_conn::lport`, `uip_conn::tcpstateflags`, `UIP_CLOSED`, `uip_conn`, `UIP_CONNS`, and `uip_conns`.

Referenced by `smtp_send()`, and `webclient_get()`.

6.6.3.2 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.

Examples:

[hello-world.c](#), and [telnetd.c](#).

Definition at line 529 of file uip.c.

References UIP_LISTENPORTS.

Referenced by [hello_world_init\(\)](#), [httpd_init\(\)](#), and [telnetd_init\(\)](#).

6.6.3.3 void uip_send (const void * data, int 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 function 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.

Examples:

[dhcpc.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 1888 of file uip.c.

References [uip_sappdata](#), and [uip_slen](#).

6.6.3.4 struct uip_udp_conn* uip_udp_new (uip_ipaddr_t * ripaddr, u16_t rport)

Set up a new UDP connection.

This function sets up a new UDP connection. The function will automatically allocate an unused local port for the new connection. However, another port can be chosen by using the [uip_udp_bind\(\)](#) call, after the [uip_udp_new\(\)](#) function has been called.

Example:

```
uip_ipaddr_t addr;
struct uip_udp_conn *c;

uip_ipaddr(&addr, 192,168,2,1);
c = uip_udp_new(&addr, HTONS(12345));
if(c != NULL) {
    uip_udp_bind(c, HTONS(12344));
}
```

Parameters:

ripaddr 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.

Examples:

[dhcpc.c](#), and [resolv.c](#).

Definition at line 473 of file uip.c.

References [htons\(\)](#), [uip_udp_conn::lport](#), [uip_udp_conn](#), [UIP_UDP_CONNS](#), and [uip_udp_conns](#).

Referenced by [resolv_conf\(\)](#).

6.6.3.5 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.

Definition at line 518 of file uip.c.

References [UIP_LISTENPORTS](#).

6.7 uIP conversion functions

6.7.1 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)
Construct an IP address from four bytes.
- #define `uip_ip6addr`(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7)
Construct an IPv6 address from eight 16-bit words.
- #define `uip_ipaddr_copy`(dest, src)
Copy an IP address to another IP address.
- #define `uip_ipaddr_cmp`(addr1, addr2)
Compare two IP addresses.
- #define `uip_ipaddr_maskcmp`(addr1, addr2, mask)
Compare two IP addresses with netmasks.
- #define `uip_ipaddr_mask`(dest, src, mask)
Mask out the network part of an IP address.
- #define `uip_ipaddr1`(addr)
Pick the first octet of an IP address.
- #define `uip_ipaddr2`(addr)
Pick the second octet of an IP address.
- #define `uip_ipaddr3`(addr)
Pick the third octet of an IP address.
- #define `uip_ipaddr4`(addr)
Pick the fourth octet of an IP address.
- #define `HTONS`(n)
Convert 16-bit quantity from host byte order to network byte order.
- #define `ntohs` htons

Functions

- `u16_t htons` (`u16_t` val)
Convert 16-bit quantity from host byte order to network byte order.

6.7.2 Define Documentation

6.7.2.1 #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.

Examples:

[dhcpc.c](#), [hello-world.c](#), [resolv.c](#), [smtp.c](#), and [telnetd.c](#).

Definition at line 1070 of file uip.h.

Referenced by [hello_world_init\(\)](#), [htons\(\)](#), [httpd_init\(\)](#), [resolv_appcall\(\)](#), [resolv_conf\(\)](#), [smtp_send\(\)](#), [telnetd_init\(\)](#), [uip_arp_arpin\(\)](#), and [uip_process\(\)](#).

6.7.2.2 #define uip_ip6addr(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7)

Construct an IPv6 address from eight 16-bit words.

This function constructs an IPv6 address.

Definition at line 852 of file uip.h.

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

Construct an IP address from four bytes.

This function constructs an IP address of the type that uIP handles internally from four bytes. The function is handy for specifying IP addresses to use with e.g. the [uip_connect\(\)](#) function.

Example:

```
uip_ipaddr_t ipaddr;
struct uip_conn *c;

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

Parameters:

addr A pointer to a `uip_ipaddr_t` variable that will be filled in with the IP address.

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.

Examples:

[dhcpc.c](#), [example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 840 of file uip.h.

6.7.2.4 #define uip_ipaddr1(addr)

Pick the first octet of an IP address.

Picks out the first octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;  
u8_t octet;  
  
uip_ipaddr(&ipaddr, 1, 2, 3, 4);  
octet = uip_ipaddr1(&ipaddr);
```

In the example above, the variable "octet" will contain the value 1.

Examples:

[dhcpc.c](#).

Definition at line 995 of file uip.h.

6.7.2.5 #define uip_ipaddr2(addr)

Pick the second octet of an IP address.

Picks out the second octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;  
u8_t octet;  
  
uip_ipaddr(&ipaddr, 1, 2, 3, 4);  
octet = uip_ipaddr2(&ipaddr);
```

In the example above, the variable "octet" will contain the value 2.

Examples:

[dhcpc.c](#).

Definition at line 1015 of file uip.h.

6.7.2.6 #define uip_ipaddr3(addr)

Pick the third octet of an IP address.

Picks out the third octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;  
u8_t octet;  
  
uip_ipaddr(&ipaddr, 1, 2, 3, 4);  
octet = uip_ipaddr3(&ipaddr);
```

In the example above, the variable "octet" will contain the value 3.

Examples:[dhcpc.c](#).

Definition at line 1035 of file uip.h.

6.7.2.7 #define uip_ipaddr4(addr)

Pick the fourth octet of an IP address.

Picks out the fourth octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;  
u8_t octet;  
  
uip_ipaddr(&ipaddr, 1, 2, 3, 4);  
octet = uip_ipaddr4(&ipaddr);
```

In the example above, the variable "octet" will contain the value 4.

Examples:[dhcpc.c](#).

Definition at line 1055 of file uip.h.

6.7.2.8 #define uip_ipaddr_cmp(addr1, addr2)

Compare two IP addresses.

Compares two IP addresses.

Example:

```
uip_ipaddr_t ipaddr1, ipaddr2;  
  
uip_ipaddr(&ipaddr1, 192, 16, 1, 2);  
if(uip_ipaddr_cmp(&ipaddr2, &ipaddr1)) {  
    printf("They are the same");  
}
```

Parameters:

addr1 The first IP address.

addr2 The second IP address.

Definition at line 911 of file uip.h.

Referenced by uip_arp_arpin(), uip_arp_out(), and uip_process().

6.7.2.9 #define uip_ipaddr_copy(dest, src)

Copy an IP address to another IP address.

Copies an IP address from one place to another.

Example:


```
uip_ipaddr_t ipaddr1, ipaddr2;

uip_ipaddr(&ipaddr1, 192,16,1,2);
uip_ipaddr_copy(&ipaddr2, &ipaddr1);
```

Parameters:

dest The destination for the copy.

src The source from where to copy.

Examples:

[smtp.c](#).

Definition at line 882 of file uip.h.

Referenced by `smtp_configure()`, `uip_arp_out()`, and `uip_process()`.

6.7.2.10 #define uip_ipaddr_mask(dest, src, mask)

Mask out the network part of an IP address.

Masks out the network part of an IP address, given the address and the netmask.

Example:

```
uip_ipaddr_t ipaddr1, ipaddr2, netmask;

uip_ipaddr(&ipaddr1, 192,16,1,2);
uip_ipaddr(&netmask, 255,255,255,0);
uip_ipaddr_mask(&ipaddr2, &ipaddr1, &netmask);
```

In the example above, the variable "ipaddr2" will contain the IP address 192.168.1.0.

Parameters:

dest Where the result is to be placed.

src The IP address.

mask The netmask.

Definition at line 972 of file uip.h.

6.7.2.11 #define uip_ipaddr_maskcmp(addr1, addr2, mask)

Compare two IP addresses with netmasks.

Compares two IP addresses with netmasks. The masks are used to mask out the bits that are to be compared.

Example:

```
uip_ipaddr_t ipaddr1, ipaddr2, mask;

uip_ipaddr(&mask, 255,255,255,0);
uip_ipaddr(&ipaddr1, 192,16,1,2);
uip_ipaddr(&ipaddr2, 192,16,1,3);
if(uip_ipaddr_maskcmp(&ipaddr1, &ipaddr2, &mask)) {
    printf("They are the same");
}
```

Parameters:

- addr1* The first IP address.
addr2 The second IP address.
mask The netmask.

Definition at line 941 of file uip.h.

Referenced by uip_arp_out().

6.7.3 Function Documentation

6.7.3.1 `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.

Examples:

[example-mainloop-with-arp.c](#), [resolv.c](#), and [webclient.c](#).

Definition at line 1882 of file uip.c.

References `HTONS`.

Referenced by `uip_chksum()`, `uip_connect()`, `uip_ipchksum()`, `uip_udp_new()`, and `webclient_get()`.

6.8 Variables used in uIP device drivers

6.8.1 Detailed Description

uIP has a few global variables that are used in device drivers for uIP.

Variables

- [u16_t uip_len](#)

The length of the packet in the uip_buf buffer.

6.8.2 Variable Documentation

6.8.2.1 [u16_t uip_len](#)

The length of the packet in the uip_buf buffer.

The global variable uip_len holds the length of the packet in the uip_buf buffer.

When the network device driver calls the uIP input function, uip_len should be set to the length of the packet in the uip_buf buffer.

When sending packets, the device driver should use the contents of the uip_len variable to determine the length of the outgoing packet.

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 155 of file uip.c.

Referenced by uip_arp_arpin(), uip_process(), and uip_split_output().

6.9 The uIP TCP/IP stack

6.9.1 Detailed Description

uIP is an implementation of the TCP/IP protocol stack intended for small 8-bit and 16-bit microcontrollers. uIP provides the necessary protocols for Internet communication, with a very small code footprint and RAM requirements - the uIP code size is on the order of a few kilobytes and RAM usage is on the order of a few hundred bytes.

Files

- file [uip.h](#)
Header file for the uIP TCP/IP stack.
- file [uip.c](#)
The uIP TCP/IP stack code.

Modules

- [uIP configuration functions](#)
The uIP configuration functions are used for setting run-time parameters in uIP such as IP addresses.
- [uIP initialization functions](#)
The uIP initialization functions are used for booting uIP.
- [uIP device driver functions](#)
These functions are used by a network device driver for interacting with uIP.
- [uIP application functions](#)
Functions used by an application running on top of uIP.
- [uIP conversion functions](#)
These functions can be used for converting between different data formats used by uIP.
- [Variables used in uIP device drivers](#)
uIP has a few global variables that are used in device drivers for uIP.
- [uIP Address Resolution Protocol](#)
The Address Resolution Protocol ARP is used for mapping between IP addresses and link level addresses such as the Ethernet MAC addresses.
- [uIP TCP throughput booster hack](#)
The basic uIP TCP implementation only allows each TCP connection to have a single TCP segment in flight at any given time.
- [Architecture specific uIP functions](#)
The functions in the architecture specific module implement the IP check sum and 32-bit additions.

Data Structures

- struct `uip_conn`
Representation of a uIP TCP connection.
- struct `uip_udp_conn`
Representation of a uIP UDP connection.
- struct `uip_stats`
The structure holding the TCP/IP statistics that are gathered if `UIP_STATISTICS` is set to 1.
- struct `uip_tcpip_hdr`
- struct `uip_icmpip_hdr`
- struct `uip_udpip_hdr`
- struct `uip_eth_addr`
Representation of a 48-bit Ethernet address.

Defines

- #define `UIP_ACKDATA` 1
- #define `UIP_NEWDATA` 2
- #define `UIP_REXMIT` 4
- #define `UIP_POLL` 8
- #define `UIP_CLOSE` 16
- #define `UIP_ABORT` 32
- #define `UIP_CONNECTED` 64
- #define `UIP_TIMEDOUT` 128
- #define `UIP_DATA` 1
- #define `UIP_TIMER` 2
- #define `UIP_POLL_REQUEST` 3
- #define `UIP_UDP_SEND_CONN` 4
- #define `UIP_UDP_TIMER` 5
- #define `UIP_CLOSED` 0
- #define `UIP_SYN_RCVD` 1
- #define `UIP_SYN_SENT` 2
- #define `UIP_ESTABLISHED` 3
- #define `UIP_FIN_WAIT_1` 4
- #define `UIP_FIN_WAIT_2` 5
- #define `UIP_CLOSING` 6
- #define `UIP_TIME_WAIT` 7
- #define `UIP_LAST_ACK` 8
- #define `UIP_TS_MASK` 15
- #define `UIP_STOPPED` 16
- #define `UIP_APPDATA_SIZE`
The buffer size available for user data in the `uip_buf` buffer.
- #define `UIP_PROTO_ICMP` 1
- #define `UIP_PROTO_TCP` 6
- #define `UIP_PROTO_UDP` 17

- #define [UIP_PROTO_ICMP6](#) 58
- #define [UIP_IPH_LEN](#) 20
- #define [UIP_UDPH_LEN](#) 8
- #define [UIP_TCPH_LEN](#) 20
- #define [UIP_IPUDPH_LEN](#) (UIP_UDPH_LEN + UIP_IPH_LEN)
- #define [UIP_IPTCPH_LEN](#) (UIP_TCPH_LEN + UIP_IPH_LEN)
- #define [UIP_TCPIP_HLEN](#) UIP_IPTCPH_LEN
- #define [TCP_FIN](#) 0x01
- #define [TCP_SYN](#) 0x02
- #define [TCP_RST](#) 0x04
- #define [TCP_PSH](#) 0x08
- #define [TCP_ACK](#) 0x10
- #define [TCP_URG](#) 0x20
- #define [TCP_CTL](#) 0x3f
- #define [TCP_OPT_END](#) 0
- #define [TCP_OPT_NOOP](#) 1
- #define [TCP_OPT_MSS](#) 2
- #define [TCP_OPT_MSS_LEN](#) 4
- #define [ICMP_ECHO_REPLY](#) 0
- #define [ICMP_ECHO](#) 8
- #define [ICMP6_ECHO_REPLY](#) 129
- #define [ICMP6_ECHO](#) 128
- #define [ICMP6_NEIGHBOR_SOLICITATION](#) 135
- #define [ICMP6_NEIGHBOR_ADVERTISEMENT](#) 136
- #define [ICMP6_FLAG_S](#) (1 << 6)
- #define [ICMP6_OPTION_SOURCE_LINK_ADDRESS](#) 1
- #define [ICMP6_OPTION_TARGET_LINK_ADDRESS](#) 2
- #define [BUF](#) ((struct [uip_tcpip_hdr](#) *)&[uip_buf](#)[UIP_LLH_LEN])
- #define [FBUF](#) ((struct [uip_tcpip_hdr](#) *)&[uip_reassbuf](#)[0])
- #define [ICMPBUF](#) ((struct [uip_icmpip_hdr](#) *)&[uip_buf](#)[UIP_LLH_LEN])
- #define [UDPBUF](#) ((struct [uip_udpip_hdr](#) *)&[uip_buf](#)[UIP_LLH_LEN])
- #define [UIP_STAT](#)(s)
- #define [UIP_LOG](#)(m)

Typedefs

- typedef [u16_t](#) [uip_ip4addr_t](#) [2]
Representation of an IP address.
- typedef [u16_t](#) [uip_ip6addr_t](#) [8]
- typedef [uip_ip4addr_t](#) [uip_ipaddr_t](#)

Functions

- void [uip_process](#) ([u8_t](#) flag)
- [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.
- `u16_t uip_udpchksum` (void)
Calculate the UDP checksum of the packet in uip_buf and uip_appdata.
- void `uip_setipid` (u16_t id)
uIP initialization function.
- void `uip_add32` (u8_t *op32, u16_t op16)
Carry out a 32-bit addition.
- void `uip_init` (void)
uIP initialization function.
- `uip_conn * uip_connect` (uip_ipaddr_t *ripaddr, u16_t rport)
Connect to a remote host using TCP.
- `uip_udp_conn * uip_udp_new` (uip_ipaddr_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.
- void `uip_send` (const void *data, int len)
Send data on the current connection.

Variables

- void * `uip_appdata`
Pointer to the application data in the packet buffer.
- `uip_conn * uip_conn`
Pointer to the current TCP connection.
- `uip_conn uip_conns` [UIP_CONNS]
- `uip_udp_conn * uip_udp_conn`
The current UDP connection.
- `uip_udp_conn uip_udp_conns` [UIP_UDP_CONNS]
- `uip_stats uip_stat`

The uIP TCP/IP statistics.

- [u8_t uip_flags](#)
- [uip_ipaddr_t uip_hostaddr](#)
- [uip_ipaddr_t uip_netmask](#)
- [uip_ipaddr_t uip_draddr](#)
- [uip_ipaddr_t uip_hostaddr](#)
- [uip_ipaddr_t uip_draddr](#)
- [uip_ipaddr_t uip_netmask](#)
- [uip_eth_addr uip_ethaddr](#) = { {0,0,0,0,0,0} }
- [u8_t uip_buf](#) [UIP_BUFSIZE+2]

The uIP packet buffer.

- [void * uip_appdata](#)

Pointer to the application data in the packet buffer.

- [void * uip_sappdata](#)
- [u16_t uip_len](#)

The length of the packet in the uip_buf buffer.

- [u16_t uip_slen](#)
- [u8_t uip_flags](#)
- [uip_conn * uip_conn](#)

Pointer to the current TCP connection.

- [uip_conn uip_conns](#) [UIP_CONNS]
- [u16_t uip_listenports](#) [UIP_LISTENPORTS]
- [uip_udp_conn * uip_udp_conn](#)

The current UDP connection.

- [uip_udp_conn uip_udp_conns](#) [UIP_UDP_CONNS]
- [u8_t uip_acc32](#) [4]

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

6.9.2 Define Documentation

6.9.2.1 #define UIP_APPDATA_SIZE

The buffer size available for user data in the [uip_buf](#) buffer.

This macro holds the available size for user data in the [uip_buf](#) buffer. The macro is intended to be used for checking bounds of available user data.

Example:

```
snprintf(uip_appdata, UIP_APPDATA_SIZE, "%u\n", i);
```

Definition at line 1506 of file uip.h.

6.9.3 Function Documentation

6.9.3.1 `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.

Definition at line 1882 of file `uip.c`.

References `HTONS`.

Referenced by `uip_chksum()`, `uip_connect()`, `uip_ipchksum()`, `uip_udp_new()`, and `webclient_get()`.

6.9.3.2 `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.

Definition at line 249 of file `uip.c`.

Referenced by `uip_split_output()`.

6.9.3.3 `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.

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.

Definition at line 311 of file `uip.c`.

References `htons()`.

6.9.3.4 struct `uip_conn*` `uip_connect` (`uip_ipaddr_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 specified 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 available only if support for active open has been configured by defining `UIP_ACTIVE_OPEN` to 1 in `uiptopt.h`.

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

```
uip_ipaddr_t ipaddr;

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

Parameters:

ripaddr The IP address of the remote host.

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.

Definition at line 407 of file `uip.c`.

References `htons()`, `uip_conn::lport`, `uip_conn::tcpstateflags`, `UIP_CLOSED`, `uip_conn`, `uip_conns`, and `UIP_CONNS`.

Referenced by `smtp_send()`, and `webclient_get()`.

6.9.3.5 void `uip_init` (void)

uIP initialization function.

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

Definition at line 379 of file `uip.c`.

References `UIP_LISTENPORTS`.

6.9.3.6 `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.

Definition at line 318 of file `uip.c`.

References `DEBUG_PRINTF`, `htons()`, `UIP_IPH_LEN`, and `UIP_LLH_LEN`.

Referenced by `uip_process()`, and `uip_split_output()`.

6.9.3.7 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.

Definition at line 529 of file uip.c.

References UIP_LISTENPORTS.

Referenced by hello_world_init(), httpd_init(), and telnetd_init().

6.9.3.8 void uip_send (const void * data, int 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 function 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.

Definition at line 1888 of file uip.c.

References uip_sappdata, and uip_slen.

6.9.3.9 void uip_setipid (u16_t id)

uIP initialization function.

This function may be used at boot time to set the initial ip_id.

Definition at line 181 of file uip.c.

6.9.3.10 `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.

Returns:

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

Definition at line 364 of file `uip.c`.

References `UIP_PROTO_TCP`.

Referenced by `uip_split_output()`.

6.9.3.11 `struct uip_udp_conn* uip_udp_new (uip_ipaddr_t * ripaddr, u16_t rport)`

Set up a new UDP connection.

This function sets up a new UDP connection. The function will automatically allocate an unused local port for the new connection. However, another port can be chosen by using the `uip_udp_bind()` call, after the `uip_udp_new()` function has been called.

Example:

```
uip_ipaddr_t addr;
struct uip_udp_conn *c;

uip_ipaddr(&addr, 192,168,2,1);
c = uip_udp_new(&addr, HTONS(12345));
if(c != NULL) {
    uip_udp_bind(c, HTONS(12344));
}
```

Parameters:

ripaddr 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.

Definition at line 473 of file `uip.c`.

References `htons()`, `uip_udp_conn::lport`, `uip_udp_conn`, `uip_udp_conns`, and `UIP_UDP_CONNS`.

Referenced by `resolv_conf()`.

6.9.3.12 `u16_t uip_udpchksum (void)`

Calculate the UDP checksum of the packet in `uip_buf` and `uip_appdata`.

The UDP checksum is the Internet checksum of data contents of the UDP segment, and a pseudo-header as defined in RFC768.

Returns:

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

Referenced by `uip_process()`.

6.9.3.13 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.

Definition at line 518 of file uip.c.

References UIP_LISTENPORTS.

6.9.4 Variable Documentation

6.9.4.1 void* 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\(\)](#).

Definition at line 143 of file uip.c.

Referenced by uip_process(), and uip_split_output().

6.9.4.2 void* 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\(\)](#).

Examples:

[dhcpc.c](#), [resolv.c](#), [telnetd.c](#), and [webclient.c](#).

Definition at line 143 of file uip.c.

Referenced by uip_process(), and uip_split_output().

6.9.4.3 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);
    if(uip_len <= UIP_LLH_LEN + UIP_TCPIP_HLEN) {
        hwsend(&uip_buf[UIP_LLH_LEN], uip_len - UIP_LLH_LEN);
    } else {
        hwsend(&uip_buf[UIP_LLH_LEN], UIP_TCPIP_HLEN);
        hwsend(uip_appdata, uip_len - UIP_TCPIP_HLEN - UIP_LLH_LEN);
    }
}
```

Definition at line 139 of file uip.c.

Referenced by uip_process().

6.9.4.4 struct uip_conn* uip_conn

Pointer to the current TCP connection.

The [uip_conn](#) pointer can be used to access the current TCP connection.

Definition at line 163 of file uip.c.

Referenced by uip_connect().

6.9.4.5 struct uip_conn* uip_conn

Pointer to the current TCP connection.

The [uip_conn](#) pointer can be used to access the current TCP connection.

Examples:

[hello-world.c](#), [smtp.c](#), and [webclient.c](#).

Definition at line 163 of file uip.c.

Referenced by uip_connect().

6.9.4.6 u16_t uip_len

The length of the packet in the uip_buf buffer.

The global variable uip_len holds the length of the packet in the uip_buf buffer.

When the network device driver calls the uIP input function, uip_len should be set to the length of the packet in the uip_buf buffer.

When sending packets, the device driver should use the contents of the uip_len variable to determine the length of the outgoing packet.

Definition at line 155 of file uip.c.

Referenced by uip_arp_arpin(), uip_process(), and uip_split_output().

6.9.4.7 struct uip_stats uip_stat

The uIP TCP/IP statistics.

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

Referenced by `uip_process()`.

6.10 Architecture specific uIP functions

6.10.1 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

- `u8_t uip_acc32` [4]
4-byte array used for the 32-bit sequence number calculations.

6.10.2 Function Documentation

6.10.2.1 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.

Definition at line 249 of file uip.c.

Referenced by uip_split_output().

6.10.2.2 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.

6.10.2.3 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.

6.10.2.4 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.

6.11 uIP Address Resolution Protocol

6.11.1 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.h](#)
Macros and definitions for the ARP module.
- file [uip_arp.c](#)
Implementation of the ARP Address Resolution Protocol.

Data Structures

- struct [uip_eth_hdr](#)
The Ethernet header.

Defines

- #define [UIP_ETHTYPE_ARP](#) 0x0806
- #define [UIP_ETHTYPE_IP](#) 0x0800
- #define [UIP_ETHTYPE_IP6](#) 0x86dd
- #define [uip_arp_ipin\(\)](#)
- #define [ARP_REQUEST](#) 1
- #define [ARP_REPLY](#) 2
- #define [ARP_HWTYPE_ETH](#) 1
- #define [BUF](#) ((struct arp_hdr *)&[uip_buf](#)[0])
- #define [IPBUF](#) ((struct ethip_hdr *)&[uip_buf](#)[0])

Functions

- void [uip_arp_init](#) (void)
Initialize the ARP module.
- 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.

Variables

- `uip_eth_addr` `uip_ethaddr`

6.11.2 Function Documentation

6.11.2.1 void `uip_arp_in` (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`.

Examples:

[example-mainloop-with-arp.c](#).

Definition at line 278 of file `uip_arp.c`.

References `uip_eth_addr::addr`, `ARP_REPLY`, `ARP_REQUEST`, `BUF`, `HTONS`, `uip_ethaddr`, `UIP_ETHTYPE_ARP`, `uip_hostaddr`, `uip_ipaddr_cmp`, and `uip_len`.

6.11.2.2 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`.

Examples:

[example-mainloop-with-arp.c](#).

Definition at line 354 of file uip_arp.c.

References uip_eth_addr::addr, IPBUF, UIP_ARPTAB_SIZE, uip_draddr, uip_hostaddr, uip_ipaddr_cmp, uip_ipaddr_copy, and uip_ipaddr_maskcmp.

6.11.2.3 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.

Examples:

[example-mainloop-with-arp.c](#).

Definition at line 142 of file uip_arp.c.

References UIP_ARP_MAXAGE, and UIP_ARPTAB_SIZE.

6.12 Configuration options for uIP

6.12.1 Detailed Description

uIP is configured using the per-project configuration file [uiptopt.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 "uiptopt.h" that can be copied and modified for each project.

Note:

Most of the configuration options in the [uiptopt.h](#) should not be changed, but rather the per-project uip-conf.h file.

Files

- file [uip-conf.h](#)
An example uIP configuration file.
- file [uiptopt.h](#)
Configuration options for uIP.

Project-specific configuration options

uIP has a number of configuration options that can be overridden for each project. These are kept in a project-specific uip-conf.h file and all configuration names have the prefix UIP_CONF.

- #define [UIP_CONF_MAX_CONNECTIONS](#)
Maximum number of TCP connections.
- #define [UIP_CONF_MAX_LISTENPORTS](#)
Maximum number of listening TCP ports.
- #define [UIP_CONF_BUFFER_SIZE](#)
uIP buffer size.
- #define [UIP_CONF_BYTE_ORDER](#)
CPU byte order.
- #define [UIP_CONF_LOGGING](#)
Logging on or off.
- #define [UIP_CONF_UDP](#)
UDP support on or off.
- #define [UIP_CONF_UDP_CHECKSUMS](#)
UDP checksums on or off.
- #define [UIP_CONF_STATISTICS](#)

uIP statistics on or off

- typedef uint8_t [u8_t](#)
8 bit datatype
- typedef uint16_t [u16_t](#)
16 bit datatype
- typedef unsigned short [uip_stats_t](#)
Statistics datatype.

Static configuration options

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 applicable only if uIP should be run over Ethernet.

All of these should be changed to suit your project.

- #define [UIP_FIXEDADDR](#)
Determines if uIP should use a fixed IP address or not.
- #define [UIP_PINGADDRCONF](#)
Ping IP address assignment.
- #define [UIP_FIXEDETHADDR](#)
Specifies if the uIP ARP module should be compiled with a fixed Ethernet MAC address or not.

IP configuration options

- #define [UIP_TTL](#) 64
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.

UDP configuration options

- #define [UIP_UDP](#)
Toggles whether 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.

TCP configuration options

- #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_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` 5

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 - UIP_TCPIP_HLEN)

The TCP maximum segment size.

- #define `UIP_RECEIVE_WINDOW`

The size of the advertised receiver's window.

- #define `UIP_TIME_WAIT_TIMEOUT` 120

How long a connection should stay in the TIME_WAIT state.

ARP configuration options

- #define `UIP_ARPTAB_SIZE`

The size of the ARP table.

- #define `UIP_ARP_MAXAGE` 120

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

General configuration options

- `#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_BROADCAST`
Broadcast support.
- `#define UIP_LLH_LEN`
The link level header length.
- `void uip_log (char *msg)`
Print out a uIP log message.

CPU architecture configuration

The CPU architecture configuration is where the endianness 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.

- `#define UIP_BYTE_ORDER`
The byte order of the CPU architecture on which uIP is to be run.

Application specific configurations

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 type of the application structure by typedef:ing the type `uip_tcp_appstate_t` and `uip_udp_appstate_t`.

The file containing the definitions must be included in the `uiptopt.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;
};
typedef struct httpd_state uip_tcp_appstate_t
```


- `#define UIP_APPCALL smtp_appcall`
The name of the application function that uIP should call in response to TCP/IP events.
- `typedef smtp_state uip_tcp_appstate_t`
The type of the application state that is to be stored in the `uip_conn` structure.
- `typedef int uip_udp_appstate_t`
The type of the application state that is to be stored in the `uip_conn` structure.

Defines

- `#define UIP_LITTLE_ENDIAN 3412`
- `#define UIP_BIG_ENDIAN 1234`

6.12.2 Define Documentation

6.12.2.1 `#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 configuration option can be turned off to reduce the code size of uIP.

Definition at line 233 of file `uipt.h`.

6.12.2.2 `#define UIP_ARP_MAXAGE 120`

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

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

Definition at line 358 of file `uipt.h`.

Referenced by `uip_arp_timer()`.

6.12.2.3 `#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.

Definition at line 349 of file `uipt.h`.

Referenced by `uip_arp_init()`, `uip_arp_out()`, and `uip_arp_timer()`.

6.12.2.4 `#define UIP_BROADCAST`

Broadcast support.

This flag configures IP broadcast support. This is useful only together with UDP.

Definition at line 423 of file `uipt.h`.

6.12.2.5 `#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.

Definition at line 379 of file `uiptopt.h`.

Referenced by `uip_split_output()`.

6.12.2.6 `#define UIP_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).

Definition at line 475 of file `uiptopt.h`.

6.12.2.7 `#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 approximately 30 bytes of memory.

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 245 of file `uiptopt.h`.

Referenced by `uip_connect()`.

6.12.2.8 `#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 [uiptopt.h](#) file. If not, the macros [uip_sethostaddr\(\)](#), [uip_setdraddr\(\)](#) and [uip_setnetmask\(\)](#) should be used instead.

Definition at line 97 of file `uiptopt.h`.

6.12.2.9 `#define UIP_FIXEETHADDR`

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.

Definition at line 127 of file `uiptopt.h`.

6.12.2.10 `#define UIP_LISTENPORTS`

The maximum number of simultaneously listening TCP ports.

Each listening TCP port requires 2 bytes of memory.

Definition at line 259 of file uipopt.h.

Referenced by uip_init(), uip_listen(), and uip_unlisten().

6.12.2.11 #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.

Definition at line 448 of file uipopt.h.

Referenced by uip_ipchksum(), uip_process(), and uip_split_output().

6.12.2.12 #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.

Definition at line 408 of file uipopt.h.

6.12.2.13 #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.

Definition at line 288 of file uipopt.h.

Referenced by uip_process().

6.12.2.14 #define UIP_MAXSYNRTX 5

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.

Definition at line 297 of file uipopt.h.

Referenced by uip_process().

6.12.2.15 #define UIP_PINGADDRCONF

Ping IP address assignment.

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.

Definition at line 114 of file uipopt.h.

6.12.2.16 **#define UIP_REASSEMBLY**

Turn on support for IP packet reassembly.

uIP supports reassembly of fragmented IP packets. This feature requires an additional 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.

Definition at line 156 of file uipopt.h.

6.12.2.17 **#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) if the application is slow to process incoming data, or high (32768 bytes) if the application processes data quickly.

Definition at line 317 of file uipopt.h.

6.12.2.18 **#define UIP_RTO 3**

The initial retransmission timeout counted in timer pulses.

This should not be changed.

Definition at line 280 of file uipopt.h.

Referenced by uip_process().

6.12.2.19 **#define UIP_STATISTICS**

Determines if statistics support should be compiled in.

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

Definition at line 393 of file uipopt.h.

6.12.2.20 **#define UIP_TCP_MSS (UIP_BUFSIZE - UIP_LLH_LEN - UIP_TCPIP_HLEN)**

The TCP maximum segment size.

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

Definition at line 305 of file uipopt.h.

6.12.2.21 **#define UIP_TIME_WAIT_TIMEOUT 120**

How long a connection should stay in the TIME_WAIT state.

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

Definition at line 328 of file uipopt.h.

Referenced by uip_process().

6.12.2.22 `#define UIP_TTL 64`

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

This should normally not be changed.

Definition at line 141 of file `uiop.h`.

6.12.2.23 `#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.

Definition at line 195 of file `uiop.h`.

6.12.2.24 `#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.

Definition at line 273 of file `uiop.h`.

6.12.3 Typedef Documentation

6.12.3.1 `typedef uint16_t u16_t`

16 bit datatype

This typedef defines the 16-bit type used throughout uIP.

Examples:

[dhcpc.c](#), [dhcpc.h](#), [resolv.c](#), [resolv.h](#), [smtp.c](#), [smtp.h](#), [telnetd.c](#), and [uip-conf.h](#).

Definition at line 76 of file `uip-conf.h`.

6.12.3.2 `typedef uint8_t u8_t`

8 bit datatype

This typedef defines the 8-bit type used throughout uIP.

Examples:

[dhcpc.c](#), [dhcpc.h](#), [resolv.c](#), [smtp.h](#), [telnetd.c](#), [telnetd.h](#), and [uip-conf.h](#).

Definition at line 67 of file `uip-conf.h`.

6.12.3.3 `typedef unsigned short uip_stats_t`

Statistics datatype.

This typedef defines the datatype used for keeping statistics in uIP.

Definition at line 86 of file `uip-conf.h`.

6.12.3.4 typedef `uip_tcp_appstate_t`

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

This usually is typedef:ed to a struct holding application state information.

Examples:

`smtp.h`, `telnetd.h`, and `webclient.h`.

Definition at line 98 of file `smtp.h`.

6.12.3.5 typedef `uip_udp_appstate_t`

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

This usually is typedef:ed to a struct holding application state information.

Examples:

`dhcpc.h`.

Definition at line 47 of file `resolv.h`.

6.12.4 Function Documentation

6.12.4.1 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.

6.13 uIP TCP throughput booster hack

6.13.1 Detailed Description

The basic uIP TCP implementation only allows each TCP connection to have a single TCP segment in flight at any given time.

Because of the delayed ACK algorithm employed by most TCP receivers, uIP's limit on the amount of in-flight TCP segments seriously reduces the maximum achievable throughput for sending data from uIP.

The uip-split module is a hack which tries to remedy this situation. By splitting maximum sized outgoing TCP segments into two, the delayed ACK algorithm is not invoked at TCP receivers. This improves the throughput when sending data from uIP by orders of magnitude.

The uip-split module uses the uip-fw module (uIP IP packet forwarding) for sending packets. Therefore, the uip-fw module must be set up with the appropriate network interfaces for this module to work.

Files

- file [uip-split.h](#)

Module for splitting outbound TCP segments in two to avoid the delayed ACK throughput degradation.

Functions

- void [uip_split_output](#) (void)

Handle outgoing packets.

6.13.2 Function Documentation

6.13.2.1 void uip_split_output (void)

Handle outgoing packets.

This function inspects an outgoing packet in the uip_buf buffer and sends it out using the uip_fw_output() function. If the packet is a full-sized TCP segment it will be split into two segments and transmitted separately. This function should be called instead of the actual device driver output function, or the uip_fw_output() function.

The headers of the outgoing packet is assumed to be in the uip_buf buffer and the payload is assumed to be wherever uip_appdata points. The length of the outgoing packet is assumed to be in the uip_len variable.

Definition at line 49 of file uip-split.c.

References BUF, uip_acc32, uip_add32(), uip_appdata, UIP_BUFSIZE, uip_ipchksum(), UIP_IPH_LEN, uip_len, UIP_LLH_LEN, UIP_PROTO_TCP, uip_tcpchksum(), and UIP_TCPIP_HLEN.

6.14 Local continuations

6.14.1 Detailed Description

Local continuations form the basis for implementing protothreads.

A local continuation can be *set* in a specific function to capture the state of the function. After a local continuation has been set can be *resumed* in order to restore the state of the function at the point where the local continuation was set.

Files

- file [lc.h](#)

Local continuations.

- file [lc-switch.h](#)

Implementation of local continuations based on switch() statment.

- file [lc-addrlabels.h](#)

Implementation of local continuations based on the "Labels as values" feature of gcc.

Defines

- `#define __LC_SWTICH_H__`
- `#define LC_INIT(s) s = 0;`
- `#define LC_RESUME(s) switch(s) { case 0:`
- `#define LC_SET(s) s = __LINE__; case __LINE__:`
- `#define LC_END(s) }`
- `#define LC_INIT(s) s = NULL`
- `#define LC_RESUME(s)`
- `#define LC_SET(s) do { ({ __label__ resume; resume: (s) = &&resume; }); } while(0)`
- `#define LC_END(s)`

Typedefs

- typedef unsigned short [lc_t](#)
- typedef void * [lc_t](#)

6.15 Timer library

6.15.1 Detailed Description

The timer library provides functions for setting, resetting and restarting timers, and for checking if a timer has expired.

An application must "manually" check if its timers have expired; this is not done automatically.

A timer is declared as a `struct timer` and all access to the timer is made by a pointer to the declared timer.

Note:

The timer library uses the [Clock library](#) to measure time. Intervals should be specified in the format used by the clock library.

Files

- file [timer.h](#)

Timer library header file.

- file [timer.c](#)

Timer library implementation.

Data Structures

- struct [timer](#)

A timer.

Functions

- void [timer_set](#) (struct [timer](#) *t, clock_time_t interval)

Set a timer.

- void [timer_reset](#) (struct [timer](#) *t)

Reset the timer with the same interval.

- void [timer_restart](#) (struct [timer](#) *t)

Restart the timer from the current point in time.

- int [timer_expired](#) (struct [timer](#) *t)

Check if a timer has expired.

6.15.2 Function Documentation

6.15.2.1 `int timer_expired (struct timer * t)`

Check if a timer has expired.

This function tests if a timer has expired and returns true or false depending on its status.

Parameters:

t A pointer to the timer

Returns:

Non-zero if the timer has expired, zero otherwise.

Examples:

[dhcpc.c](#), [example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 121 of file `timer.c`.

References `clock_time()`, `interval`, and `start`.

6.15.2.2 `void timer_reset (struct timer * t)`

Reset the timer with the same interval.

This function resets the timer with the same interval that was given to the [timer_set\(\)](#) function. The start point of the interval is the exact time that the timer last expired. Therefore, this function will cause the timer to be stable over time, unlike the `timer_rester()` function.

Parameters:

t A pointer to the timer.

See also:

[timer_restart\(\)](#)

Examples:

[example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 84 of file `timer.c`.

References `interval`, and `start`.

6.15.2.3 `void timer_restart (struct timer * t)`

Restart the timer from the current point in time.

This function restarts a timer with the same interval that was given to the [timer_set\(\)](#) function. The timer will start at the current time.

Note:

A periodic timer will drift if this function is used to reset it. For preioric timers, use the [timer_reset\(\)](#) function instead.

Parameters:

t A pointer to the timer.

See also:[timer_reset\(\)](#)

Definition at line 104 of file timer.c.

References [clock_time\(\)](#), and [start](#).

6.15.2.4 void timer_set (struct [timer](#) * *t*, clock_time_t *interval*)

Set a timer.

This function is used to set a timer for a time sometime in the future. The function [timer_expired\(\)](#) will evaluate to true after the timer has expired.

Parameters:

t A pointer to the timer

interval The interval before the timer expires.

Examples:

[dhcpc.c](#), [example-mainloop-with-arp.c](#), and [example-mainloop-without-arp.c](#).

Definition at line 64 of file timer.c.

References [clock_time\(\)](#), [interval](#), and [start](#).

6.16 Clock interface

6.16.1 Detailed Description

The clock interface is the interface between the [timer library](#) and the platform specific clock functionality.

The clock interface must be implemented for each platform that uses the [timer library](#).

The clock interface does only one thing: it measures time. The clock interface provides a macro, `CLOCK_SECOND`, which corresponds to one second of system time.

See also:

[Timer library](#)

Defines

- `#define CLOCK_SECOND`
A second, measured in system clock time.

Functions

- `void clock_init (void)`
Initialize the clock library.
- `clock_time_t clock_time (void)`
Get the current clock time.

6.16.2 Function Documentation

6.16.2.1 void clock_init (void)

Initialize the clock library.

This function initializes the clock library and should be called from the `main()` function of the system.

6.16.2.2 clock_time_t clock_time (void)

Get the current clock time.

This function returns the current system clock time.

Returns:

The current clock time, measured in system ticks.

Referenced by `timer_expired()`, `timer_restart()`, and `timer_set()`.

6.17 Protosockets library

6.17.1 Detailed Description

The protosocket library provides an interface to the uIP stack that is similar to the traditional BSD socket interface.

Unlike programs written for the ordinary uIP event-driven interface, programs written with the protosocket library are executed in a sequential fashion and does not have to be implemented as explicit state machines.

Protosockets only work with TCP connections.

The protosocket library uses [Protothreads](#) protothreads to provide sequential control flow. This makes the protosockets lightweight in terms of memory, but also means that protosockets inherits the functional limitations of protothreads. Each protosocket lives only within a single function. Automatic variables (stack variables) are not retained across a protosocket library function call.

Note:

Because the protosocket library uses protothreads, local variables will not always be saved across a call to a protosocket library function. It is therefore advised that local variables are used with extreme care.

The protosocket library provides functions for sending data without having to deal with retransmissions and acknowledgements, as well as functions for reading data without having to deal with data being split across more than one TCP segment.

Because each protosocket runs as a protothread, the protosocket has to be started with a call to [PSOCK_BEGIN\(\)](#) at the start of the function in which the protosocket is used. Similarly, the protosocket protothread can be terminated by a call to [PSOCK_EXIT\(\)](#).

Files

- file [psock.h](#)
Protosocket library header file.

Data Structures

- struct [psock_buf](#)
- struct [psock](#)
The representation of a protosocket.

Defines

- #define [PSOCK_INIT](#)(psock, buffer, buffersize)
Initialize a protosocket.
- #define [PSOCK_BEGIN](#)(psock)
Start the protosocket protothread in a function.
- #define [PSOCK_SEND](#)(psock, data, datalen)

Send data.

- #define `PSOCK_SEND_STR(psock, str)`
Send a null-terminated string.
- #define `PSOCK_GENERATOR_SEND(psock, generator, arg)`
Generate data with a function and send it.
- #define `PSOCK_CLOSE(psock)`
Close a protosocket.
- #define `PSOCK_READBUF(psock)`
Read data until the buffer is full.
- #define `PSOCK_READTO(psock, c)`
Read data up to a specified character.
- #define `PSOCK_DATALEN(psock)`
The length of the data that was previously read.
- #define `PSOCK_EXIT(psock)`
Exit the protosocket's protothread.
- #define `PSOCK_CLOSE_EXIT(psock)`
Close a protosocket and exit the protosocket's protothread.
- #define `PSOCK_END(psock)`
Declare the end of a protosocket's protothread.
- #define `PSOCK_NEWDATA(psock)`
Check if new data has arrived on a protosocket.
- #define `PSOCK_WAIT_UNTIL(psock, condition)`
Wait until a condition is true.
- #define `PSOCK_WAIT_THREAD(psock, condition) PT_WAIT_THREAD(&((psock) → pt), (condition))`

Functions

- `u16_t psock_dataLEN (struct psock *psock)`
- `char psock_newdata (struct psock *s)`

6.17.2 Define Documentation

6.17.2.1 #define `PSOCK_BEGIN(psock)`

Start the protosocket protothread in a function.

This macro starts the protothread associated with the protosocket and must come before other protosocket calls in the function it is used.

Parameters:

psock (struct psock *) A pointer to the protosocket to be started.

Examples:

[hello-world.c](#), and [smtp.c](#).

Definition at line 158 of file psock.h.

6.17.2.2 #define PSOCK_CLOSE([psock](#))

Close a protosocket.

This macro closes a protosocket and can only be called from within the protothread in which the protosocket lives.

Parameters:

psock (struct psock *) A pointer to the protosocket that is to be closed.

Examples:

[hello-world.c](#), and [smtp.c](#).

Definition at line 235 of file psock.h.

6.17.2.3 #define PSOCK_CLOSE_EXIT([psock](#))

Close a protosocket and exit the protosocket's protothread.

This macro closes a protosocket and exits the protosocket's protothread.

Parameters:

psock (struct psock *) A pointer to the protosocket.

Definition at line 308 of file psock.h.

6.17.2.4 #define PSOCK_DATALEN([psock](#))

The length of the data that was previously read.

This macro returns the length of the data that was previously read using [PSOCK_READTO\(\)](#) or [PSOCK_READ\(\)](#).

Parameters:

psock (struct psock *) A pointer to the protosocket holding the data.

Definition at line 281 of file psock.h.

6.17.2.5 #define PSOCK_END([psock](#))

Declare the end of a protosocket's protothread.

This macro is used for declaring that the protosocket's protothread ends. It must always be used together with a matching [PSOCK_BEGIN\(\)](#) macro.

Parameters:

psock (struct psock *) A pointer to the protosocket.

Examples:

[hello-world.c](#), and [smtp.c](#).

Definition at line 325 of file psock.h.

6.17.2.6 #define PSOCK_EXIT(*psock*)

Exit the protosocket's protothread.

This macro terminates the protothread of the protosocket and should almost always be used in conjunction with [PSOCK_CLOSE\(\)](#).

See also:

[PSOCK_CLOSE_EXIT\(\)](#)

Parameters:

psock (struct psock *) A pointer to the protosocket.

Examples:

[smtp.c](#).

Definition at line 297 of file psock.h.

6.17.2.7 #define PSOCK_GENERATOR_SEND(*psock*, generator, arg)

Generate data with a function and send it.

Parameters:

psock Pointer to the protosocket.

generator Pointer to the generator function

arg Argument to the generator function

This function generates data and sends it over the protosocket. This can be used to dynamically generate data for a transmission, instead of generating the data in a buffer beforehand. This function reduces the need for buffer memory. The generator function is implemented by the application, and a pointer to the function is given as an argument with the call to [PSOCK_GENERATOR_SEND\(\)](#).

The generator function should place the generated data directly in the uip_appdata buffer, and return the length of the generated data. The generator function is called by the protosocket layer when the data first is sent, and once for every retransmission that is needed.

Definition at line 219 of file psock.h.

6.17.2.8 #define PSOCK_INIT(*psock*, buffer, buffersize)

Initialize a protosocket.

This macro initializes a protosocket and must be called before the protosocket is used. The initialization also specifies the input buffer for the protosocket.

Parameters:

psock (struct psock *) A pointer to the protosocket to be initialized

buffer (char *) A pointer to the input buffer for the protosocket.

buffer_size (unsigned int) The size of the input buffer.

Examples:

[hello-world.c](#), and [smtp.c](#).

Definition at line 144 of file psock.h.

Referenced by [hello_world_appcall\(\)](#), [httpd_appcall\(\)](#), and [smtp_send\(\)](#).

6.17.2.9 #define PSOCK_NEWDATA(*psock*)

Check if new data has arrived on a protosocket.

This macro is used in conjunction with the [PSOCK_WAIT_UNTIL\(\)](#) macro to check if data has arrived on a protosocket.

Parameters:

psock (struct psock *) A pointer to the protosocket.

Definition at line 339 of file psock.h.

6.17.2.10 #define PSOCK_READBUF(*psock*)

Read data until the buffer is full.

This macro will block waiting for data and read the data into the input buffer specified with the call to [PSOCK_INIT\(\)](#). Data is read until the buffer is full..

Parameters:

psock (struct psock *) A pointer to the protosocket from which data should be read.

Definition at line 250 of file psock.h.

6.17.2.11 #define PSOCK_READTO(*psock*, *c*)

Read data up to a specified character.

This macro will block waiting for data and read the data into the input buffer specified with the call to [PSOCK_INIT\(\)](#). Data is only read until the specified character appears in the data stream.

Parameters:

psock (struct psock *) A pointer to the protosocket from which data should be read.

c (char) The character at which to stop reading.

Examples:

[hello-world.c](#), and [smtp.c](#).

Definition at line 268 of file psock.h.

6.17.2.12 #define PSOCK_SEND(*psock*, *data*, *datalen*)

Send data.

This macro sends data over a protosocket. The protosocket protothread blocks until all data has been sent and is known to have been received by the remote end of the TCP connection.

Parameters:

psock (struct psock *) A pointer to the protosocket over which data is to be sent.

data (char *) A pointer to the data that is to be sent.

datalen (unsigned int) The length of the data that is to be sent.

Examples:

[smtp.c](#).

Definition at line 178 of file psock.h.

6.17.2.13 #define PSOCK_SEND_STR(*psock*, *str*)

Send a null-terminated string.

Parameters:

psock Pointer to the protosocket.

str The string to be sent.

This function sends a null-terminated string over the protosocket.

Examples:

[hello-world.c](#), and [smtp.c](#).

Definition at line 191 of file psock.h.

6.17.2.14 #define PSOCK_WAIT_UNTIL(*psock*, *condition*)

Wait until a condition is true.

This macro blocks the protothread until the specified condition is true. The macro [PSOCK_NEWDATA\(\)](#) can be used to check if new data arrives when the protosocket is waiting.

Typically, this macro is used as follows:

```
PT_THREAD(thread(struct psock *s, struct timer *t))
{
    PSOCK_BEGIN(s);

    PSOCK_WAIT_UNTIL(s, PSOCK_NEWADATA(s) || timer_expired(t));

    if (PSOCK_NEWADATA(s)) {
        PSOCK_READTO(s, '\n');
    } else {
        handle_timed_out(s);
    }

    PSOCK_END(s);
}
```

Parameters:

psock (struct psock *) A pointer to the protosocket.

condition The condition to wait for.

Definition at line 372 of file psock.h.

6.18 Memory block management functions

6.18.1 Detailed Description

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.

Files

- file `memb.c`
Memory block allocation routines.
- file `memb.h`
Memory block allocation routines.

Data Structures

- struct `memb_blocks`

Defines

- `#define MEMB_CONCAT2(s1, s2) s1##s2`
- `#define MEMB_CONCAT(s1, s2) MEMB_CONCAT2(s1, s2)`
- `#define MEMB(name, structure, num)`
Declare a memory block.

Functions

- void `memb_init` (struct `memb_blocks` *m)
Initialize a memory block that was declared with `MEMB()`.
- void * `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, void *ptr)
Deallocate a memory block from a memory block previously declared with `MEMB()`.

6.18.2 Define Documentation

6.18.2.1 #define MEMB(name, structure, num)

Value:

```
static char MEMB_CONCAT(name, _memb_count) [num]; \
static structure MEMB_CONCAT(name, _memb_mem) [num]; \
static struct memb_blocks name = {sizeof(structure), num, \
MEMB_CONCAT(name, _memb_count), \
(void *)MEMB_CONCAT(name, _memb_mem) }
```

Declare a memory block.

This macro is used to staticall declare a block of memory that can be used by the block allocation functions. The macro statically declares a C array with a size that matches the specified number of blocks and their individual sizes.

Example:

```
MEMB(connections, sizeof(struct connection), 16);
```

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.

Examples:

[telnetd.c](#).

Definition at line 98 of file memb.h.

6.18.3 Function Documentation

6.18.3.1 void * memb_alloc (struct [memb_blocks](#) * *m*)

Allocate a memory block from a block of memory declared with [MEMB\(\)](#).

Parameters:

m A memory block previously declared with [MEMB\(\)](#).

Examples:

[telnetd.c](#).

Definition at line 59 of file memb.c.

References [memb_blocks::count](#), [memb_blocks::mem](#), [memb_blocks::num](#), and [memb_blocks::size](#).

6.18.3.2 char memb_free (struct [memb_blocks](#) * *m*, void * *ptr*)

Deallocate a memory block from a memory block previously declared with [MEMB\(\)](#).

Parameters:

m m A memory block previously 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.

Examples:

[telnetd.c](#).

Definition at line 79 of file memb.c.

References memb_blocks::count, memb_blocks::mem, and memb_blocks::size.

6.18.3.3 void memb_init (struct [memb_blocks](#) * *m*)

Initialize a memory block that was declared with [MEMB\(\)](#).

Parameters:

m A memory block previously declared with [MEMB\(\)](#).

Examples:

[telnetd.c](#).

Definition at line 52 of file memb.c.

References memb_blocks::count, memb_blocks::mem, memb_blocks::num, and memb_blocks::size.

Referenced by telnetd_init().

6.19 DNS resolver

6.19.1 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-existent), the resolver code calls a callback function called `resolv_found()` that must be implemented by the module that uses the resolver.

Files

- file `resolv.h`
DNS resolver code header file.
- file `resolv.c`
DNS host name to IP address resolver.

Defines

- `#define UIP_UDP_APPCALL resolv_appcall`
- `#define NULL (void *)0`
- `#define MAX_RETRIES 8`
- `#define RESOLV_ENTRIES 4`

Functions

- void `resolv_appcall` (void)
- 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)
Initialize 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.

6.19.2 Function Documentation

6.19.2.1 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.

Examples:

[resolv.c](#), and [resolv.h](#).

Definition at line 438 of file `resolv.c`.

References `HTONS`, `NULL`, `uip_udp_new()`, and `uip_udp_remove`.

6.19.2.2 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.

Examples:

[resolv.c](#), and [resolv.h](#).

6.19.2.3 `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.

Examples:

[resolv.c](#), and [resolv.h](#).

Definition at line 422 of file `resolv.c`.

References `NULL`, and `uip_udp_conn::ripaddr`.

6.19.2.4 `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.

Examples:

[resolv.c](#), [resolv.h](#), and [webclient.c](#).

Definition at line 396 of file [resolv.c](#).

References [RESOLV_ENTRIES](#), and [STATE_DONE](#).

Referenced by [webclient_appcall\(\)](#), and [webclient_get\(\)](#).

6.19.2.5 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.

Examples:

[resolv.c](#), [resolv.h](#), and [webclient.c](#).

Definition at line 350 of file [resolv.c](#).

References [RESOLV_ENTRIES](#), and [STATE_UNUSED](#).

Referenced by [webclient_appcall\(\)](#).

6.20 SMTP E-mail sender

6.20.1 Detailed Description

The Simple Mail Transfer Protocol (SMTP) as defined by RFC821 is the standard way of sending and transferring 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.h](#)
SMTP header file.
- file [smtp.c](#)
SMTP example implementation.

Data Structures

- struct [smtp_state](#)

Defines

- #define [SMTP_ERR_OK](#) 0
Error number that signifies a non-error condition.
- #define [SMTP_SEND](#)(to, cc, from, subject, msg) smtp_send(to, cc, from, subject, msg, strlen(msg))
- #define [ISO_nl](#) 0x0a
- #define [ISO_cr](#) 0x0d
- #define [ISO_period](#) 0x2e
- #define [ISO_2](#) 0x32
- #define [ISO_3](#) 0x33
- #define [ISO_4](#) 0x34
- #define [ISO_5](#) 0x35

Functions

- void [smtp_done](#) (unsigned char error)
Callback function that is called when an e-mail transmission is done.
- void [smtp_init](#) (void)
- void [smtp_appcall](#) (void)
- void [smtp_configure](#) (char *lhostname, void *server)
Specify an SMTP server and hostname.
- unsigned char [smtp_send](#) (char *to, char *cc, char *from, char *subject, char *msg, [u16_t](#) msglen)
Send an e-mail.

6.20.2 Function Documentation

6.20.2.1 void smtp_configure (char * *lhostname*, void * *server*)

Specify 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.

Definition at line 216 of file smtp.c.

References uip_ipaddr_copy.

6.20.2.2 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 occurred, or SMTP_ERR_OK.

Examples:

[smtp.c](#), and [smtp.h](#).

Referenced by smtp_appcall().

6.20.2.3 unsigned char smtp_send (char * *to*, char * *cc*, 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.

cc The e-mail address of the CC: receivers 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.

Definition at line 233 of file smtp.c.

References smtp_state::from, HTONS, smtp_state::msg, smtp_state::msglen, NULL, PSOCK_INIT, smtp_state::subject, smtp_state::to, and uip_connect().

6.21 Telnet server

6.21.1 Detailed Description

The uIP telnet server.

Files

- file [telnetd.h](#)
Shell server.
- file [telnetd.c](#)
Shell server.
- file [shell.h](#)
Simple shell, header file.
- file [shell.c](#)
Simple shell.

Data Structures

- struct [telnetd_state](#)

Defines

- #define [TELNETD_CONF_LINELEN](#) 40
- #define [TELNETD_CONF_NUMLINES](#) 16
- #define [UIP_APPCALL](#) telnetd_appcall
- #define [ISO_nl](#) 0x0a
- #define [ISO_cr](#) 0x0d
- #define [STATE_NORMAL](#) 0
- #define [STATE_IAC](#) 1
- #define [STATE_WILL](#) 2
- #define [STATE_WONT](#) 3
- #define [STATE_DO](#) 4
- #define [STATE_DONT](#) 5
- #define [STATE_CLOSE](#) 6
- #define [TELNET_IAC](#) 255
- #define [TELNET_WILL](#) 251
- #define [TELNET_WONT](#) 252
- #define [TELNET_DO](#) 253
- #define [TELNET_DONT](#) 254
- #define [SHELL_PROMPT](#) "uIP 1.0> "

Typedefs

- typedef [telnetd_state](#) [uip_tcp_appstate_t](#)

Functions

- void [telnetd_appcall](#) (void)
- void [shell_quit](#) (char *)
Quit the shell.
- void [shell_prompt](#) (char *prompt)
Print a prompt to the shell window.
- void [shell_output](#) (char *str1, char *str2)
Print a string to the shell window.
- void [telnetd_init](#) (void)
- void [shell_init](#) (void)
Initialize the shell.
- void [shell_start](#) (void)
Start the shell back-end.
- void [shell_input](#) (char *command)
Process a shell command.

6.21.2 Function Documentation

6.21.2.1 void [shell_init](#) (void)

Initialize the shell.

Called when the shell front-end process starts. This function may be used to start listening for signals.

Examples:

[telnetd.c](#).

Definition at line 117 of file [shell.c](#).

Referenced by [telnetd_init\(\)](#).

6.21.2.2 void [shell_input](#) (char * *command*)

Process a shell command.

This function will be called by the shell GUI / telnet server when a command has been entered that should be processed by the shell back-end.

Parameters:

command The command to be processed.

Examples:

[telnetd.c](#).

Definition at line 130 of file [shell.c](#).

References [SHELL_PROMPT](#), and [shell_prompt\(\)](#).

6.21.2.3 void shell_output (char * *str1*, char * *str2*)

Print a string to the shell window.

This function is implemented by the shell GUI / telnet server and can be called by the shell back-end to output a string in the shell window. The string is automatically appended with a linebreak.

Parameters:

str1 The first half of the string to be output.

str2 The second half of the string to be output.

Examples:

[telnetd.c](#).

Definition at line 125 of file telnetd.c.

References ISO_cr, ISO_nl, NULL, and TELNETD_CONF_LINELEN.

Referenced by shell_start().

6.21.2.4 void shell_prompt (char * *prompt*)

Print a prompt to the shell window.

This function can be used by the shell back-end to print out a prompt to the shell window.

Parameters:

prompt The prompt to be printed.

Examples:

[telnetd.c](#).

Definition at line 113 of file telnetd.c.

References NULL, and TELNETD_CONF_LINELEN.

Referenced by shell_input(), and shell_start().

6.21.2.5 void shell_start (void)

Start the shell back-end.

Called by the front-end when a new shell is started.

Examples:

[telnetd.c](#).

Definition at line 122 of file shell.c.

References shell_output(), SHELL_PROMPT, and shell_prompt().

Referenced by telnetd_appcall().

6.22 Hello, world

6.22.1 Detailed Description

A small example showing how to write applications with [protosockets](#).

Files

- file [hello-world.h](#)

Header file for an example of how to write uIP applications with protosockets.

- file [hello-world.c](#)

An example of how to write uIP applications with protosockets.

Data Structures

- struct [hello_world_state](#)

Defines

- `#define UIP_APPCALL hello_world_appcall`

Functions

- void [hello_world_appcall](#) (void)
- void [hello_world_init](#) (void)

6.23 Web client

6.23.1 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.h](#)
Header file for the HTTP client.
- file [webclient.c](#)
Implementation of the HTTP client.

Data Structures

- struct [webclient_state](#)

Defines

- #define [WEBCLIENT_CONF_MAX_URLLEN](#) 100
- #define [UIP_APPCALL](#) webclient_appcall
- #define [WEBCLIENT_TIMEOUT](#) 100
- #define [WEBCLIENT_STATE_STATUSLINE](#) 0
- #define [WEBCLIENT_STATE_HEADERS](#) 1
- #define [WEBCLIENT_STATE_DATA](#) 2
- #define [WEBCLIENT_STATE_CLOSE](#) 3
- #define [HTTPFLAG_NONE](#) 0
- #define [HTTPFLAG_OK](#) 1
- #define [HTTPFLAG_MOVED](#) 2
- #define [HTTPFLAG_ERROR](#) 3
- #define [ISO_nl](#) 0x0a
- #define [ISO_cr](#) 0x0d
- #define [ISO_space](#) 0x20

Typedefs

- typedef [webclient_state](#) uip_tcp_appstate_t

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.
- void `webclient_appcall` (void)
- 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.

6.23.2 Function Documentation

6.23.2.1 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.

Examples:

[webclient.c](#), and [webclient.h](#).

Referenced by `webclient_appcall()`.

6.23.2.2 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.

Examples:

[webclient.c](#), and [webclient.h](#).

Referenced by `webclient_appcall()`.

6.23.2.3 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.

Examples:

[webclient.c](#), and [webclient.h](#).

Referenced by `webclient_appcall()`.

6.23.2.4 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, repeatedly, 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.

Examples:

[webclient.c](#), and [webclient.h](#).

Referenced by `webclient_appcall()`.

6.23.2.5 `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.

Examples:

`webclient.c`, and `webclient.h`.

Definition at line 93 of file `webclient.c`.

References `webclient_state::file`.

6.23.2.6 `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.

Examples:

`webclient.c`, and `webclient.h`.

Definition at line 140 of file `webclient.c`.

References `webclient_state::file`, `webclient_state::host`, `htons()`, `NULL`, `webclient_state::port`, `resolv_lookup()`, and `uip_connect()`.

Referenced by `webclient_appcall()`.

6.23.2.7 `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.

Examples:

[webclient.c](#), and [webclient.h](#).

Definition at line 99 of file [webclient.c](#).

References [webclient_state::host](#).

6.23.2.8 `char * webclient_mimetype (void)`

Obtain the MIME type of the current HTTP data stream.

Returns:

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

Examples:

[webclient.c](#), and [webclient.h](#).

Definition at line 87 of file [webclient.c](#).

References [webclient_state::mimetype](#).

6.23.2.9 `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.

Examples:

[webclient.c](#), and [webclient.h](#).

Definition at line 105 of file [webclient.c](#).

References [webclient_state::port](#).

6.23.2.10 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.

Examples:

[webclient.c](#), and [webclient.h](#).

Referenced by webclient_appcall().

6.24 Web server

6.24.1 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.

Files

- file [httpd-cgi.h](#)
Web server script interface header file.
- file [httpd-cgi.c](#)
Web server script interface.
- file [httpd.c](#)
Web server.

Data Structures

- struct [httpd_cgi_call](#)

Defines

- #define [HTTPD_CGI_CALL](#)(name, str, function)
HTTPD CGI function declaration.
- #define [STATE_WAITING](#) 0
- #define [STATE_OUTPUT](#) 1
- #define [ISO_nl](#) 0x0a
- #define [ISO_space](#) 0x20
- #define [ISO_bang](#) 0x21
- #define [ISO_percent](#) 0x25
- #define [ISO_period](#) 0x2e
- #define [ISO_slash](#) 0x2f
- #define [ISO_colon](#) 0x3a

Functions

- [httpd_cgifunction](#) [httpd_cgi](#) (char *name)
- void [httpd_appcall](#) (void)
- void [httpd_init](#) (void)
Initialize the web server.

6.24.2 Define Documentation

6.24.2.1 #define HTTPD_CGI_CALL(name, str, function)

HTTPD CGI function declaration.

Parameters:

name The C variable name of the function

str The string name of the function, used in the script file

function A pointer to the function that implements it

This macro is used for declaring a HTTPD CGI function. This function is then added to the list of HTTPD CGI functions with the `httpd_cgi_add()` function.

Definition at line 77 of file `httpd-cgi.h`.

6.24.3 Function Documentation

6.24.3.1 void httpd_init (void)

Initialize the web server.

This function initializes the web server and should be called at system boot-up.

Definition at line 333 of file `httpd.c`.

References `HTONS`, and `uip_listen()`.

Chapter 7

uIP 1.0 Data Structure Documentation

7.1 dhcpc_state Struct Reference

7.1.1 Detailed Description

Examples:

[dhcpc.c](#), and [dhcpc.h](#).

Definition at line 39 of file [dhcpc.h](#).

Data Fields

- [pt](#) [pt](#)
- [char](#) [state](#)
- [uip_udp_conn](#) * [conn](#)
- [timer](#) [timer](#)
- [u16_t](#) [ticks](#)
- [const void](#) * [mac_addr](#)
- [int](#) [mac_len](#)
- [u8_t](#) [serverid](#) [4]
- [u16_t](#) [lease_time](#) [2]
- [u16_t](#) [ipaddr](#) [2]
- [u16_t](#) [netmask](#) [2]
- [u16_t](#) [dnsaddr](#) [2]
- [u16_t](#) [default_router](#) [2]

7.2 hello_world_state Struct Reference

7.2.1 Detailed Description

Examples:

[hello-world.c](#), and [hello-world.h](#).

Definition at line 36 of file [hello-world.h](#).

Data Fields

- [psock](#) [p](#)
- char [inputbuffer](#) [10]
- char [name](#) [40]

7.3 httpd_cgi_call Struct Reference

7.3.1 Detailed Description

Definition at line 60 of file httpd-cgi.h.

Data Fields

- const char * [name](#)
- const httpd_cgifunction [function](#)

7.4 httpd_state Struct Reference

7.4.1 Detailed Description

Definition at line 41 of file httpd.h.

Data Fields

- unsigned char [timer](#)
- [psock](#) sin [sout](#)
- [pt](#) outputpt [scriptpt](#)
- char [inputbuf](#) [50]
- char [filename](#) [20]
- char [state](#)
- httpd_fs_file [file](#)
- int [len](#)
- char * [scriptptr](#)
- int [scriptlen](#)
- unsigned short [count](#)

7.5 memb_blocks Struct Reference

7.5.1 Detailed Description

Definition at line 105 of file memb.h.

Data Fields

- unsigned short [size](#)
- unsigned short [num](#)
- char * [count](#)
- void * [mem](#)

7.6 psock Struct Reference

```
#include <psock.h>
```

7.6.1 Detailed Description

The representation of a protosocket.

The protosocket structure is an opaque structure with no user-visible elements.

Examples:

[hello-world.h](#).

Definition at line 106 of file psock.h.

Data Fields

- [pt](#) [pt](#) [psockpt](#)
- [const](#) [u8_t](#) * [sendptr](#)
- [u8_t](#) * [readptr](#)
- [char](#) * [bufptr](#)
- [u16_t](#) [sendlen](#)
- [u16_t](#) [readlen](#)
- [psock_buf](#) [buf](#)
- [unsigned](#) [int](#) [bufsize](#)
- [unsigned](#) [char](#) [state](#)

7.7 psock_buf Struct Reference

7.7.1 Detailed Description

Definition at line 95 of file psock.h.

Data Fields

- [u8_t](#) * [ptr](#)
- unsigned short [left](#)

7.8 pt Struct Reference

7.8.1 Detailed Description

Examples:

[dhcpc.h](#).

Definition at line 54 of file pt.h.

Data Fields

- [lc_t lc](#)

7.9 smtp_state Struct Reference

7.9.1 Detailed Description

Examples:

[hello-world.h](#), [smtp.c](#), and [smtp.h](#).

Definition at line 81 of file [smtp.h](#).

Data Fields

- [u8_t](#) state
- char * [to](#)
- char * [from](#)
- char * [subject](#)
- char * [msg](#)
- [u16_t](#) msglen
- [u16_t](#) sentlen
- [u16_t](#) textlen
- [u16_t](#) sendptr

7.10 telnetd_state Struct Reference

7.10.1 Detailed Description

Examples:

[telnetd.c](#), and [telnetd.h](#).

Definition at line 69 of file [telnetd.h](#).

Data Fields

- char * [lines](#) [TELNETD_CONF_NUMLINES]
- char [buf](#) [TELNETD_CONF_LINELEN]
- char [bufptr](#)
- [u8_t](#) [numsent](#)
- [u8_t](#) [state](#)

7.11 timer Struct Reference

```
#include <timer.h>
```

7.11.1 Detailed Description

A timer.

This structure is used for declaring a timer. The timer must be set with [timer_set\(\)](#) before it can be used.

Examples:

[dhcpc.h](#), [example-mainloop-with-arp.c](#), [example-mainloop-without-arp.c](#), and [webclient.h](#).

Definition at line 74 of file [timer.h](#).

Data Fields

- clock_time_t [start](#)
- clock_time_t [interval](#)

7.12 uip_conn Struct Reference

```
#include <uip.h>
```

7.12.1 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 type of this field is configured in the "uipopt.h" header file.

Definition at line 1153 of file `uip.h`.

Data Fields

- `uip_ipaddr_t ripaddr`
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.
- [uip_tcp_appstate_t appstate](#)
The application state.

7.13 uip_eth_addr Struct Reference

```
#include <uip.h>
```

7.13.1 Detailed Description

Representation of a 48-bit Ethernet address.

Definition at line 1542 of file uip.h.

Data Fields

- [u8_t addr](#) [6]

7.14 uip_eth_hdr Struct Reference

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

7.14.1 Detailed Description

The Ethernet header.

Definition at line 63 of file uip_arp.h.

Data Fields

- [uip_eth_addr dest](#)
- [uip_eth_addr src](#)
- [u16_t type](#)

7.15 uip_icmpip_hdr Struct Reference

7.15.1 Detailed Description

Definition at line 1423 of file uip.h.

Data Fields

- [u8_t vhl](#)
- [u8_t tos](#)
- [u8_t len](#) [2]
- [u8_t ipid](#) [2]
- [u8_t ipoffset](#) [2]
- [u8_t ttl](#)
- [u8_t proto](#)
- [u16_t ipchksum](#)
- [u16_t srcipaddr](#) [2]
- [u16_t destipaddr](#) [2]
- [u8_t type](#)
- [u8_t icode](#)
- [u16_t icmpchksum](#)
- [u16_t id](#)
- [u16_t seqno](#)

7.16 uip_neighbor_addr Struct Reference

7.16.1 Detailed Description

Definition at line 47 of file uip-neighbor.h.

Data Fields

- [uip_eth_addr](#) addr

7.17 uip_stats Struct Reference

```
#include <uip.h>
```

7.17.1 Detailed Description

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

Definition at line 1232 of file uip.h.

Data Fields

- struct {
 - [uip_stats_t drop](#)
Number of dropped packets at the IP layer.
 - [uip_stats_t recv](#)
Number of received packets at the IP layer.
 - [uip_stats_t sent](#)
Number of sent packets at the IP layer.
 - [uip_stats_t vhlerr](#)
Number of packets dropped due to wrong IP version or header length.
 - [uip_stats_t hblenerr](#)
Number of packets dropped due to wrong IP length, high byte.
 - [uip_stats_t lblenerr](#)
Number of packets dropped due to wrong IP length, low byte.
 - [uip_stats_t fragerr](#)
Number of packets dropped since they were IP fragments.
 - [uip_stats_t chkerr](#)
Number of packets dropped due to IP checksum errors.
 - [uip_stats_t protoerr](#)
Number of packets dropped since they were neither ICMP, UDP nor TCP.
- } ip

IP statistics.
- struct {
 - [uip_stats_t drop](#)
Number of dropped ICMP packets.
 - [uip_stats_t recv](#)
Number of received ICMP packets.
 - [uip_stats_t sent](#)
Number of sent ICMP packets.
 - [uip_stats_t typeerr](#)
Number of ICMP packets with a wrong type.
- } icmp

ICMP statistics.
- struct {
 - [uip_stats_t drop](#)
Number of dropped TCP segments.
 - [uip_stats_t recv](#)

Number of received TCP segments.
[uip_stats_t sent](#)
Number of sent TCP segments.
[uip_stats_t chkerr](#)
Number of TCP segments with a bad checksum.
[uip_stats_t ackerr](#)
Number of TCP segments with a bad ACK number.
[uip_stats_t rst](#)
Number of received TCP RST (reset) segments.
[uip_stats_t rexmit](#)
Number of retransmitted TCP segments.
[uip_stats_t syndrop](#)
Number of dropped SYNs due to too few connections was available.
[uip_stats_t synrst](#)
Number of SYNs for closed ports, triggering a RST.
} [tcp](#)

TCP statistics.

- struct {
 [uip_stats_t drop](#)
 Number of dropped UDP segments.
 [uip_stats_t recv](#)
 Number of received UDP segments.
 [uip_stats_t sent](#)
 Number of sent UDP segments.
 [uip_stats_t chkerr](#)
 Number of UDP segments with a bad checksum.
} [udp](#)

UDP statistics.

7.18 uip_tcpip_hdr Struct Reference

7.18.1 Detailed Description

Definition at line 1386 of file uip.h.

Data Fields

- [u8_t vhl](#)
- [u8_t tos](#)
- [u8_t len](#) [2]
- [u8_t ipid](#) [2]
- [u8_t ipoffset](#) [2]
- [u8_t ttl](#)
- [u8_t proto](#)
- [u16_t ipchksum](#)
- [u16_t srcipaddr](#) [2]
- [u16_t destipaddr](#) [2]
- [u16_t srcport](#)
- [u16_t destport](#)
- [u8_t seqno](#) [4]
- [u8_t ackno](#) [4]
- [u8_t tcpoffset](#)
- [u8_t flags](#)
- [u8_t wnd](#) [2]
- [u16_t tcpchksum](#)
- [u8_t urgp](#) [2]
- [u8_t optdata](#) [4]

7.19 uip_udp_conn Struct Reference

```
#include <uip.h>
```

7.19.1 Detailed Description

Representation of a uIP UDP connection.

Examples:

[dhcpc.h](#), and [resolv.c](#).

Definition at line 1210 of file uip.h.

Data Fields

- [uip_ipaddr_t ripaddr](#)
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.
- [u8_t ttl](#)
Default time-to-live.
- [uip_udp_appstate_t appstate](#)
The application state.

7.20 uip_udpip_hdr Struct Reference

7.20.1 Detailed Description

Definition at line 1460 of file uip.h.

Data Fields

- [u8_t vhl](#)
- [u8_t tos](#)
- [u8_t len](#) [2]
- [u8_t ipid](#) [2]
- [u8_t ipoffset](#) [2]
- [u8_t ttl](#)
- [u8_t proto](#)
- [u16_t ipchksum](#)
- [u16_t srcipaddr](#) [2]
- [u16_t destipaddr](#) [2]
- [u16_t srcport](#)
- [u16_t destport](#)
- [u16_t udplen](#)
- [u16_t udpchksum](#)

7.21 webclient_state Struct Reference

7.21.1 Detailed Description

Examples:

[webclient.c](#), and [webclient.h](#).

Definition at line 55 of file [webclient.h](#).

Data Fields

- [u8_t](#) timer
- [u8_t](#) state
- [u8_t](#) httpflag
- [u16_t](#) port
- char [host](#) [40]
- char [file](#) [WEBCLIENT_CONF_MAX_URLLEN]
- [u16_t](#) getrequestptr
- [u16_t](#) getrequestleft
- char [httpheaderline](#) [200]
- [u16_t](#) httpheaderlineptr
- char [mimetype](#) [32]

Chapter 8

uIP 1.0 File Documentation

8.1 apps/hello-world/hello-world.c File Reference

8.1.1 Detailed Description

An example of how to write uIP applications with protosockets.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [hello-world.c](#).

```
#include "hello-world.h"
#include "uip.h"
#include <string.h>
```

Functions

- void [hello_world_init](#) (void)
- void [hello_world_appcall](#) (void)

8.2 apps/hello-world/hello-world.h File Reference

8.2.1 Detailed Description

Header file for an example of how to write uIP applications with protosockets.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [hello-world.h](#).

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

Data Structures

- struct [hello_world_state](#)

Defines

- #define [UIP_APPCALL](#) hello_world_appcall

Functions

- void [hello_world_appcall](#) (void)
- void [hello_world_init](#) (void)

8.3 apps/resolv/resolv.c File Reference

8.3.1 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.

Definition in file [resolv.c](#).

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

Defines

- `#define NULL` (void *)0
- `#define MAX_RETRIES` 8
- `#define DNS_FLAG1_RESPONSE` 0x80
- `#define DNS_FLAG1_OPCODE_STATUS` 0x10
- `#define DNS_FLAG1_OPCODE_INVERSE` 0x08
- `#define DNS_FLAG1_OPCODE_STANDARD` 0x00
- `#define DNS_FLAG1_AUTHORATIVE` 0x04
- `#define DNS_FLAG1_TRUNC` 0x02
- `#define DNS_FLAG1_RD` 0x01
- `#define DNS_FLAG2_RA` 0x80
- `#define DNS_FLAG2_ERR_MASK` 0x0f
- `#define DNS_FLAG2_ERR_NONE` 0x00
- `#define DNS_FLAG2_ERR_NAME` 0x03
- `#define STATE_UNUSED` 0
- `#define STATE_NEW` 1
- `#define STATE Asking` 2
- `#define STATE_DONE` 3
- `#define STATE_ERROR` 4
- `#define RESOLV_ENTRIES` 4

Functions

- void [resolv_appcall](#) (void)
- 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 `resolve_conf` (`u16_t *dnsserver`)

Configure which DNS server to use for queries.

- void `resolve_init` (void)

Initialize the resolver.

8.4 apps/resolv/resolv.h File Reference

8.4.1 Detailed Description

DNS resolver code header file.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [resolv.h](#).

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

Appication specific configurations

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 type of the application structure by typedef'ing the type `uip_tcp_appstate_t` and `uip_udp_appstate_t`.

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;
};
typedef struct httpd_state uip_tcp_appstate_t
```

- typedef int [uip_udp_appstate_t](#)

The type of the application state that is to be stored in the [uip_conn](#) structure.

Defines

- #define [UIP_UDP_APPCALL](#) `resolv_appcall`

Functions

- void [resolv_appcall](#) (void)
- 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)`
Initialize 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.

8.5 apps/smtp/smtp.c File Reference

8.5.1 Detailed Description

SMTP example implementation.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [smtp.c](#).

```
#include "smtp.h"
#include "smtp-strings.h"
#include "psock.h"
#include "uip.h"
#include <string.h>
```

Defines

- #define [ISO_nl](#) 0x0a
- #define [ISO_cr](#) 0x0d
- #define [ISO_period](#) 0x2e
- #define [ISO_2](#) 0x32
- #define [ISO_3](#) 0x33
- #define [ISO_4](#) 0x34
- #define [ISO_5](#) 0x35

Functions

- void [smtp_appcall](#) (void)
- void [smtp_configure](#) (char *lhostname, void *server)
Specificy an SMTP server and hostname.
- unsigned char [smtp_send](#) (char *to, char *cc, char *from, char *subject, char *msg, [u16_t](#) msglen)
Send an e-mail.
- void [smtp_init](#) (void)

8.6 apps/smtp/smtp.h File Reference

8.6.1 Detailed Description

SMTP header file.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [smtp.h](#).

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

Data Structures

- struct [smtp_state](#)

Appication specific configurations

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 type of the application structure by typedef'ing the type [uip_tcp_appstate_t](#) and [uip_udp_appstate_t](#).

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;
};
typedef struct httpd_state uip_tcp_appstate_t
```

- `#define UIP_APPCALL smtp_appcall`
The name of the application function that uIP should call in response to TCP/IP events.
- `typedef smtp_state uip_tcp_appstate_t`
The type of the application state that is to be stored in the [uip_conn](#) structure.

Defines

- `#define SMTP_ERR_OK 0`
Error number that signifies a non-error condition.
- `#define SMTP_SEND(to, cc, from, subject, msg) smtp_send(to, cc, from, subject, msg, strlen(msg))`

Functions

- void [smtp_done](#) (unsigned char error)
Callback function that is called when an e-mail transmission is done.
- void [smtp_init](#) (void)
- void [smtp_appcall](#) (void)

8.7 apps/telnetd/shell.c File Reference

8.7.1 Detailed Description

Simple shell.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [shell.c](#).

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

Defines

- #define [SHELL_PROMPT](#) "uIP 1.0> "

Functions

- void [shell_init](#) (void)
Initialize the shell.
- void [shell_start](#) (void)
Start the shell back-end.
- void [shell_input](#) (char *command)
Process a shell command.

8.8 apps/telnetd/shell.h File Reference

8.8.1 Detailed Description

Simple shell, header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [shell.h](#).

Functions

- void [shell_init](#) (void)
Initialize the shell.
- void [shell_start](#) (void)
Start the shell back-end.
- void [shell_input](#) (char *command)
Process a shell command.
- void [shell_quit](#) (char *)
Quit the shell.
- void [shell_output](#) (char *str1, char *str2)
Print a string to the shell window.
- void [shell_prompt](#) (char *prompt)
Print a prompt to the shell window.

8.9 apps/telnetd/telnetd.c File Reference

8.9.1 Detailed Description

Shell server.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [telnetd.c](#).

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

Defines

- #define [ISO_nl](#) 0x0a
- #define [ISO_cr](#) 0x0d
- #define [STATE_NORMAL](#) 0
- #define [STATE_IAC](#) 1
- #define [STATE_WILL](#) 2
- #define [STATE_WONT](#) 3
- #define [STATE_DO](#) 4
- #define [STATE_DONT](#) 5
- #define [STATE_CLOSE](#) 6
- #define [TELNET_IAC](#) 255
- #define [TELNET_WILL](#) 251
- #define [TELNET_WONT](#) 252
- #define [TELNET_DO](#) 253
- #define [TELNET_DONT](#) 254

Functions

- void [shell_quit](#) (char *)
Quit the shell.
- void [shell_prompt](#) (char *prompt)
Print a prompt to the shell window.
- void [shell_output](#) (char *str1, char *str2)
Print a string to the shell window.
- void [telnetd_init](#) (void)
- void [telnetd_appcall](#) (void)

8.10 apps/telnetd/telnetd.h File Reference

8.10.1 Detailed Description

Shell server.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [telnetd.h](#).

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

Data Structures

- struct [telnetd_state](#)

Defines

- #define [TELNETD_CONF_LINELEN](#) 40
- #define [TELNETD_CONF_NUMLINES](#) 16
- #define [UIP_APPCALL](#) telnetd_appcall

Typedefs

- typedef [telnetd_state](#) uip_tcp_appstate_t

Functions

- void [telnetd_appcall](#) (void)

8.11 apps/webclient/webclient.c File Reference

8.11.1 Detailed Description

Implementation of the HTTP client.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [webclient.c](#).

```
#include "uip.h"
#include "uiplib.h"
#include "webclient.h"
#include "resolv.h"
#include <string.h>
```

Defines

- #define [WEBCLIENT_TIMEOUT](#) 100
- #define [WEBCLIENT_STATE_STATUSLINE](#) 0
- #define [WEBCLIENT_STATE_HEADERS](#) 1
- #define [WEBCLIENT_STATE_DATA](#) 2
- #define [WEBCLIENT_STATE_CLOSE](#) 3
- #define [HTTPFLAG_NONE](#) 0
- #define [HTTPFLAG_OK](#) 1
- #define [HTTPFLAG_MOVED](#) 2
- #define [HTTPFLAG_ERROR](#) 3
- #define [ISO_nl](#) 0x0a
- #define [ISO_cr](#) 0x0d
- #define [ISO_space](#) 0x20

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.
- void `webclient_appcall` (void)

8.12 apps/webclient/webclient.h File Reference

8.12.1 Detailed Description

Header file for the HTTP client.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [webclient.h](#).

```
#include "webclient-strings.h"
#include "uipopt.h"
```

Data Structures

- struct [webclient_state](#)

Defines

- #define [WEBCLIENT_CONF_MAX_URLLEN](#) 100
- #define [UIP_APPCALL](#) webclient_appcall

Typedefs

- typedef [webclient_state](#) [uip_tcp_appstate_t](#)

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.
- void [webclient_appcall](#) (void)
- 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.

8.13 apps/webserver/httpd-cgi.c File Reference

8.13.1 Detailed Description

Web server script interface.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [httpd-cgi.c](#).

```
#include "uip.h"
#include "psock.h"
#include "httpd.h"
#include "httpd-cgi.h"
#include "httpd-fs.h"
#include <stdio.h>
#include <string.h>
```

Functions

- `httpd_cgifunction` [httpd_cgi](#) (char *name)

8.14 apps/webserver/httpd-cgi.h File Reference

8.14.1 Detailed Description

Web server script interface header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [httpd-cgi.h](#).

```
#include "psock.h"
#include "httpd.h"
```

Data Structures

- struct [httpd_cgi_call](#)

Defines

- #define [HTTPD_CGI_CALL](#)(name, str, function)
HTTPD CGI function declaration.

Functions

- httpd_cgifunction [httpd_cgi](#) (char *name)

8.15 apps/webserver/httpd.c File Reference

8.15.1 Detailed Description

Web server.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [httpd.c](#).

```
#include "uip.h"
#include "httpd.h"
#include "httpd-fs.h"
#include "httpd-cgi.h"
#include "http-strings.h"
#include <string.h>
```

Defines

- #define [STATE_WAITING](#) 0
- #define [STATE_OUTPUT](#) 1
- #define [ISO_nl](#) 0x0a
- #define [ISO_space](#) 0x20
- #define [ISO_bang](#) 0x21
- #define [ISO_percent](#) 0x25
- #define [ISO_period](#) 0x2e
- #define [ISO_slash](#) 0x2f
- #define [ISO_colon](#) 0x3a

Functions

- void [httpd_appcall](#) (void)
- void [httpd_init](#) (void)

Initialize the web server.

8.16 lib/memb.c File Reference

8.16.1 Detailed Description

Memory block allocation routines.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [memb.c](#).

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

Functions

- void [memb_init](#) (struct [memb_blocks](#) *m)
Initialize a memory block that was declared with [MEMB\(\)](#).
- void * [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, void *ptr)
Deallocate a memory block from a memory block previously declared with [MEMB\(\)](#).

8.17 lib/memb.h File Reference

8.17.1 Detailed Description

Memory block allocation routines.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [memb.h](#).

Data Structures

- struct [memb_blocks](#)

Defines

- #define [MEMB_CONCAT2](#)(s1, s2) s1##s2
- #define [MEMB_CONCAT](#)(s1, s2) [MEMB_CONCAT2](#)(s1, s2)
- #define [MEMB](#)(name, structure, num)

Declare a memory block.

Functions

- void [memb_init](#) (struct [memb_blocks](#) *m)
Initialize a memory block that was declared with [MEMB](#)().
- void * [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, void *ptr)
Deallocate a memory block from a memory block previously declared with [MEMB](#)().

8.18 uip/lc-addrlabels.h File Reference

8.18.1 Detailed Description

Implementation of local continuations based on the "Labels as values" feature of gcc.

Author:

Adam Dunkels <adam@sics.se>

This implementation of local continuations is based on a special feature of the GCC C compiler called "labels as values". This feature allows assigning pointers with the address of the code corresponding to a particular C label.

For more information, see the GCC documentation: <http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values>

Thanks to dividium for finding the nice local scope label implementation.

Definition in file [lc-addrlabels.h](#).

Defines

- #define [LC_INIT](#)(s) s = NULL
- #define [LC_RESUME](#)(s)
- #define [LC_SET](#)(s) do { ({ __label__ resume; resume: (s) = &&resume; }); } while(0)
- #define [LC_END](#)(s)

Typedefs

- typedef void * [lc_t](#)

8.19 uip/lc-switch.h File Reference

8.19.1 Detailed Description

Implementation of local continuations based on switch() statment.

Author:

Adam Dunkels <adam@sics.se>

This implementation of local continuations uses the C switch() statement to resume execution of a function somewhere inside the function's body. The implementation is based on the fact that switch() statements are able to jump directly into the bodies of control structures such as if() or while() statmenets.

This implementation borrows heavily from Simon Tatham's coroutines implementation in C: <http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>

Definition in file [lc-switch.h](#).

Defines

- #define [__LC_SWTICH_H__](#)
- #define [LC_INIT](#)(s) s = 0;
- #define [LC_RESUME](#)(s) switch(s) { case 0:
- #define [LC_SET](#)(s) s = __LINE__; case __LINE__:
- #define [LC_END](#)(s) }

Typedefs

- typedef unsigned short [lc_t](#)

8.20 uip/lc.h File Reference

8.20.1 Detailed Description

Local continuations.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [lc.h](#).

```
#include "lc-switch.h"
```

8.21 uip/psock.h File Reference

8.21.1 Detailed Description

Protosocket library header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [psock.h](#).

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

Data Structures

- struct [psock_buf](#)
- struct [psock](#)

The representation of a protosocket.

Defines

- #define [PSOCK_INIT](#)(psock, buffer, buffersize)
Initialize a protosocket.
- #define [PSOCK_BEGIN](#)(psock)
Start the protosocket protothread in a function.
- #define [PSOCK_SEND](#)(psock, data, datalen)
Send data.
- #define [PSOCK_SEND_STR](#)(psock, str)
Send a null-terminated string.
- #define [PSOCK_GENERATOR_SEND](#)(psock, generator, arg)
Generate data with a function and send it.
- #define [PSOCK_CLOSE](#)(psock)
Close a protosocket.
- #define [PSOCK_READBUF](#)(psock)
Read data until the buffer is full.
- #define [PSOCK_READTO](#)(psock, c)
Read data up to a specified character.
- #define [PSOCK_DATALEN](#)(psock)
The length of the data that was previously read.

- #define `PSOCK_EXIT(psock)`
Exit the protosocket's protothread.
- #define `PSOCK_CLOSE_EXIT(psock)`
Close a protosocket and exit the protosocket's protothread.
- #define `PSOCK_END(psock)`
Declare the end of a protosocket's protothread.
- #define `PSOCK_NEWDATA(psock)`
Check if new data has arrived on a protosocket.
- #define `PSOCK_WAIT_UNTIL(psock, condition)`
Wait until a condition is true.
- #define `PSOCK_WAIT_THREAD(psock, condition) PT_WAIT_THREAD(&((psock) → pt), (condition))`

Functions

- `u16_t psock_datalen` (struct `psock` *`psock`)
- `char psock_newdata` (struct `psock` *`s`)

8.22 uip/pt.h File Reference

8.22.1 Detailed Description

Protothreads implementation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [pt.h](#).

```
#include "lc.h"
```

Data Structures

- struct [pt](#)

Initialization

- #define [PT_INIT](#)(pt)
Initialize a protothread.

Declaration and definition

- #define [PT_THREAD](#)(name_args)
Declaration of a protothread.
- #define [PT_BEGIN](#)(pt)
Declare the start of a protothread inside the C function implementing the protothread.
- #define [PT_END](#)(pt)
Declare the end of a protothread.

Blocked wait

- #define [PT_WAIT_UNTIL](#)(pt, condition)
Block and wait until condition is true.
- #define [PT_WAIT_WHILE](#)(pt, cond)
Block and wait while condition is true.

Hierarchical protothreads

- #define [PT_WAIT_THREAD](#)(pt, thread)
Block and wait until a child protothread completes.
- #define [PT_SPAWN](#)(pt, child, thread)
Spawn a child protothread and wait until it exits.

Exiting and restarting

- #define [PT_RESTART](#)(pt)
Restart the protothread.
- #define [PT_EXIT](#)(pt)
Exit the protothread.

Calling a protothread

- #define [PT_SCHEDULE](#)(f)
Schedule a protothread.

Yielding from a protothread

- #define [PT_YIELD](#)(pt)
Yield from the current protothread.
- #define [PT_YIELD_UNTIL](#)(pt, cond)
Yield from the protothread until a condition occurs.

Defines

- #define [PT_WAITING](#) 0
- #define [PT_EXITED](#) 1
- #define [PT_ENDED](#) 2
- #define [PT_YIELDED](#) 3

8.23 uip/timer.c File Reference

8.23.1 Detailed Description

Timer library implementation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [timer.c](#).

```
#include "clock.h"  
#include "timer.h"
```

Functions

- void [timer_set](#) (struct [timer](#) *t, clock_time_t interval)
Set a timer.
- void [timer_reset](#) (struct [timer](#) *t)
Reset the timer with the same interval.
- void [timer_restart](#) (struct [timer](#) *t)
Restart the timer from the current point in time.
- int [timer_expired](#) (struct [timer](#) *t)
Check if a timer has expired.

8.24 uip/timer.h File Reference

8.24.1 Detailed Description

Timer library header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [timer.h](#).

```
#include "clock.h"
```

Data Structures

- struct [timer](#)

A timer.

Functions

- void [timer_set](#) (struct [timer](#) *t, clock_time_t interval)

Set a timer.

- void [timer_reset](#) (struct [timer](#) *t)

Reset the timer with the same interval.

- void [timer_restart](#) (struct [timer](#) *t)

Restart the timer from the current point in time.

- int [timer_expired](#) (struct [timer](#) *t)

Check if a timer has expired.

8.25 uip/uip-neighbor.c File Reference

8.25.1 Detailed Description

Database of link-local neighbors, used by IPv6 code and to be used by a future ARP code rewrite.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [uip-neighbor.c](#).

```
#include "uip-neighbor.h"
#include <string.h>
```

Defines

- #define [MAX_TIME](#) 128
- #define [ENTRIES](#) 8

Functions

- void [uip_neighbor_init](#) (void)
- void [uip_neighbor_periodic](#) (void)
- void [uip_neighbor_add](#) ([uip_ipaddr_t](#) ipaddr, struct [uip_neighbor_addr](#) *addr)
- void [uip_neighbor_update](#) ([uip_ipaddr_t](#) ipaddr)
- [uip_neighbor_addr](#) * [uip_neighbor_lookup](#) ([uip_ipaddr_t](#) ipaddr)

8.26 uip/uip-neighbor.h File Reference

8.26.1 Detailed Description

Header file for database of link-local neighbors, used by IPv6 code and to be used by future ARP code.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [uip-neighbor.h](#).

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

Data Structures

- struct [uip_neighbor_addr](#)

Functions

- void [uip_neighbor_init](#) (void)
- void [uip_neighbor_add](#) ([uip_ipaddr_t](#) ipaddr, struct [uip_neighbor_addr](#) *addr)
- void [uip_neighbor_update](#) ([uip_ipaddr_t](#) ipaddr)
- [uip_neighbor_addr](#) * [uip_neighbor_lookup](#) ([uip_ipaddr_t](#) ipaddr)
- void [uip_neighbor_periodic](#) (void)

8.27 uip/uip-split.h File Reference

8.27.1 Detailed Description

Module for splitting outbound TCP segments in two to avoid the delayed ACK throughput degradation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [uip-split.h](#).

Functions

- void [uip_split_output](#) (void)
Handle outgoing packets.

8.28 uip/uip.c File Reference

8.28.1 Detailed Description

The uIP TCP/IP stack code.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [uip.c](#).

```
#include "uip.h"
#include "uipopt.h"
#include "uip_arch.h"
#include <string.h>
```

Defines

- #define [DEBUG_PRINTF\(\)](#)
- #define [TCP_FIN](#) 0x01
- #define [TCP_SYN](#) 0x02
- #define [TCP_RST](#) 0x04
- #define [TCP_PSH](#) 0x08
- #define [TCP_ACK](#) 0x10
- #define [TCP_URG](#) 0x20
- #define [TCP_CTL](#) 0x3f
- #define [TCP_OPT_END](#) 0
- #define [TCP_OPT_NOOP](#) 1
- #define [TCP_OPT_MSS](#) 2
- #define [TCP_OPT_MSS_LEN](#) 4
- #define [ICMP_ECHO_REPLY](#) 0
- #define [ICMP_ECHO](#) 8
- #define [ICMP6_ECHO_REPLY](#) 129
- #define [ICMP6_ECHO](#) 128
- #define [ICMP6_NEIGHBOR_SOLICITATION](#) 135
- #define [ICMP6_NEIGHBOR_ADVERTISEMENT](#) 136
- #define [ICMP6_FLAG_S](#) (1 << 6)
- #define [ICMP6_OPTION_SOURCE_LINK_ADDRESS](#) 1
- #define [ICMP6_OPTION_TARGET_LINK_ADDRESS](#) 2
- #define [BUF](#) ((struct [uip_tcpip_hdr](#) *)&[uip_buf](#)[UIP_LLH_LEN])
- #define [FBUF](#) ((struct [uip_tcpip_hdr](#) *)&[uip_reassbuf](#)[0])
- #define [ICMPBUF](#) ((struct [uip_icmpip_hdr](#) *)&[uip_buf](#)[UIP_LLH_LEN])
- #define [UDPBUF](#) ((struct [uip_udpip_hdr](#) *)&[uip_buf](#)[UIP_LLH_LEN])
- #define [UIP_STAT](#)(s)
- #define [UIP_LOG](#)(m)

Functions

- void `uip_setipid` (`u16_t` id)
uIP initialization function.
- 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`.
- void `uip_init` (void)
uIP initialization function.
- `uip_conn` * `uip_connect` (`uip_ipaddr_t` *ripaddr, `u16_t` rport)
Connect to a remote host using TCP.
- `uip_udp_conn` * `uip_udp_new` (`uip_ipaddr_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.
- void `uip_process` (`u8_t` flag)
- `u16_t` `htons` (`u16_t` val)
Convert 16-bit quantity from host byte order to network byte order.
- void `uip_send` (const void *data, int len)
Send data on the current connection.

Variables

- `uip_ipaddr_t` `uip_hostaddr`
- `uip_ipaddr_t` `uip_draddr`
- `uip_ipaddr_t` `uip_netmask`
- `uip_eth_addr` `uip_ethaddr` = { {0,0,0,0,0,0} }
- `u8_t` `uip_buf` [UIP_BUFSIZE+2]
The uIP packet buffer.
- void * `uip_appdata`

Pointer to the application data in the packet buffer.

- void * [uip_sappdata](#)
- [u16_t uip_len](#)

The length of the packet in the uip_buf buffer.

- [u16_t uip_slen](#)
- [u8_t uip_flags](#)
- [uip_conn * uip_conn](#)

Pointer to the current TCP connection.

- [uip_conn uip_conns](#) [UIP_CONNS]
- [u16_t uip_listenports](#) [UIP_LISTENPORTS]
- [uip_udp_conn * uip_udp_conn](#)

The current UDP connection.

- [uip_udp_conn uip_udp_conns](#) [UIP_UDP_CONNS]
- [u8_t uip_acc32](#) [4]

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

8.29 uip/uip.h File Reference

8.29.1 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.

Definition in file [uip.h](#).

```
#include "uiptopt.h"
```

Data Structures

- struct [uip_conn](#)
Representation of a uIP TCP connection.
- struct [uip_udp_conn](#)
Representation of a uIP UDP connection.
- struct [uip_stats](#)
The structure holding the TCP/IP statistics that are gathered if UIP_STATISTICS is set to 1.
- struct [uip_tcpip_hdr](#)
- struct [uip_icmpip_hdr](#)
- struct [uip_udpip_hdr](#)
- struct [uip_eth_addr](#)
Representation of a 48-bit Ethernet address.

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_input()`
Process an incoming packet.
- #define `uip_periodic(conn)`
Periodic processing for a connection identified by its number.
- #define `uip_conn_active(conn)` (`uip_conns[conn].tcpstateflags != UIP_CLOSED`)
- #define `uip_periodic_conn(conn)`
Perform periodic processing for a connection identified by a pointer to its structure.
- #define `uip_poll_conn(conn)`
Reuqest that a particular connection should be polled.
- #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_outstanding(conn)` (`((conn) → len)`)
- #define `uip_datalen()`
The length of any incoming data that is currently avaiable (if avaiable) 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_udpconnection()`
Is the current connection a UDP connection?
- #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 maximum segment size (MSS) of the current connection.
- `#define uip_mss()`
Get the current maximum segment size that can be sent on the current connection.
- `#define uip_udp_remove(conn)`
Removed a UDP connection.
- `#define uip_udp_bind(conn, port)`
Bind a UDP connection to a local port.
- `#define uip_udp_send(len)`
Send a UDP datagram of length len on the current connection.
- `#define uip_ipaddr(addr, addr0, addr1, addr2, addr3)`
Construct an IP address from four bytes.
- `#define uip_ip6addr(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7)`
Construct an IPv6 address from eight 16-bit words.
- `#define uip_ipaddr_copy(dest, src)`
Copy an IP address to another IP address.
- `#define uip_ipaddr_cmp(addr1, addr2)`
Compare two IP addresses.
- `#define uip_ipaddr_maskcmp(addr1, addr2, mask)`

Compare two IP addresses with netmasks.

- #define `uip_ipaddr_mask`(dest, src, mask)
Mask out the network part of an IP address.
- #define `uip_ipaddr1`(addr)
Pick the first octet of an IP address.
- #define `uip_ipaddr2`(addr)
Pick the second octet of an IP address.
- #define `uip_ipaddr3`(addr)
Pick the third octet of an IP address.
- #define `uip_ipaddr4`(addr)
Pick the fourth octet of an IP address.
- #define `HTONS`(n)
Convert 16-bit quantity from host byte order to network byte order.
- #define `ntohs` htons
- #define `UIP_ACKDATA` 1
- #define `UIP_NEWDATA` 2
- #define `UIP_REXMIT` 4
- #define `UIP_POLL` 8
- #define `UIP_CLOSE` 16
- #define `UIP_ABORT` 32
- #define `UIP_CONNECTED` 64
- #define `UIP_TIMEDOUT` 128
- #define `UIP_DATA` 1
- #define `UIP_TIMER` 2
- #define `UIP_POLL_REQUEST` 3
- #define `UIP_UDP_SEND_CONN` 4
- #define `UIP_UDP_TIMER` 5
- #define `UIP_CLOSED` 0
- #define `UIP_SYN_RCVD` 1
- #define `UIP_SYN_SENT` 2
- #define `UIP_ESTABLISHED` 3
- #define `UIP_FIN_WAIT_1` 4
- #define `UIP_FIN_WAIT_2` 5
- #define `UIP_CLOSING` 6
- #define `UIP_TIME_WAIT` 7
- #define `UIP_LAST_ACK` 8
- #define `UIP_TS_MASK` 15
- #define `UIP_STOPPED` 16
- #define `UIP_APPDATA_SIZE`
The buffer size available for user data in the `uip_buf` buffer.
- #define `UIP_PROTO_ICMP` 1
- #define `UIP_PROTO_TCP` 6

- `#define UIP_PROTO_UDP 17`
- `#define UIP_PROTO_ICMP6 58`
- `#define UIP_IPH_LEN 20`
- `#define UIP_UDPH_LEN 8`
- `#define UIP_TCPH_LEN 20`
- `#define UIP_IPUDPH_LEN (UIP_UDPH_LEN + UIP_IPH_LEN)`
- `#define UIP_IPTCPH_LEN (UIP_TCPH_LEN + UIP_IPH_LEN)`
- `#define UIP_TCPIP_HLEN UIP_IPTCPH_LEN`

Typedefs

- `typedef u16_t uip_ip4addr_t [2]`
Representation of an IP address.
- `typedef u16_t uip_ip6addr_t [8]`
- `typedef uip_ip4addr_t uip_ipaddr_t`

Functions

- `void uip_init (void)`
uIP initialization function.
- `void uip_setipid (u16_t id)`
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 (uip_ipaddr_t *ripaddr, u16_t port)`
Connect to a remote host using TCP.
- `void uip_send (const void *data, int len)`
Send data on the current connection.
- `uip_udp_conn * uip_udp_new (uip_ipaddr_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.
- `void uip_process (u8_t flag)`
- `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.
- [u16_t uip_udpchksum](#) (void)
Calculate the UDP checksum of the packet in uip_buf and uip_appdata.

Variables

- [u8_t uip_buf](#) [UIP_BUFSIZE+2]
The uIP packet buffer.
- [void * uip_appdata](#)
Pointer to the application data in the packet buffer.
- [u16_t uip_len](#)
The length of the packet in the uip_buf buffer.
- [uip_conn * uip_conn](#)
Pointer to the current TCP connection.
- [uip_conn uip_conns](#) [UIP_CONNS]
- [u8_t uip_acc32](#) [4]
4-byte array used for the 32-bit sequence number calculations.
- [uip_udp_conn * uip_udp_conn](#)
The current UDP connection.
- [uip_udp_conn uip_udp_conns](#) [UIP_UDP_CONNS]
- [uip_stats uip_stat](#)
The uIP TCP/IP statistics.
- [u8_t uip_flags](#)
- [uip_ipaddr_t uip_hostaddr](#)
- [uip_ipaddr_t uip_netmask](#)
- [uip_ipaddr_t uip_draddr](#)

8.30 uip/uip_arch.h File Reference

8.30.1 Detailed Description

Declarations of architecture specific functions.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [uip_arch.h](#).

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

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.

8.31 uip/uiplib_arp.c File Reference

8.31.1 Detailed Description

Implementation of the ARP Address Resolution Protocol.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [uip_arp.c](#).

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

Defines

- #define [ARP_REQUEST](#) 1
- #define [ARP_REPLY](#) 2
- #define [ARP_HWTYPE_ETH](#) 1
- #define [BUF](#) ((struct arp_hdr *)&[uip_buf](#)[0])
- #define [IPBUF](#) ((struct ethip_hdr *)&[uip_buf](#)[0])

Functions

- void [uip_arp_init](#) (void)
Initialize the ARP module.
- void [uip_arp_timer](#) (void)
Periodic ARP processing function.
- 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.

8.32 uip/uip_arp.h File Reference

8.32.1 Detailed Description

Macros and definitions for the ARP module.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file [uip_arp.h](#).

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

Data Structures

- struct [uip_eth_hdr](#)
The Ethernet header.

Defines

- #define [UIP_ETHTYPE_ARP](#) 0x0806
- #define [UIP_ETHTYPE_IP](#) 0x0800
- #define [UIP_ETHTYPE_IP6](#) 0x86dd
- #define [uip_arp_ipin](#)()
- #define [uip_setethaddr](#)(eaddr)
Specify the Ethernet MAC address.

Functions

- void [uip_arp_init](#) (void)
Initialize the ARP module.
- 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.

Variables

- [uip_eth_addr](#) [uip_ethaddr](#)

8.33 uip/uiop.h File Reference

8.33.1 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 "uiop.h" file that comes with the uIP distribution.

Definition in file [uiop.h](#).

```
#include "uip-conf.h"
```

Static configuration options

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 applicable only if uIP should be run over Ethernet.

All of these should be changed to suit your project.

- #define [UIP_FIXEDADDR](#)

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

- #define [UIP_PINGADDRCONF](#)

Ping IP address assignment.

- #define [UIP_FIXEDETHADDR](#)

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

IP configuration options

- #define [UIP_TTL](#) 64

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.

UDP configuration options

- #define `UIP_UDP`
Toggles whether 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.

TCP configuration options

- #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_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` 5
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 - UIP_TCPIP_HLEN)
The TCP maximum segment size.
- #define `UIP_RECEIVE_WINDOW`
The size of the advertised receiver's window.
- #define `UIP_TIME_WAIT_TIMEOUT` 120
How long a connection should stay in the TIME_WAIT state.

ARP configuration options

- #define [UIP_ARPTAB_SIZE](#)
The size of the ARP table.
- #define [UIP_ARP_MAXAGE](#) 120
The maximum age of ARP table entries measured in 10ths of seconds.

General configuration options

- #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_BROADCAST](#)
Broadcast support.
- #define [UIP_LLH_LEN](#)
The link level header length.
- void [uip_log](#) (char *msg)
Print out a uIP log message.

CPU architecture configuration

The CPU architecture configuration is where the endianness 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.

- #define [UIP_BYTE_ORDER](#)
The byte order of the CPU architecture on which uIP is to be run.

Defines

- #define [UIP_LITTLE_ENDIAN](#) 3412
- #define [UIP_BIG_ENDIAN](#) 1234

8.34 unix/uip-conf.h File Reference

8.34.1 Detailed Description

An example uIP configuration file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file [uip-conf.h](#).

```
#include <inttypes.h>
```

```
#include "webserver.h"
```

Project-specific configuration options

uIP has a number of configuration options that can be overridden for each project. These are kept in a project-specific uip-conf.h file and all configuration names have the prefix UIP_CONF.

- `#define UIP_CONF_MAX_CONNECTIONS`
Maximum number of TCP connections.
- `#define UIP_CONF_MAX_LISTENPORTS`
Maximum number of listening TCP ports.
- `#define UIP_CONF_BUFFER_SIZE`
uIP buffer size.
- `#define UIP_CONF_BYTE_ORDER`
CPU byte order.
- `#define UIP_CONF_LOGGING`
Logging on or off.
- `#define UIP_CONF_UDP`
UDP support on or off.
- `#define UIP_CONF_UDP_CHECKSUMS`
UDP checksums on or off.
- `#define UIP_CONF_STATISTICS`
uIP statistics on or off
- `typedef uint8_t u8_t`
8 bit datatype
- `typedef uint16_t u16_t`
16 bit datatype
- `typedef unsigned short uip_stats_t`
Statistics datatype.

Chapter 9

uIP 1.0 Example Documentation

9.1 dhcpc.c

```
1 /*
2  * Copyright (c) 2005, Swedish Institute of Computer Science
3  * All rights reserved.
4  *
5  * Redistribution and use in source and binary forms, with or without
6  * modification, are permitted provided that the following conditions
7  * are met:
8  * 1. Redistributions of source code must retain the above copyright
9  *    notice, this list of conditions and the following disclaimer.
10 * 2. Redistributions in binary form must reproduce the above copyright
11 *    notice, this list of conditions and the following disclaimer in the
12 *    documentation and/or other materials provided with the distribution.
13 * 3. Neither the name of the Institute nor the names of its contributors
14 *    may be used to endorse or promote products derived from this software
15 *    without specific prior written permission.
16 *
17 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
18 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
19 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
20 * ARE DISCLAIMED. IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
21 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
22 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
23 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
24 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
25 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
26 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
27 * SUCH DAMAGE.
28 *
29 * This file is part of the uIP TCP/IP stack
30 *
31 * @(#) $Id: dhcpc.c,v 1.2 2006/06/11 21:46:37 adam Exp $
32 */
33
34 #include <stdio.h>
35 #include <string.h>
36
37 #include "uip.h"
38 #include "dhcpc.h"
39 #include "timer.h"
40 #include "pt.h"
41
42 #define STATE_INITIAL          0
43 #define STATE_SENDING          1
44 #define STATE_OFFER_RECEIVED  2
```

```

45 #define STATE_CONFIG_RECEIVED 3
46
47 static struct dhcp_state s;
48
49 struct dhcp_msg {
50     u8_t op, htype, hlen, hops;
51     u8_t xid[4];
52     u16_t secs, flags;
53     u8_t ciaddr[4];
54     u8_t yiaddr[4];
55     u8_t siaddr[4];
56     u8_t giaddr[4];
57     u8_t chaddr[16];
58 #ifndef UIP_CONF_DHCP_LIGHT
59     u8_t sname[64];
60     u8_t file[128];
61 #endif
62     u8_t options[312];
63 };
64
65 #define BOOTP_BROADCAST 0x8000
66
67 #define DHCP_REQUEST      1
68 #define DHCP_REPLY       2
69 #define DHCP_HTYPE_ETHERNET 1
70 #define DHCP_HLEN_ETHERNET 6
71 #define DHCP_MSG_LEN      236
72
73 #define DHCP_SERVER_PORT  67
74 #define DHCP_CLIENT_PORT  68
75
76 #define DHCPDISCOVER  1
77 #define DHCPOFFER     2
78 #define DHCPREQUEST   3
79 #define DHCPDECLINE   4
80 #define DHCPACK        5
81 #define DHCPNAK        6
82 #define DHCPRELEASE   7
83
84 #define DHCP_OPTION_SUBNET_MASK  1
85 #define DHCP_OPTION_ROUTER       3
86 #define DHCP_OPTION_DNS_SERVER   6
87 #define DHCP_OPTION_REQ_IPADDR   50
88 #define DHCP_OPTION_LEASE_TIME   51
89 #define DHCP_OPTION_MSG_TYPE     53
90 #define DHCP_OPTION_SERVER_ID    54
91 #define DHCP_OPTION_REQ_LIST     55
92 #define DHCP_OPTION_END          255
93
94 static const u8_t xid[4] = {0xad, 0xde, 0x12, 0x23};
95 static const u8_t magic_cookie[4] = {99, 130, 83, 99};
96 /*-----*/
97 static u8_t *
98 add_msg_type(u8_t *optptr, u8_t type)
99 {
100     *optptr++ = DHCP_OPTION_MSG_TYPE;
101     *optptr++ = 1;
102     *optptr++ = type;
103     return optptr;
104 }
105 /*-----*/
106 static u8_t *
107 add_server_id(u8_t *optptr)
108 {
109     *optptr++ = DHCP_OPTION_SERVER_ID;
110     *optptr++ = 4;
111     memcpy(optptr, s.serverid, 4);

```

```

112     return optptr + 4;
113 }
114 /*-----*/
115 static u8_t *
116 add_req_ipaddr(u8_t *optptr)
117 {
118     *optptr++ = DHCP_OPTION_REQ_IPADDR;
119     *optptr++ = 4;
120     memcpy(optptr, s.ipaddr, 4);
121     return optptr + 4;
122 }
123 /*-----*/
124 static u8_t *
125 add_req_options(u8_t *optptr)
126 {
127     *optptr++ = DHCP_OPTION_REQ_LIST;
128     *optptr++ = 3;
129     *optptr++ = DHCP_OPTION_SUBNET_MASK;
130     *optptr++ = DHCP_OPTION_ROUTER;
131     *optptr++ = DHCP_OPTION_DNS_SERVER;
132     return optptr;
133 }
134 /*-----*/
135 static u8_t *
136 add_end(u8_t *optptr)
137 {
138     *optptr++ = DHCP_OPTION_END;
139     return optptr;
140 }
141 /*-----*/
142 static void
143 create_msg(register struct dhcp_msg *m)
144 {
145     m->op = DHCP_REQUEST;
146     m->htype = DHCP_HTYPE_ETHERNET;
147     m->hlen = s.mac_len;
148     m->hops = 0;
149     memcpy(m->xid, xid, sizeof(m->xid));
150     m->secs = 0;
151     m->flags = HTONS(BOOTP_BROADCAST); /* Broadcast bit. */
152     /* uip_ipaddr_copy(m->ciaddr, uip_hostaddr); */
153     memcpy(m->ciaddr, uip_hostaddr, sizeof(m->ciaddr));
154     memset(m->yiaddr, 0, sizeof(m->yiaddr));
155     memset(m->siaddr, 0, sizeof(m->siaddr));
156     memset(m->giaddr, 0, sizeof(m->giaddr));
157     memcpy(m->chaddr, s.mac_addr, s.mac_len);
158     memset(&m->chaddr[s.mac_len], 0, sizeof(m->chaddr) - s.mac_len);
159 #ifndef UIP_CONF_DHCP_LIGHT
160     memset(m->sname, 0, sizeof(m->sname));
161     memset(m->file, 0, sizeof(m->file));
162 #endif
163
164     memcpy(m->options, magic_cookie, sizeof(magic_cookie));
165 }
166 /*-----*/
167 static void
168 send_discover(void)
169 {
170     u8_t *end;
171     struct dhcp_msg *m = (struct dhcp_msg *)uip_appdata;
172
173     create_msg(m);
174
175     end = add_msg_type(&m->options[4], DHCPDISCOVER);
176     end = add_req_options(end);
177     end = add_end(end);
178 }

```

```

179 uip_send(uip_appdata, end - (u8_t *)uip_appdata);
180 }
181 /*-----*/
182 static void
183 send_request(void)
184 {
185     u8_t *end;
186     struct dhcp_msg *m = (struct dhcp_msg *)uip_appdata;
187
188     create_msg(m);
189
190     end = add_msg_type(&m->options[4], DHCPREQUEST);
191     end = add_server_id(end);
192     end = add_req_ipaddr(end);
193     end = add_end(end);
194
195     uip_send(uip_appdata, end - (u8_t *)uip_appdata);
196 }
197 /*-----*/
198 static u8_t
199 parse_options(u8_t *optptr, int len)
200 {
201     u8_t *end = optptr + len;
202     u8_t type = 0;
203
204     while(optptr < end) {
205         switch(*optptr) {
206             case DHCP_OPTION_SUBNET_MASK:
207                 memcpy(s.netmask, optptr + 2, 4);
208                 break;
209             case DHCP_OPTION_ROUTER:
210                 memcpy(s.default_router, optptr + 2, 4);
211                 break;
212             case DHCP_OPTION_DNS_SERVER:
213                 memcpy(s.dnsaddr, optptr + 2, 4);
214                 break;
215             case DHCP_OPTION_MSG_TYPE:
216                 type = *(optptr + 2);
217                 break;
218             case DHCP_OPTION_SERVER_ID:
219                 memcpy(s.serverid, optptr + 2, 4);
220                 break;
221             case DHCP_OPTION_LEASE_TIME:
222                 memcpy(s.lease_time, optptr + 2, 4);
223                 break;
224             case DHCP_OPTION_END:
225                 return type;
226         }
227         optptr += optptr[1] + 2;
228     }
229     return type;
230 }
231 }
232 /*-----*/
233 static u8_t
234 parse_msg(void)
235 {
236     struct dhcp_msg *m = (struct dhcp_msg *)uip_appdata;
237
238     if(m->op == DHCP_REPLY &&
239         memcmp(m->xid, xid, sizeof(xid)) == 0 &&
240         memcmp(m->chaddr, s.mac_addr, s.mac_len) == 0) {
241         memcpy(s.ipaddr, m->yiaddr, 4);
242         return parse_options(&m->options[4], uip_datalen());
243     }
244     return 0;
245 }

```

```

246 /*-----*/
247 static
248 PT_THREAD(handle_dhcp(void))
249 {
250     PT_BEGIN(&s.pt);
251
252     /* try_again:*/
253     s.state = STATE_SENDING;
254     s.ticks = CLOCK_SECOND;
255
256     do {
257         send_discover();
258         timer_set(&s.timer, s.ticks);
259         PT_WAIT_UNTIL(&s.pt, uip_newdata() || timer_expired(&s.timer));
260
261         if(uip_newdata() && parse_msg() == DHCPOFFER) {
262             s.state = STATE_OFFER_RECEIVED;
263             break;
264         }
265
266         if(s.ticks < CLOCK_SECOND * 60) {
267             s.ticks *= 2;
268         }
269     } while(s.state != STATE_OFFER_RECEIVED);
270
271     s.ticks = CLOCK_SECOND;
272
273     do {
274         send_request();
275         timer_set(&s.timer, s.ticks);
276         PT_WAIT_UNTIL(&s.pt, uip_newdata() || timer_expired(&s.timer));
277
278         if(uip_newdata() && parse_msg() == DHCPACK) {
279             s.state = STATE_CONFIG_RECEIVED;
280             break;
281         }
282
283         if(s.ticks <= CLOCK_SECOND * 10) {
284             s.ticks += CLOCK_SECOND;
285         } else {
286             PT_RESTART(&s.pt);
287         }
288     } while(s.state != STATE_CONFIG_RECEIVED);
289
290     #if 0
291     printf("Got IP address %d.%d.%d.%d\n",
292           uip_ipaddr1(s.ipaddr), uip_ipaddr2(s.ipaddr),
293           uip_ipaddr3(s.ipaddr), uip_ipaddr4(s.ipaddr));
294     printf("Got netmask %d.%d.%d.%d\n",
295           uip_ipaddr1(s.netmask), uip_ipaddr2(s.netmask),
296           uip_ipaddr3(s.netmask), uip_ipaddr4(s.netmask));
297     printf("Got DNS server %d.%d.%d.%d\n",
298           uip_ipaddr1(s.dnsaddr), uip_ipaddr2(s.dnsaddr),
299           uip_ipaddr3(s.dnsaddr), uip_ipaddr4(s.dnsaddr));
300     printf("Got default router %d.%d.%d.%d\n",
301           uip_ipaddr1(s.default_router), uip_ipaddr2(s.default_router),
302           uip_ipaddr3(s.default_router), uip_ipaddr4(s.default_router));
303     printf("Lease expires in %ld seconds\n",
304           ntohs(s.lease_time[0])*65536ul + ntohs(s.lease_time[1]));
305     #endif
306
307     dhcpc_configured(&s);
308
309     /* timer_stop(&s.timer);*/
310
311     /*
312     * PT_END restarts the thread so we do this instead. Eventually we

```

```
313     * should reacquire expired leases here.
314     */
315     while(1) {
316         PT_YIELD(&s.pt);
317     }
318
319     PT_END(&s.pt);
320 }
321 /*-----*/
322 void
323 dhcpc_init(const void *mac_addr, int mac_len)
324 {
325     uip_ipaddr_t addr;
326
327     s.mac_addr = mac_addr;
328     s.mac_len  = mac_len;
329
330     s.state = STATE_INITIAL;
331     uip_ipaddr(addr, 255,255,255,255);
332     s.conn = uip_udp_new(&addr, HTONS(DHCP_SERVER_PORT));
333     if(s.conn != NULL) {
334         uip_udp_bind(s.conn, HTONS(DHCP_CLIENT_PORT));
335     }
336     PT_INIT(&s.pt);
337 }
338 /*-----*/
339 void
340 dhcpc_appcall(void)
341 {
342     handle_dhcp();
343 }
344 /*-----*/
345 void
346 dhcpc_request(void)
347 {
348     u16_t ipaddr[2];
349
350     if(s.state == STATE_INITIAL) {
351         uip_ipaddr(ipaddr, 0,0,0,0);
352         uip_sethostaddr(ipaddr);
353         /* handle_dhcp(PROCESS_EVENT_NONE, NULL); */
354     }
355 }
356 /*-----*/
```


9.2 dhcpc.h

```

1 /*
2  * Copyright (c) 2005, Swedish Institute of Computer Science
3  * All rights reserved.
4  *
5  * Redistribution and use in source and binary forms, with or without
6  * modification, are permitted provided that the following conditions
7  * are met:
8  * 1. Redistributions of source code must retain the above copyright
9  *    notice, this list of conditions and the following disclaimer.
10 * 2. Redistributions in binary form must reproduce the above copyright
11 *    notice, this list of conditions and the following disclaimer in the
12 *    documentation and/or other materials provided with the distribution.
13 * 3. Neither the name of the Institute nor the names of its contributors
14 *    may be used to endorse or promote products derived from this software
15 *    without specific prior written permission.
16 *
17 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
18 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
19 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
20 * ARE DISCLAIMED. IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
21 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
22 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
23 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
24 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
25 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
26 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
27 * SUCH DAMAGE.
28 *
29 * This file is part of the uIP TCP/IP stack
30 *
31 * @(#) $Id: dhcpc.h,v 1.3 2006/06/11 21:46:37 adam Exp $
32 */
33 #ifndef __DHCP_H__
34 #define __DHCP_H__
35
36 #include "timer.h"
37 #include "pt.h"
38
39 struct dhcpc_state {
40     struct pt pt;
41     char state;
42     struct uip_udp_conn *conn;
43     struct timer timer;
44     u16_t ticks;
45     const void *mac_addr;
46     int mac_len;
47
48     u8_t serverid[4];
49
50     u16_t lease_time[2];
51     u16_t ipaddr[2];
52     u16_t netmask[2];
53     u16_t dnsaddr[2];
54     u16_t default_router[2];
55 };
56
57 void dhcpc_init(const void *mac_addr, int mac_len);
58 void dhcpc_request(void);
59
60 void dhcpc_appcall(void);
61
62 void dhcpc_configured(const struct dhcpc_state *s);
63
64 typedef struct dhcpc_state uip_udp_appstate_t;
65 #define UIP_UDP_APPCALL dhcpc_appcall

```

```
66  
67  
68 #endif /* __DHCP_H__ */
```

9.3 example-mainloop-with-arp.c

```

1 #include "uip.h"
2 #include "uip_arp.h"
3 #include "network-device.h"
4 #include "httpd.h"
5 #include "timer.h"
6
7 #define BUF ((struct uip_eth_hdr *)&uip_buf[0])
8
9 /*-----*/
10 int
11 main(void)
12 {
13     int i;
14     uip_ipaddr_t ipaddr;
15     struct timer periodic_timer, arp_timer;
16
17     timer_set(&periodic_timer, CLOCK_SECOND / 2);
18     timer_set(&arp_timer, CLOCK_SECOND * 10);
19
20     network_device_init();
21     uip_init();
22
23     uip_ipaddr(ipaddr, 192,168,0,2);
24     uip_sethostaddr(ipaddr);
25
26     httpd_init();
27
28     while(1) {
29         uip_len = network_device_read();
30         if(uip_len > 0) {
31             if(BUF->type == htons(UIP_ETHTYPE_IP)) {
32                 uip_arp_ipin();
33                 uip_input();
34                 /* If the above function invocation resulted in data that
35                  should be sent out on the network, the global variable
36                  uip_len is set to a value > 0. */
37                 if(uip_len > 0) {
38                     uip_arp_out();
39                     network_device_send();
40                 }
41             } else if(BUF->type == htons(UIP_ETHTYPE_ARP)) {
42                 uip_arp_arpin();
43                 /* If the above function invocation resulted in data that
44                  should be sent out on the network, the global variable
45                  uip_len is set to a value > 0. */
46                 if(uip_len > 0) {
47                     network_device_send();
48                 }
49             }
50
51             } else if(timer_expired(&periodic_timer)) {
52                 timer_reset(&periodic_timer);
53                 for(i = 0; i < UIP_CONNS; i++) {
54                     uip_periodic(i);
55                     /* If the above function invocation resulted in data that
56                     should be sent out on the network, the global variable
57                     uip_len is set to a value > 0. */
58                     if(uip_len > 0) {
59                         uip_arp_out();
60                         network_device_send();
61                     }
62                 }
63
64             }
65
66             #if UIP_UDP
67                 for(i = 0; i < UIP_UDP_CONNS; i++) {

```

```
66     uip_udp_periodic(i);
67     /* If the above function invocation resulted in data that
68        should be sent out on the network, the global variable
69        uip_len is set to a value > 0. */
70     if(uip_len > 0) {
71         uip_arp_out();
72         network_device_send();
73     }
74 }
75 #endif /* UIP_UDP */
76
77     /* Call the ARP timer function every 10 seconds. */
78     if(timer_expired(&arp_timer)) {
79         timer_reset(&arp_timer);
80         uip_arp_timer();
81     }
82 }
83 }
84 return 0;
85 }
86 /*-----*/
```

9.4 example-mainloop-without-arp.c

```

1 #include "uip.h"
2 #include "uip_arp.h"
3 #include "network-device.h"
4 #include "httpd.h"
5 #include "timer.h"
6
7 /*-----*/
8 int
9 main(void)
10 {
11     int i;
12     uip_ipaddr_t ipaddr;
13     struct timer periodic_timer;
14
15     timer_set(&periodic_timer, CLOCK_SECOND / 2);
16
17     network_device_init();
18     uip_init();
19
20     uip_ipaddr(ipaddr, 192,168,0,2);
21     uip_sethostaddr(ipaddr);
22
23     httpd_init();
24
25     while(1) {
26         uip_len = network_device_read();
27         if(uip_len > 0) {
28             uip_input();
29             /* If the above function invocation resulted in data that
30              should be sent out on the network, the global variable
31              uip_len is set to a value > 0. */
32             if(uip_len > 0) {
33                 network_device_send();
34             }
35         } else if(timer_expired(&periodic_timer)) {
36             timer_reset(&periodic_timer);
37             for(i = 0; i < UIP_CONNS; i++) {
38                 uip_periodic(i);
39                 /* If the above function invocation resulted in data that
40                  should be sent out on the network, the global variable
41                  uip_len is set to a value > 0. */
42                 if(uip_len > 0) {
43                     network_device_send();
44                 }
45             }
46
47             #if UIP_UDP
48                 for(i = 0; i < UIP_UDP_CONNS; i++) {
49                     uip_udp_periodic(i);
50                     /* If the above function invocation resulted in data that
51                      should be sent out on the network, the global variable
52                      uip_len is set to a value > 0. */
53                     if(uip_len > 0) {
54                         network_device_send();
55                     }
56                 }
57             #endif /* UIP_UDP */
58         }
59     }
60     return 0;
61 }
62 /*-----*/

```

9.5 hello-world.c

```
1 /**
2  * \addtogroup helloworld
3  * @{
4  */
5
6 /**
7  * \file
8  *      An example of how to write uIP applications
9  *      with protosockets.
10 * \author
11 *      Adam Dunkels <adam@sics.se>
12 */
13
14 /*
15 * This is a short example of how to write uIP applications using
16 * protosockets.
17 */
18
19 /*
20 * We define the application state (struct hello_world_state) in the
21 * hello-world.h file, so we need to include it here. We also include
22 * uip.h (since this cannot be included in hello-world.h) and
23 * <string.h>, since we use the memcpy() function in the code.
24 */
25 #include "hello-world.h"
26 #include "uip.h"
27 #include <string.h>
28
29 /*
30 * Declaration of the protosocket function that handles the connection
31 * (defined at the end of the code).
32 */
33 static int handle_connection(struct hello_world_state *s);
34 /*-----*/
35 /*
36 * The initialization function. We must explicitly call this function
37 * from the system initialization code, some time after uip_init() is
38 * called.
39 */
40 void
41 hello_world_init(void)
42 {
43     /* We start to listen for connections on TCP port 1000. */
44     uip_listen(HTONS(1000));
45 }
46 /*-----*/
47 /*
48 * In hello-world.h we have defined the UIP_APPCALL macro to
49 * hello_world_appcall so that this function is uIP's application
50 * function. This function is called whenever an uIP event occurs
51 * (e.g. when a new connection is established, new data arrives, sent
52 * data is acknowledged, data needs to be retransmitted, etc.).
53 */
54 void
55 hello_world_appcall(void)
56 {
57     /*
58      * The uip_conn structure has a field called "appstate" that holds
59      * the application state of the connection. We make a pointer to
60      * this to access it easier.
61      */
62     struct hello_world_state *s = &(uip_conn->appstate);
63
64     /*
65      * If a new connection was just established, we should initialize
```

```
66  * the protosocket in our applications' state structure.
67  */
68  if(uiplib_connected()) {
69      PSOCK_INIT(&s->p, s->inputbuffer, sizeof(s->inputbuffer));
70  }
71
72  /*
73   * Finally, we run the protosocket function that actually handles
74   * the communication. We pass it a pointer to the application state
75   * of the current connection.
76   */
77  handle_connection(s);
78 }
79 /*-----*/
80 /*
81  * This is the protosocket function that handles the communication. A
82  * protosocket function must always return an int, but must never
83  * explicitly return - all return statements are hidden in the PSOCK
84  * macros.
85  */
86 static int
87 handle_connection(struct hello_world_state *s)
88 {
89     PSOCK_BEGIN(&s->p);
90
91     PSOCK_SEND_STR(&s->p, "Hello. What is your name?\n");
92     PSOCK_READTO(&s->p, '\n');
93     strncpy(s->name, s->inputbuffer, sizeof(s->name));
94     PSOCK_SEND_STR(&s->p, "Hello ");
95     PSOCK_SEND_STR(&s->p, s->name);
96     PSOCK_CLOSE(&s->p);
97
98     PSOCK_END(&s->p);
99 }
100 /*-----*/
```

9.6 hello-world.h

```
1 /**
2  * \addtogroup apps
3  * @{
4  */
5
6 /**
7  * \defgroup helloworld Hello, world
8  * @{
9  *
10 * A small example showing how to write applications with
11 * \ref psock "protosockets".
12 */
13
14 /**
15 * \file
16 *      Header file for an example of how to write uIP applications
17 *      with protosockets.
18 * \author
19 *      Adam Dunkels <adam@sics.se>
20 */
21
22 #ifndef __HELLO_WORLD_H__
23 #define __HELLO_WORLD_H__
24
25 /* Since this file will be included by uip.h, we cannot include uip.h
26    here. But we might need to include uipopt.h if we need the u8_t and
27    u16_t datatypes. */
28 #include "uipopt.h"
29
30 #include "psock.h"
31
32 /* Next, we define the uip_tcp_appstate_t datatype. This is the state
33    of our application, and the memory required for this state is
34    allocated together with each TCP connection. One application state
35    for each TCP connection. */
36 typedef struct hello_world_state {
37     struct psock p;
38     char inputbuffer[10];
39     char name[40];
40 } uip_tcp_appstate_t;
41
42 /* Finally we define the application function to be called by uIP. */
43 void hello_world_appcall(void);
44 #ifndef UIP_APPCALL
45 #define UIP_APPCALL hello_world_appcall
46 #endif /* UIP_APPCALL */
47
48 void hello_world_init(void);
49
50 #endif /* __HELLO_WORLD_H__ */
51 /** @} */
52 /** @} */
```


9.7 resolv.c

```
1 /**
2  * \addtogroup apps
3  * @{
4  */
5
6 /**
7  * \defgroup resolv DNS resolver
8  * @{
9  *
10 * The uIP DNS resolver functions are used to lookup a hostname and
11 * map it to a numerical IP address. It maintains a list of resolved
12 * hostnames that can be queried with the resolv_lookup()
13 * function. New hostnames can be resolved using the resolv_query()
14 * function.
15 *
16 * When a hostname has been resolved (or found to be non-existent),
17 * the resolver code calls a callback function called resolv_found()
18 * that must be implemented by the module that uses the resolver.
19 */
20
21 /**
22 * \file
23 * DNS host name to IP address resolver.
24 * \author Adam Dunkels <adam@dunkels.com>
25 *
26 * This file implements a DNS host name to IP address resolver.
27 */
28
29 /*
30 * Copyright (c) 2002-2003, Adam Dunkels.
31 * All rights reserved.
32 *
33 * Redistribution and use in source and binary forms, with or without
34 * modification, are permitted provided that the following conditions
35 * are met:
36 * 1. Redistributions of source code must retain the above copyright
37 * notice, this list of conditions and the following disclaimer.
38 * 2. Redistributions in binary form must reproduce the above copyright
39 * notice, this list of conditions and the following disclaimer in the
40 * documentation and/or other materials provided with the distribution.
41 * 3. The name of the author may not be used to endorse or promote
42 * products derived from this software without specific prior
43 * written permission.
44 *
45 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
46 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
47 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
48 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
49 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
50 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
51 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
52 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
53 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56 *
57 * This file is part of the uIP TCP/IP stack.
58 *
59 * $Id: resolv.c,v 1.5 2006/06/11 21:46:37 adam Exp $
60 *
61 */
62
63 #include "resolv.h"
64 #include "uip.h"
65
```

```

66 #include <string.h>
67
68 #ifndef NULL
69 #define NULL (void *)0
70 #endif /* NULL */
71
72 /** \internal The maximum number of retries when asking for a name. */
73 #define MAX_RETRIES 8
74
75 /** \internal The DNS message header. */
76 struct dns_hdr {
77     u16_t id;
78     u8_t flags1, flags2;
79 #define DNS_FLAG1_RESPONSE      0x80
80 #define DNS_FLAG1_OPCODE_STATUS 0x10
81 #define DNS_FLAG1_OPCODE_INVERSE 0x08
82 #define DNS_FLAG1_OPCODE_STANDARD 0x00
83 #define DNS_FLAG1_AUTHORATIVE 0x04
84 #define DNS_FLAG1_TRUNC      0x02
85 #define DNS_FLAG1_RD         0x01
86 #define DNS_FLAG2_RA         0x80
87 #define DNS_FLAG2_ERR_MASK   0x0f
88 #define DNS_FLAG2_ERR_NONE   0x00
89 #define DNS_FLAG2_ERR_NAME    0x03
90     u16_t numquestions;
91     u16_t numanswers;
92     u16_t numauthrr;
93     u16_t numextrarr;
94 };
95
96 /** \internal The DNS answer message structure. */
97 struct dns_answer {
98     /* DNS answer record starts with either a domain name or a pointer
99     to a name already present somewhere in the packet. */
100     u16_t type;
101     u16_t class;
102     u16_t ttl[2];
103     u16_t len;
104     uip_ipaddr_t ipaddr;
105 };
106
107 struct namemap {
108 #define STATE_UNUSED 0
109 #define STATE_NEW 1
110 #define STATE_ASKING 2
111 #define STATE_DONE 3
112 #define STATE_ERROR 4
113     u8_t state;
114     u8_t tmr;
115     u8_t retries;
116     u8_t seqno;
117     u8_t err;
118     char name[32];
119     uip_ipaddr_t ipaddr;
120 };
121
122 #ifndef UIP_CONF_RESOLV_ENTRIES
123 #define RESOLV_ENTRIES 4
124 #else /* UIP_CONF_RESOLV_ENTRIES */
125 #define RESOLV_ENTRIES UIP_CONF_RESOLV_ENTRIES
126 #endif /* UIP_CONF_RESOLV_ENTRIES */
127
128
129 static struct namemap names[RESOLV_ENTRIES];
130
131 static u8_t seqno;
132

```

```

133 static struct uip_udp_conn *resolv_conn = NULL;
134
135
136 /*-----*/
137 /** \internal
138  * Walk through a compact encoded DNS name and return the end of it.
139  *
140  * \return The end of the name.
141  */
142 /*-----*/
143 static unsigned char *
144 parse_name(unsigned char *query)
145 {
146     unsigned char n;
147
148     do {
149         n = *query++;
150
151         while(n > 0) {
152             /* printf("%c", *query);*/
153             ++query;
154             --n;
155         };
156         /* printf(".");*/
157     } while(*query != 0);
158     /* printf("\n");*/
159     return query + 1;
160 }
161 /*-----*/
162 /** \internal
163  * Runs through the list of names to see if there are any that have
164  * not yet been queried and, if so, sends out a query.
165  */
166 /*-----*/
167 static void
168 check_entries(void)
169 {
170     register struct dns_hdr *hdr;
171     char *query, *nptr, *nameptr;
172     static u8_t i;
173     static u8_t n;
174     register struct namemap *namemapptr;
175
176     for(i = 0; i < RESOLV_ENTRIES; ++i) {
177         namemapptr = &names[i];
178         if(namemapptr->state == STATE_NEW ||
179            namemapptr->state == STATE Asking) {
180             if(namemapptr->state == STATE Asking) {
181                 if(--namemapptr->tmr == 0) {
182                     if(++namemapptr->retries == MAX_RETRIES) {
183                         namemapptr->state = STATE_ERROR;
184                         resolv_found(namemapptr->name, NULL);
185                         continue;
186                     }
187                     namemapptr->tmr = namemapptr->retries;
188                 } else {
189                     /* printf("Timer %d\n", namemapptr->tmr);*/
190                     /* Its timer has not run out, so we move on to next
191                        entry. */
192                     continue;
193                 }
194             } else {
195                 namemapptr->state = STATE Asking;
196                 namemapptr->tmr = 1;
197                 namemapptr->retries = 0;
198             }
199             hdr = (struct dns_hdr *)uip_appdata;

```

```

200     memset(hdr, 0, sizeof(struct dns_hdr));
201     hdr->id = htons(i);
202     hdr->flags1 = DNS_FLAG1_RD;
203     hdr->numquestions = HTONS(1);
204     query = (char *)uip_appdata + 12;
205     nameptr = namemaptr->name;
206     --nameptr;
207     /* Convert hostname into suitable query format. */
208     do {
209         ++nameptr;
210         nptr = query;
211         ++query;
212         for(n = 0; *nameptr != '.' && *nameptr != 0; ++nameptr) {
213             *query = *nameptr;
214             ++query;
215             ++n;
216         }
217         *nptr = n;
218     } while(*nameptr != 0);
219     {
220         static unsigned char endquery[] =
221             {0,0,1,0,1};
222         memcpy(query, endquery, 5);
223     }
224     uip_udp_send((unsigned char)(query + 5 - (char *)uip_appdata));
225     break;
226 }
227 }
228 }
229 /*-----*/
230 /** \internal
231  * Called when new UDP data arrives.
232  */
233 /*-----*/
234 static void
235 newdata(void)
236 {
237     char *nameptr;
238     struct dns_answer *ans;
239     struct dns_hdr *hdr;
240     static u8_t nquestions, nanswers;
241     static u8_t i;
242     register struct namemap *namemaptr;
243
244     hdr = (struct dns_hdr *)uip_appdata;
245     /* printf("ID %d\n", htons(hdr->id));
246     printf("Query %d\n", hdr->flags1 & DNS_FLAG1_RESPONSE);
247     printf("Error %d\n", hdr->flags2 & DNS_FLAG2_ERR_MASK);
248     printf("Num questions %d, answers %d, authrr %d, extrarr %d\n",
249     htons(hdr->numquestions),
250     htons(hdr->numanswers),
251     htons(hdr->numauthrr),
252     htons(hdr->numextrarr));
253     */
254
255     /* The ID in the DNS header should be our entry into the name
256        table. */
257     i = htons(hdr->id);
258     namemaptr = &names[i];
259     if(i < RESOLV_ENTRIES &&
260        namemaptr->state == STATE_ASKING) {
261
262         /* This entry is now finished. */
263         namemaptr->state = STATE_DONE;
264         namemaptr->err = hdr->flags2 & DNS_FLAG2_ERR_MASK;
265
266         /* Check for error. If so, call callback to inform. */

```

```

267     if(namemapptr->err != 0) {
268         namemapptr->state = STATE_ERROR;
269         resolv_found(namemapptr->name, NULL);
270         return;
271     }
272
273     /* We only care about the question(s) and the answers. The authrr
274        and the extrarr are simply discarded. */
275     nquestions = htons(hdr->numquestions);
276     nanswers = htons(hdr->numanswers);
277
278     /* Skip the name in the question. XXX: This should really be
279        checked against the name in the question, to be sure that they
280        match. */
281     nameptr = parse_name((char *)uip_appdata + 12) + 4;
282
283     while(nanswers > 0) {
284         /* The first byte in the answer resource record determines if it
285            is a compressed record or a normal one. */
286         if(*nameptr & 0xc0) {
287             /* Compressed name. */
288             nameptr += 2;
289             /* printf("Compressed answer\n"); */
290         } else {
291             /* Not compressed name. */
292             nameptr = parse_name((char *)nameptr);
293         }
294
295         ans = (struct dns_answer *)nameptr;
296         /* printf("Answer: type %x, class %x, ttl %x, length %x\n",
297            htons(ans->type), htons(ans->class), (htons(ans->ttl[0])
298            << 16) | htons(ans->ttl[1]), htons(ans->len)); */
299
300         /* Check for IP address type and Internet class. Others are
301            discarded. */
302         if(ans->type == HTONS(1) &&
303            ans->class == HTONS(1) &&
304            ans->len == HTONS(4)) {
305             /* printf("IP address %d.%d.%d.%d\n",
306                htons(ans->ipaddr[0]) >> 8,
307                htons(ans->ipaddr[0]) & 0xff,
308                htons(ans->ipaddr[1]) >> 8,
309                htons(ans->ipaddr[1]) & 0xff); */
310             /* XXX: we should really check that this IP address is the one
311                we want. */
312             namemapptr->ipaddr[0] = ans->ipaddr[0];
313             namemapptr->ipaddr[1] = ans->ipaddr[1];
314
315             resolv_found(namemapptr->name, namemapptr->ipaddr);
316             return;
317         } else {
318             nameptr = nameptr + 10 + htons(ans->len);
319         }
320         --nanswers;
321     }
322 }
323
324 }
325 /*-----*/
326 /** \internal
327  * The main UDP function.
328  */
329 /*-----*/
330 void
331 resolv_appcall(void)
332 {
333     if(uip_udp_conn->rport == HTONS(53)) {

```

```

334     if(uip_poll()) {
335         check_entries();
336     }
337     if(uip_newdata()) {
338         newdata();
339     }
340 }
341 }
342 /*-----*/
343 /**
344  * Queues a name so that a question for the name will be sent out.
345  *
346  * \param name The hostname that is to be queried.
347  */
348 /*-----*/
349 void
350 resolv_query(char *name)
351 {
352     static u8_t i;
353     static u8_t lseq, lseqi;
354     register struct namemap *nameptr;
355
356     lseq = lseqi = 0;
357
358     for(i = 0; i < RESOLV_ENTRIES; ++i) {
359         nameptr = &names[i];
360         if(nameptr->state == STATE_UNUSED) {
361             break;
362         }
363         if(seqno - nameptr->seqno > lseq) {
364             lseq = seqno - nameptr->seqno;
365             lseqi = i;
366         }
367     }
368
369     if(i == RESOLV_ENTRIES) {
370         i = lseqi;
371         nameptr = &names[i];
372     }
373
374     /* printf("Using entry %d\n", i);*/
375
376     strcpy(nameptr->name, name);
377     nameptr->state = STATE_NEW;
378     nameptr->seqno = seqno;
379     ++seqno;
380 }
381 /*-----*/
382 /**
383  * Look up a hostname in the array of known hostnames.
384  *
385  * \note This function only looks in the internal array of known
386  * hostnames, it does not send out a query for the hostname if none
387  * was found. The function resolv_query() can be used to send a query
388  * for a hostname.
389  *
390  * \return A pointer to a 4-byte representation of the hostname's IP
391  * address, or NULL if the hostname was not found in the array of
392  * hostnames.
393  */
394 /*-----*/
395 u16_t *
396 resolv_lookup(char *name)
397 {
398     static u8_t i;
399     struct namemap *nameptr;
400

```

```

401  /* Walk through the list to see if the name is in there. If it is
402     not, we return NULL. */
403  for(i = 0; i < RESOLV_ENTRIES; ++i) {
404      nameptr = &names[i];
405      if(nameptr->state == STATE_DONE &&
406         strcmp(name, nameptr->name) == 0) {
407          return nameptr->ipaddr;
408      }
409  }
410  return NULL;
411 }
412 /*-----*/
413 /**
414  * Obtain the currently configured DNS server.
415  *
416  * \return A pointer to a 4-byte representation of the IP address of
417  * the currently configured DNS server or NULL if no DNS server has
418  * been configured.
419  */
420 /*-----*/
421 ul6_t *
422 resolv_getserver(void)
423 {
424     if(resolv_conn == NULL) {
425         return NULL;
426     }
427     return resolv_conn->ripaddr;
428 }
429 /*-----*/
430 /**
431  * Configure which DNS server to use for queries.
432  *
433  * \param dnsserver A pointer to a 4-byte representation of the IP
434  * address of the DNS server to be configured.
435  */
436 /*-----*/
437 void
438 resolv_conf(ul6_t *dnsserver)
439 {
440     if(resolv_conn != NULL) {
441         uip_udp_remove(resolv_conn);
442     }
443     resolv_conn = uip_udp_new(dnsserver, HTONS(53));
444 }
445 /*-----*/
446 /**
447  * Initialize the resolver.
448  */
449 /*-----*/
450 /*-----*/
451 void
452 resolv_init(void)
453 {
454     static u8_t i;
455     for(i = 0; i < RESOLV_ENTRIES; ++i) {
456         names[i].state = STATE_DONE;
457     }
458 }
459
460 }
461 /*-----*/
462
463 /** @} */
464 /** @} */

```

9.8 resolv.h

```
1 /**
2  * \addtogroup resolv
3  * @{
4  */
5 /**
6  * \file
7  * DNS resolver code header file.
8  * \author Adam Dunkels <adam@dunkels.com>
9  */
10
11 /*
12  * Copyright (c) 2002-2003, Adam Dunkels.
13  * All rights reserved.
14  *
15  * Redistribution and use in source and binary forms, with or without
16  * modification, are permitted provided that the following conditions
17  * are met:
18  * 1. Redistributions of source code must retain the above copyright
19  * notice, this list of conditions and the following disclaimer.
20  * 2. Redistributions in binary form must reproduce the above copyright
21  * notice, this list of conditions and the following disclaimer in the
22  * documentation and/or other materials provided with the distribution.
23  * 3. The name of the author may not be used to endorse or promote
24  * products derived from this software without specific prior
25  * written permission.
26  *
27  * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
28  * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
29  * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
30  * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
31  * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
32  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
33  * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
34  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
35  * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
36  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
37  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
38  *
39  * This file is part of the uIP TCP/IP stack.
40  *
41  * $Id: resolv.h,v 1.4 2006/06/11 21:46:37 adam Exp $
42  */
43
44 #ifndef __RESOLV_H__
45 #define __RESOLV_H__
46
47 typedef int uip_udp_appstate_t;
48 void resolv_appcall(void);
49 #define UIP_UDP_APPCALL resolv_appcall
50
51 #include "uipopt.h"
52
53 /**
54  * Callback function which is called when a hostname is found.
55  *
56  * This function must be implemented by the module that uses the DNS
57  * resolver. It is called when a hostname is found, or when a hostname
58  * was not found.
59  *
60  * \param name A pointer to the name that was looked up. \param
61  * \param ipaddr A pointer to a 4-byte array containing the IP address of the
62  * \param hostname, or NULL if the hostname could not be found.
63  */
64 void resolv_found(char *name, u16_t *ipaddr);
65
```

```
66 /* Functions. */
67 void resolv_conf(u16_t *dnsserver);
68 u16_t *resolv_getserver(void);
69 void resolv_init(void);
70 u16_t *resolv_lookup(char *name);
71 void resolv_query(char *name);
72
73 #endif /* __RESOLV_H__ */
74
75 /** @} */
```

9.9 smtp.c

```
1 /**
2  * \addtogroup apps
3  * @{
4  */
5
6 /**
7  * \defgroup smtp SMTP E-mail sender
8  * @{
9  *
10 * The Simple Mail Transfer Protocol (SMTP) as defined by RFC821 is
11 * the standard way of sending and transferring e-mail on the
12 * Internet. This simple example implementation is intended as an
13 * example of how to implement protocols in uIP, and is able to send
14 * out e-mail but has not been extensively tested.
15 */
16
17 /**
18 * \file
19 * SMTP example implementation
20 * \author Adam Dunkels <adam@dunkels.com>
21 */
22
23 /*
24 * Copyright (c) 2004, Adam Dunkels.
25 * All rights reserved.
26 *
27 * Redistribution and use in source and binary forms, with or without
28 * modification, are permitted provided that the following conditions
29 * are met:
30 * 1. Redistributions of source code must retain the above copyright
31 *    notice, this list of conditions and the following disclaimer.
32 * 2. Redistributions in binary form must reproduce the above copyright
33 *    notice, this list of conditions and the following disclaimer in the
34 *    documentation and/or other materials provided with the distribution.
35 * 3. Neither the name of the Institute nor the names of its contributors
36 *    may be used to endorse or promote products derived from this software
37 *    without specific prior written permission.
38 *
39 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
40 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
41 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
42 * ARE DISCLAIMED. IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
43 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
44 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
45 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
46 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
47 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
48 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
49 * SUCH DAMAGE.
50 *
51 * This file is part of the uIP TCP/IP stack.
52 *
53 * Author: Adam Dunkels <adam@sics.se>
54 *
55 * $Id: smtp.c,v 1.4 2006/06/11 21:46:37 adam Exp $
56 */
57 #include "smtp.h"
58
59 #include "smtp-strings.h"
60 #include "psock.h"
61 #include "uip.h"
62
63 #include <string.h>
64
65 static struct smtp_state s;
```

```

66
67 static char *localhostname;
68 static uip_ipaddr_t smtpserver;
69
70 #define ISO_nl 0x0a
71 #define ISO_cr 0x0d
72
73 #define ISO_period 0x2e
74
75 #define ISO_2 0x32
76 #define ISO_3 0x33
77 #define ISO_4 0x34
78 #define ISO_5 0x35
79
80
81 /*-----*/
82 static
83 PT_THREAD(smtp_thread(void))
84 {
85     PSOCK_BEGIN(&s.psock);
86
87     PSOCK_READTO(&s.psock, ISO_nl);
88
89     if(strncmp(s.inputbuffer, smtp_220, 3) != 0) {
90         PSOCK_CLOSE(&s.psock);
91         smtp_done(2);
92         PSOCK_EXIT(&s.psock);
93     }
94
95     PSOCK_SEND_STR(&s.psock, (char *)smtp_helo);
96     PSOCK_SEND_STR(&s.psock, localhostname);
97     PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
98
99     PSOCK_READTO(&s.psock, ISO_nl);
100
101     if(s.inputbuffer[0] != ISO_2) {
102         PSOCK_CLOSE(&s.psock);
103         smtp_done(3);
104         PSOCK_EXIT(&s.psock);
105     }
106
107     PSOCK_SEND_STR(&s.psock, (char *)smtp_mail_from);
108     PSOCK_SEND_STR(&s.psock, s.from);
109     PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
110
111     PSOCK_READTO(&s.psock, ISO_nl);
112
113     if(s.inputbuffer[0] != ISO_2) {
114         PSOCK_CLOSE(&s.psock);
115         smtp_done(4);
116         PSOCK_EXIT(&s.psock);
117     }
118
119     PSOCK_SEND_STR(&s.psock, (char *)smtp_rcpt_to);
120     PSOCK_SEND_STR(&s.psock, s.to);
121     PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
122
123     PSOCK_READTO(&s.psock, ISO_nl);
124
125     if(s.inputbuffer[0] != ISO_2) {
126         PSOCK_CLOSE(&s.psock);
127         smtp_done(5);
128         PSOCK_EXIT(&s.psock);
129     }
130
131     if(s.cc != 0) {
132         PSOCK_SEND_STR(&s.psock, (char *)smtp_rcpt_to);

```

```

133     PSOCK_SEND_STR(&s.psock, s.cc);
134     PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
135
136     PSOCK_READTO(&s.psock, ISO_nl);
137
138     if(s.inputbuffer[0] != ISO_2) {
139         PSOCK_CLOSE(&s.psock);
140         smtp_done(6);
141         PSOCK_EXIT(&s.psock);
142     }
143 }
144
145 PSOCK_SEND_STR(&s.psock, (char *)smtp_data);
146
147 PSOCK_READTO(&s.psock, ISO_nl);
148
149 if(s.inputbuffer[0] != ISO_3) {
150     PSOCK_CLOSE(&s.psock);
151     smtp_done(7);
152     PSOCK_EXIT(&s.psock);
153 }
154
155 PSOCK_SEND_STR(&s.psock, (char *)smtp_to);
156 PSOCK_SEND_STR(&s.psock, s.to);
157 PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
158
159 if(s.cc != 0) {
160     PSOCK_SEND_STR(&s.psock, (char *)smtp_cc);
161     PSOCK_SEND_STR(&s.psock, s.cc);
162     PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
163 }
164
165 PSOCK_SEND_STR(&s.psock, (char *)smtp_from);
166 PSOCK_SEND_STR(&s.psock, s.from);
167 PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
168
169 PSOCK_SEND_STR(&s.psock, (char *)smtp_subject);
170 PSOCK_SEND_STR(&s.psock, s.subject);
171 PSOCK_SEND_STR(&s.psock, (char *)smtp_crnl);
172
173 PSOCK_SEND(&s.psock, s.msg, s.msglen);
174
175 PSOCK_SEND_STR(&s.psock, (char *)smtp_crnlperiodcrnl);
176
177 PSOCK_READTO(&s.psock, ISO_nl);
178 if(s.inputbuffer[0] != ISO_2) {
179     PSOCK_CLOSE(&s.psock);
180     smtp_done(8);
181     PSOCK_EXIT(&s.psock);
182 }
183
184 PSOCK_SEND_STR(&s.psock, (char *)smtp_quit);
185 smtp_done(SMTP_ERR_OK);
186 PSOCK_END(&s.psock);
187 }
188 /*-----*/
189 void
190 smtp_appcall(void)
191 {
192     if(uiplib_closed()) {
193         s.connected = 0;
194         return;
195     }
196     if(uiplib_aborted() || uip_timedout()) {
197         s.connected = 0;
198         smtp_done(1);
199         return;

```

```

200     }
201     smtp_thread();
202 }
203 /*-----*/
204 /**
205  * Specificy an SMTP server and hostname.
206  *
207  * This function is used to configure the SMTP module with an SMTP
208  * server and the hostname of the host.
209  *
210  * \param lhostname The hostname of the uIP host.
211  *
212  * \param server A pointer to a 4-byte array representing the IP
213  * address of the SMTP server to be configured.
214  */
215 void
216 smtp_configure(char *lhostname, void *server)
217 {
218     localhostname = lhostname;
219     uip_ipaddr_copy(smtpserver, server);
220 }
221 /*-----*/
222 /**
223  * Send an e-mail.
224  *
225  * \param to The e-mail address of the receiver of the e-mail.
226  * \param cc The e-mail address of the CC: receivers of the e-mail.
227  * \param from The e-mail address of the sender of the e-mail.
228  * \param subject The subject of the e-mail.
229  * \param msg The actual e-mail message.
230  * \param msglen The length of the e-mail message.
231  */
232 unsigned char
233 smtp_send(char *to, char *cc, char *from,
234           char *subject, char *msg, u16_t msglen)
235 {
236     struct uip_conn *conn;
237
238     conn = uip_connect(smtpserver, HTONS(25));
239     if(conn == NULL) {
240         return 0;
241     }
242     s.connected = 1;
243     s.to = to;
244     s.cc = cc;
245     s.from = from;
246     s.subject = subject;
247     s.msg = msg;
248     s.msglen = msglen;
249
250     PSOCK_INIT(&s.psock, s.inputbuffer, sizeof(s.inputbuffer));
251
252     return 1;
253 }
254 /*-----*/
255 void
256 smtp_init(void)
257 {
258     s.connected = 0;
259 }
260 /*-----*/
261 /** @} */
262 /** @} */

```

9.10 smtp.h

```
1
2 /**
3  * \addtogroup smtp
4  * @{
5  */
6
7
8 /**
9  * \file
10 * SMTP header file
11 * \author Adam Dunkels <adam@dunkels.com>
12 */
13
14 /*
15 * Copyright (c) 2002, Adam Dunkels.
16 * All rights reserved.
17 *
18 * Redistribution and use in source and binary forms, with or without
19 * modification, are permitted provided that the following conditions
20 * are met:
21 * 1. Redistributions of source code must retain the above copyright
22 *    notice, this list of conditions and the following disclaimer.
23 * 2. Redistributions in binary form must reproduce the above copyright
24 *    notice, this list of conditions and the following disclaimer in the
25 *    documentation and/or other materials provided with the distribution.
26 * 3. The name of the author may not be used to endorse or promote
27 *    products derived from this software without specific prior
28 *    written permission.
29 *
30 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
31 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
32 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
33 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
34 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
35 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
36 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
37 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
38 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
39 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
40 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
41 *
42 * This file is part of the uIP TCP/IP stack.
43 *
44 * $Id: smtp.h,v 1.4 2006/06/11 21:46:37 adam Exp $
45 */
46
47 #ifndef __SMTP_H__
48 #define __SMTP_H__
49
50 #include "uipopt.h"
51
52 /**
53  * Error number that signifies a non-error condition.
54  */
55 #define SMTP_ERR_OK 0
56
57 /**
58  * Callback function that is called when an e-mail transmission is
59  * done.
60  *
61  * This function must be implemented by the module that uses the SMTP
62  * module.
63  *
64  * \param error The number of the error if an error occurred, or
65  * SMTP_ERR_OK.
```

```
66  */
67 void smtp_done(unsigned char error);
68
69 void smtp_init(void);
70
71 /* Functions. */
72 void smtp_configure(char *localhostname, u16_t *smtpserver);
73 unsigned char smtp_send(char *to, char *from,
74                         char *subject, char *msg,
75                         u16_t msglen);
76 #define SMTP_SEND(to, cc, from, subject, msg) \
77     smtp_send(to, cc, from, subject, msg, strlen(msg))
78
79 void smtp_appcall(void);
80
81 struct smtp_state {
82     u8_t state;
83     char *to;
84     char *from;
85     char *subject;
86     char *msg;
87     u16_t msglen;
88
89     u16_t sentlen, textlen;
90     u16_t sendptr;
91
92 };
93
94
95 #ifndef UIP_APPCALL
96 #define UIP_APPCALL    smtp_appcall
97 #endif
98 typedef struct smtp_state uip_tcp_appstate_t;
99
100
101 #endif /* __SMTP_H__ */
102
103 /** @} */
```

9.11 telnetd.c

```
1 /**
2  * \addtogroup telnetd
3  * @{
4  */
5
6 /**
7  * \file
8  *      Shell server
9  * \author
10 *      Adam Dunkels <adam@sics.se>
11 */
12
13 /*
14 * Copyright (c) 2003, Adam Dunkels.
15 * All rights reserved.
16 *
17 * Redistribution and use in source and binary forms, with or without
18 * modification, are permitted provided that the following conditions
19 * are met:
20 * 1. Redistributions of source code must retain the above copyright
21 *    notice, this list of conditions and the following disclaimer.
22 * 2. Redistributions in binary form must reproduce the above copyright
23 *    notice, this list of conditions and the following disclaimer in the
24 *    documentation and/or other materials provided with the distribution.
25 * 3. The name of the author may not be used to endorse or promote
26 *    products derived from this software without specific prior
27 *    written permission.
28 *
29 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
30 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
31 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
32 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
33 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
34 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
35 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
36 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
37 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
38 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
39 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
40 *
41 * This file is part of the uIP TCP/IP stack
42 *
43 * $Id: telnetd.c,v 1.2 2006/06/07 09:43:54 adam Exp $
44 *
45 */
46
47 #include "uip.h"
48 #include "telnetd.h"
49 #include "memb.h"
50 #include "shell.h"
51
52 #include <string.h>
53
54 #define ISO_nl      0x0a
55 #define ISO_cr      0x0d
56
57 struct telnetd_line {
58     char line[TELNETD_CONF_LINELEN];
59 };
60 MEMB(linemem, struct telnetd_line, TELNETD_CONF_NUMLINES);
61
62 #define STATE_NORMAL 0
63 #define STATE_IAC    1
64 #define STATE_WILL   2
65 #define STATE_WONT   3
```



```

66 #define STATE_DO      4
67 #define STATE_DONT    5
68 #define STATE_CLOSE   6
69
70 static struct telnetd_state s;
71
72 #define TELNET_IAC     255
73 #define TELNET_WILL    251
74 #define TELNET_WONT    252
75 #define TELNET_DO      253
76 #define TELNET_DONT    254
77 /*-----*/
78 static char *
79 alloc_line(void)
80 {
81     return memb_alloc(&linemem);
82 }
83 /*-----*/
84 static void
85 dealloc_line(char *line)
86 {
87     memb_free(&linemem, line);
88 }
89 /*-----*/
90 void
91 shell_quit(char *str)
92 {
93     s.state = STATE_CLOSE;
94 }
95 /*-----*/
96 static void
97 sendline(char *line)
98 {
99     static unsigned int i;
100
101     for(i = 0; i < TELNETD_CONF_NUMLINES; ++i) {
102         if(s.lines[i] == NULL) {
103             s.lines[i] = line;
104             break;
105         }
106     }
107     if(i == TELNETD_CONF_NUMLINES) {
108         dealloc_line(line);
109     }
110 }
111 /*-----*/
112 void
113 shell_prompt(char *str)
114 {
115     char *line;
116     line = alloc_line();
117     if(line != NULL) {
118         strncpy(line, str, TELNETD_CONF_LINELEN);
119         /* petstciiconv_toascii(line, TELNETD_CONF_LINELEN); */
120         sendline(line);
121     }
122 }
123 /*-----*/
124 void
125 shell_output(char *str1, char *str2)
126 {
127     static unsigned len;
128     char *line;
129
130     line = alloc_line();
131     if(line != NULL) {
132         len = strlen(str1);

```

```

133     strncpy(line, str1, TELNETD_CONF_LINELEN);
134     if(len < TELNETD_CONF_LINELEN) {
135         strncpy(line + len, str2, TELNETD_CONF_LINELEN - len);
136     }
137     len = strlen(line);
138     if(len < TELNETD_CONF_LINELEN - 2) {
139         line[len] = ISO_cr;
140         line[len+1] = ISO_n1;
141         line[len+2] = 0;
142     }
143     /*     petSCIIconv_toascii(line, TELNETD_CONF_LINELEN); */
144     sendline(line);
145 }
146 }
147 /*-----*/
148 void
149 telnetd_init(void)
150 {
151     uip_listen(HTONS(23));
152     memb_init(&linemem);
153     shell_init();
154 }
155 /*-----*/
156 static void
157 acked(void)
158 {
159     static unsigned int i;
160
161     while(s.numsent > 0) {
162         dealloc_line(s.lines[0]);
163         for(i = 1; i < TELNETD_CONF_NUMLINES; ++i) {
164             s.lines[i - 1] = s.lines[i];
165         }
166         s.lines[TELNETD_CONF_NUMLINES - 1] = NULL;
167         --s.numsent;
168     }
169 }
170 /*-----*/
171 static void
172 senddata(void)
173 {
174     static char *bufptr, *lineptr;
175     static int buflen, linelen;
176
177     bufptr = uip_appdata;
178     buflen = 0;
179     for(s.numsent = 0; s.numsent < TELNETD_CONF_NUMLINES &&
180         s.lines[s.numsent] != NULL ; ++s.numsent) {
181         lineptr = s.lines[s.numsent];
182         linelen = strlen(lineptr);
183         if(linelen > TELNETD_CONF_LINELEN) {
184             linelen = TELNETD_CONF_LINELEN;
185         }
186         if(buflen + linelen < uip_mss()) {
187             memcpy(bufptr, lineptr, linelen);
188             bufptr += linelen;
189             buflen += linelen;
190         } else {
191             break;
192         }
193     }
194     uip_send(uip_appdata, buflen);
195 }
196 /*-----*/
197 static void
198 closed(void)
199 {

```

```

200     static unsigned int i;
201
202     for(i = 0; i < TELNETD_CONF_NUMLINES; ++i) {
203         if(s.lines[i] != NULL) {
204             dealloc_line(s.lines[i]);
205         }
206     }
207 }
208 /*-----*/
209 static void
210 get_char(u8_t c)
211 {
212     if(c == ISO_cr) {
213         return;
214     }
215
216     s.buf[(int)s.bufptr] = c;
217     if(s.buf[(int)s.bufptr] == ISO_nl ||
218        s.bufptr == sizeof(s.buf) - 1) {
219         if(s.bufptr > 0) {
220             s.buf[(int)s.bufptr] = 0;
221             /*      petsciiconv_topetscii(s.buf, TELNETD_CONF_LINELEN); */
222         }
223         shell_input(s.buf);
224         s.bufptr = 0;
225     } else {
226         ++s.bufptr;
227     }
228 }
229 /*-----*/
230 static void
231 sendopt(u8_t option, u8_t value)
232 {
233     char *line;
234     line = alloc_line();
235     if(line != NULL) {
236         line[0] = TELNET_IAC;
237         line[1] = option;
238         line[2] = value;
239         line[3] = 0;
240         sendline(line);
241     }
242 }
243 /*-----*/
244 static void
245 newdata(void)
246 {
247     u16_t len;
248     u8_t c;
249     char *dataptr;
250
251
252     len = uip_datalen();
253     dataptr = (char *)uip_appdata;
254
255     while(len > 0 && s.bufptr < sizeof(s.buf)) {
256         c = *dataptr;
257         ++dataptr;
258         --len;
259         switch(s.state) {
260             case STATE_IAC:
261                 if(c == TELNET_IAC) {
262                     get_char(c);
263                     s.state = STATE_NORMAL;
264                 } else {
265                     switch(c) {
266                         case TELNET_WILL:

```

```

267         s.state = STATE_WILL;
268         break;
269     case TELNET_WONT:
270         s.state = STATE_WONT;
271         break;
272     case TELNET_DO:
273         s.state = STATE_DO;
274         break;
275     case TELNET_DONT:
276         s.state = STATE_DONT;
277         break;
278     default:
279         s.state = STATE_NORMAL;
280         break;
281     }
282 }
283 break;
284 case STATE_WILL:
285     /* Reply with a DONT */
286     sendopt(TELNET_DONT, c);
287     s.state = STATE_NORMAL;
288     break;
289
290 case STATE_WONT:
291     /* Reply with a DONT */
292     sendopt(TELNET_DONT, c);
293     s.state = STATE_NORMAL;
294     break;
295 case STATE_DO:
296     /* Reply with a WONT */
297     sendopt(TELNET_WONT, c);
298     s.state = STATE_NORMAL;
299     break;
300 case STATE_DONT:
301     /* Reply with a WONT */
302     sendopt(TELNET_WONT, c);
303     s.state = STATE_NORMAL;
304     break;
305 case STATE_NORMAL:
306     if(c == TELNET_IAC) {
307         s.state = STATE_IAC;
308     } else {
309         get_char(c);
310     }
311     break;
312 }
313
314
315 }
316
317 }
318 /*-----*/
319 void
320 telnetd_appcall(void)
321 {
322     static unsigned int i;
323     if(uiplib_connected()) {
324         /* tcp_markconn(uiplib_conn, &s);*/
325         for(i = 0; i < TELNETD_CONF_NUMLINES; ++i) {
326             s.lines[i] = NULL;
327         }
328         s.bufptr = 0;
329         s.state = STATE_NORMAL;
330
331         shell_start();
332     }
333 }

```

```
334     if(s.state == STATE_CLOSE) {
335         s.state = STATE_NORMAL;
336         uip_close();
337         return;
338     }
339
340     if(uip_closed() ||
341        uip_aborted() ||
342        uip_timedout()) {
343         closed();
344     }
345
346     if(uip_acked()) {
347         acked();
348     }
349
350     if(uip_newdata()) {
351         newdata();
352     }
353
354     if(uip_rexmit() ||
355        uip_newdata() ||
356        uip_acked() ||
357        uip_connected() ||
358        uip_poll()) {
359         senddata();
360     }
361 }
362 /*-----*/
363 /** @} */
```

9.12 telnetd.h

```
1 /**
2  * \addtogroup apps
3  * @{
4  */
5
6 /**
7  * \defgroup telnetd Telnet server
8  * @{
9  *
10 * The uIP telnet server
11 *
12 */
13
14 /**
15 * \file
16 *      Shell server
17 * \author
18 *      Adam Dunkels <adam@sics.se>
19 */
20
21 /*
22 * Copyright (c) 2003, Adam Dunkels.
23 * All rights reserved.
24 *
25 * Redistribution and use in source and binary forms, with or without
26 * modification, are permitted provided that the following conditions
27 * are met:
28 * 1. Redistributions of source code must retain the above copyright
29 *    notice, this list of conditions and the following disclaimer.
30 * 2. Redistributions in binary form must reproduce the above
31 *    copyright notice, this list of conditions and the following
32 *    disclaimer in the documentation and/or other materials provided
33 *    with the distribution.
34 * 3. The name of the author may not be used to endorse or promote
35 *    products derived from this software without specific prior
36 *    written permission.
37 *
38 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
39 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
40 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
41 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
42 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
43 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
44 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
45 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
46 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
47 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
48 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
49 *
50 * This file is part of the uIP TCP/IP stack
51 *
52 * $Id: telnetd.h,v 1.2 2006/06/07 09:43:54 adam Exp $
53 *
54 */
55 #ifndef __TELNETD_H__
56 #define __TELNETD_H__
57
58 #include "uiptopt.h"
59
60 void telnetd_appcall(void);
61
62 #ifndef TELNETD_CONF_LINELEN
63 #define TELNETD_CONF_LINELEN 40
64 #endif
65 #ifndef TELNETD_CONF_NUMLINES
```

```
66 #define TELNETD_CONF_NUMLINES 16
67 #endif
68
69 struct telnetd_state {
70     char *lines[TELNETD_CONF_NUMLINES];
71     char buf[TELNETD_CONF_LINELEN];
72     char bufptr;
73     u8_t numsent;
74     u8_t state;
75 };
76
77 typedef struct telnetd_state uip_tcp_appstate_t;
78
79 #ifndef UIP_APPCALL
80 #define UIP_APPCALL    telnetd_appcall
81 #endif
82
83 #endif /* __TELNETD_H__ */
84 /** @} */
```

9.13 uip-code-style.c

```

1 /* This is the official code style of uIP. */
2
3 /**
4  * \defgroup codestyle Coding style
5  *
6  * This is how a Doxygen module is documented - start with a \defgroup
7  * Doxygen keyword at the beginning of the file to define a module,
8  * and use the \addtogroup Doxygen keyword in all other files that
9  * belong to the same module. Typically, the \defgroup is placed in
10 * the .h file and \addtogroup in the .c file.
11 *
12 * @{
13 */
14
15 /**
16 * \file
17 *      A brief description of what this file is.
18 * \author
19 *      Adam Dunkels <adam@sics.se>
20 *
21 *      Every file that is part of a documented module has to have
22 *      a \file block, else it will not show up in the Doxygen
23 *      "Modules" * section.
24 */
25
26 /* Single line comments look like this. */
27
28 /*
29 * Multi-line comments look like this. Comments should preferably be
30 * full sentences, filled to look like real paragraphs.
31 */
32
33 #include "uip.h"
34
35 /*
36 * Make sure that non-global variables are all maked with the static
37 * keyword. This keeps the size of the symbol table down.
38 */
39 static int flag;
40
41 /*
42 * All variables and functions that are visible outside of the file
43 * should have the module name prepended to them. This makes it easy
44 * to know where to look for function and variable definitions.
45 *
46 * Put dividers (a single-line comment consisting only of dashes)
47 * between functions.
48 */
49 /-----*/
50 /**
51 * \brief      Use Doxygen documentation for functions.
52 * \param c    Briefly describe all parameters.
53 * \return     Briefly describe the return value.
54 * \retval 0   Functions that return a few specified values
55 * \retval 1   can use the \retval keyword instead of \return.
56 *
57 *      Put a longer description of what the function does
58 *      after the preamble of Doxygen keywords.
59 *
60 *      This template should always be used to document
61 *      functions. The text following the introduction is used
62 *      as the function's documentation.
63 *
64 *      Function prototypes have the return type on one line,
65 *      the name and arguments on one line (with no space

```



```
66 *          between the name and the first parenthesis), followed
67 *          by a single curly bracket on its own line.
68 */
69 void
70 code_style_example_function(void)
71 {
72     /*
73      * Local variables should always be declared at the start of the
74      * function.
75      */
76     int i;          /* Use short variable names for loop
77                      counters. */
78
79     /*
80      * There should be no space between keywords and the first
81      * parenthesis. There should be spaces around binary operators, no
82      * spaces between a unary operator and its operand.
83      *
84      * Curly brackets following for(), if(), do, and case() statements
85      * should follow the statement on the same line.
86      */
87     for(i = 0; i < 10; ++i) {
88         /*
89          * Always use full blocks (curly brackets) after if(), for(), and
90          * while() statements, even though the statement is a single line
91          * of code. This makes the code easier to read and modifications
92          * are less error prone.
93          */
94         if(i == c) {
95             return c;          /* No parenthesis around return values. */
96         } else {              /* The else keyword is placed inbetween
97                               curly brackets, always on its own line. */
98             c++;
99         }
100     }
101 }
102 /*-----*/
103 /*
104  * Static (non-global) functions do not need Doxygen comments. The
105  * name should not be prepended with the module name - doing so would
106  * create confusion.
107  */
108 static void
109 an_example_function(void)
110 {
111
112 }
113 /*-----*/
114
115 /* The following stuff ends the \defgroup block at the beginning of
116    the file: */
117
118 /** @} */
```

9.14 uip-conf.h

```
1 /**
2  * \addtogroup uipopt
3  * @{
4  */
5
6 /**
7  * \name Project-specific configuration options
8  * @{
9  *
10 * uIP has a number of configuration options that can be overridden
11 * for each project. These are kept in a project-specific uip-conf.h
12 * file and all configuration names have the prefix UIP_CONF.
13 */
14
15 /*
16 * Copyright (c) 2006, Swedish Institute of Computer Science.
17 * All rights reserved.
18 *
19 * Redistribution and use in source and binary forms, with or without
20 * modification, are permitted provided that the following conditions
21 * are met:
22 * 1. Redistributions of source code must retain the above copyright
23 *    notice, this list of conditions and the following disclaimer.
24 * 2. Redistributions in binary form must reproduce the above copyright
25 *    notice, this list of conditions and the following disclaimer in the
26 *    documentation and/or other materials provided with the distribution.
27 * 3. Neither the name of the Institute nor the names of its contributors
28 *    may be used to endorse or promote products derived from this software
29 *    without specific prior written permission.
30 *
31 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
32 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
33 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
34 * ARE DISCLAIMED. IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
35 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
36 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
37 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
38 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
39 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
40 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
41 * SUCH DAMAGE.
42 *
43 * This file is part of the uIP TCP/IP stack
44 *
45 * $Id: uip-conf.h,v 1.6 2006/06/12 08:00:31 adam Exp $
46 */
47
48 /**
49 * \file
50 *      An example uIP configuration file
51 * \author
52 *      Adam Dunkels <adam@sics.se>
53 */
54
55 #ifndef __UIP_CONF_H__
56 #define __UIP_CONF_H__
57
58 #include <inttypes.h>
59
60 /**
61 * 8 bit datatype
62 *
63 * This typedef defines the 8-bit type used throughout uIP.
64 *
65 * \hideinitializer
```

```
66 */
67 typedef uint8_t u8_t;
68
69 /**
70  * 16 bit datatype
71  *
72  * This typedef defines the 16-bit type used throughout uIP.
73  *
74  * \hideinitializer
75  */
76 typedef uint16_t u16_t;
77
78 /**
79  * Statistics datatype
80  *
81  * This typedef defines the datatype used for keeping statistics in
82  * uIP.
83  *
84  * \hideinitializer
85  */
86 typedef unsigned short uip_stats_t;
87
88 /**
89  * Maximum number of TCP connections.
90  *
91  * \hideinitializer
92  */
93 #define UIP_CONF_MAX_CONNECTIONS 40
94
95 /**
96  * Maximum number of listening TCP ports.
97  *
98  * \hideinitializer
99  */
100 #define UIP_CONF_MAX_LISTENPORTS 40
101
102 /**
103  * uIP buffer size.
104  *
105  * \hideinitializer
106  */
107 #define UIP_CONF_BUFFER_SIZE 420
108
109 /**
110  * CPU byte order.
111  *
112  * \hideinitializer
113  */
114 #define UIP_CONF_BYTE_ORDER LITTLE_ENDIAN
115
116 /**
117  * Logging on or off
118  *
119  * \hideinitializer
120  */
121 #define UIP_CONF_LOGGING 1
122
123 /**
124  * UDP support on or off
125  *
126  * \hideinitializer
127  */
128 #define UIP_CONF_UDP 0
129
130 /**
131  * UDP checksums on or off
132  *
```

```
133  * \hideinitializer
134  */
135 #define UIP_CONF_UDP_CHECKSUMS    1
136
137 /**
138  * uIP statistics on or off
139  *
140  * \hideinitializer
141  */
142 #define UIP_CONF_STATISTICS        1
143
144 /* Here we include the header file for the application(s) we use in
145    our project. */
146 /*#include "smtp.h"*/
147 /*#include "hello-world.h"*/
148 /*#include "telnetd.h"*/
149 #include "webserver.h"
150 /*#include "dhcpc.h"*/
151 /*#include "resolv.h"*/
152 /*#include "webclient.h"*/
153
154 #endif /* __UIP_CONF_H__ */
155
156 /** @} */
157 /** @} */
```

9.15 webclient.c

```

1 /**
2  * \addtogroup apps
3  * @{
4  */
5
6 /**
7  * \defgroup webclient Web client
8  * @{
9  *
10 * This example shows a HTTP client that is able to download web pages
11 * and files from web servers. It requires a number of callback
12 * functions to be implemented by the module that utilizes the code:
13 * webclient_datahandler(), webclient_connected(),
14 * webclient_timedout(), webclient_aborted(), webclient_closed().
15 */
16
17 /**
18 * \file
19 * Implementation of the HTTP client.
20 * \author Adam Dunkels <adam@dunkels.com>
21 */
22
23 /*
24 * Copyright (c) 2002, Adam Dunkels.
25 * All rights reserved.
26 *
27 * Redistribution and use in source and binary forms, with or without
28 * modification, are permitted provided that the following conditions
29 * are met:
30 * 1. Redistributions of source code must retain the above copyright
31 * notice, this list of conditions and the following disclaimer.
32 * 2. Redistributions in binary form must reproduce the above
33 * copyright notice, this list of conditions and the following
34 * disclaimer in the documentation and/or other materials provided
35 * with the distribution.
36 * 3. The name of the author may not be used to endorse or promote
37 * products derived from this software without specific prior
38 * written permission.
39 *
40 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
41 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
42 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
43 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
44 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
45 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
46 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
47 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
48 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
49 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
50 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
51 *
52 * This file is part of the uIP TCP/IP stack.
53 *
54 * $Id: webclient.c,v 1.2 2006/06/11 21:46:37 adam Exp $
55 *
56 */
57
58 #include "uip.h"
59 #include "uiplib.h"
60 #include "webclient.h"
61 #include "resolv.h"
62
63 #include <string.h>
64
65 #define WEBCLIENT_TIMEOUT 100

```

```

66
67 #define WEBCLIENT_STATE_STATUSLINE 0
68 #define WEBCLIENT_STATE_HEADERS 1
69 #define WEBCLIENT_STATE_DATA 2
70 #define WEBCLIENT_STATE_CLOSE 3
71
72 #define HTTPFLAG_NONE 0
73 #define HTTPFLAG_OK 1
74 #define HTTPFLAG_MOVED 2
75 #define HTTPFLAG_ERROR 3
76
77
78 #define ISO_nl 0x0a
79 #define ISO_cr 0x0d
80 #define ISO_space 0x20
81
82
83 static struct webclient_state s;
84
85 /*-----*/
86 char *
87 webclient_mimetype(void)
88 {
89     return s.mimetype;
90 }
91 /*-----*/
92 char *
93 webclient_filename(void)
94 {
95     return s.file;
96 }
97 /*-----*/
98 char *
99 webclient_hostname(void)
100 {
101     return s.host;
102 }
103 /*-----*/
104 unsigned short
105 webclient_port(void)
106 {
107     return s.port;
108 }
109 /*-----*/
110 void
111 webclient_init(void)
112 {
113
114 }
115 /*-----*/
116 static void
117 init_connection(void)
118 {
119     s.state = WEBCLIENT_STATE_STATUSLINE;
120
121     s.getrequestleft = sizeof(http_get) - 1 + 1 +
122         sizeof(http_10) - 1 +
123         sizeof(http_crnl) - 1 +
124         sizeof(http_host) - 1 +
125         sizeof(http_crnl) - 1 +
126         strlen(http_user_agent_fields) +
127         strlen(s.file) + strlen(s.host);
128     s.getrequestptr = 0;
129
130     s.httpheaderlineptr = 0;
131 }
132 /*-----*/

```

```

133 void
134 webclient_close(void)
135 {
136     s.state = WEBCLIENT_STATE_CLOSE;
137 }
138 /*-----*/
139 unsigned char
140 webclient_get(char *host, u16_t port, char *file)
141 {
142     struct uip_conn *conn;
143     uip_ipaddr_t *ipaddr;
144     static uip_ipaddr_t addr;
145
146     /* First check if the host is an IP address. */
147     ipaddr = &addr;
148     if(uiplib_ipaddrconv(host, (unsigned char *)addr) == 0) {
149         ipaddr = (uip_ipaddr_t *)resolv_lookup(host);
150
151         if(ipaddr == NULL) {
152             return 0;
153         }
154     }
155
156     conn = uip_connect(ipaddr, htons(port));
157
158     if(conn == NULL) {
159         return 0;
160     }
161
162     s.port = port;
163     strncpy(s.file, file, sizeof(s.file));
164     strncpy(s.host, host, sizeof(s.host));
165
166     init_connection();
167     return 1;
168 }
169 /*-----*/
170 static unsigned char *
171 copy_string(unsigned char *dest,
172             const unsigned char *src, unsigned char len)
173 {
174     strncpy(dest, src, len);
175     return dest + len;
176 }
177 /*-----*/
178 static void
179 senddata(void)
180 {
181     u16_t len;
182     char *getrequest;
183     char *cptr;
184
185     if(s.getrequestleft > 0) {
186         cptr = getrequest = (char *)uip_appdata;
187
188         cptr = copy_string(cptr, http_get, sizeof(http_get) - 1);
189         cptr = copy_string(cptr, s.file, strlen(s.file));
190         *cptr++ = ISO_space;
191         cptr = copy_string(cptr, http_10, sizeof(http_10) - 1);
192
193         cptr = copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);
194
195         cptr = copy_string(cptr, http_host, sizeof(http_host) - 1);
196         cptr = copy_string(cptr, s.host, strlen(s.host));
197         cptr = copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);
198
199         cptr = copy_string(cptr, http_user_agent_fields,

```

```

200             strlen(http_user_agent_fields));
201
202     len = s.getrequestleft > uip_mss()?
203         uip_mss():
204         s.getrequestleft;
205     uip_send(&(getrequest[s.getrequestptr]), len);
206 }
207 }
208 /*-----*/
209 static void
210 acked(void)
211 {
212     u16_t len;
213
214     if(s.getrequestleft > 0) {
215         len = s.getrequestleft > uip_mss()?
216             uip_mss():
217             s.getrequestleft;
218         s.getrequestleft -= len;
219         s.getrequestptr += len;
220     }
221 }
222 /*-----*/
223 static u16_t
224 parse_statusline(u16_t len)
225 {
226     char *cptr;
227
228     while(len > 0 && s.httpheaderlineptr < sizeof(s.httpheaderline)) {
229         s.httpheaderline[s.httpheaderlineptr] = *(char *)uip_appdata;
230         ++((char *)uip_appdata);
231         --len;
232         if(s.httpheaderline[s.httpheaderlineptr] == ISO_nl) {
233
234             if((strncmp(s.httpheaderline, http_10,
235                 sizeof(http_10) - 1) == 0) ||
236                 (strncmp(s.httpheaderline, http_11,
237                     sizeof(http_11) - 1) == 0)) {
238                 cptr = &(s.httpheaderline[9]);
239                 s.httpflag = HTTPFLAG_NONE;
240                 if(strncmp(cptr, http_200, sizeof(http_200) - 1) == 0) {
241                     /* 200 OK */
242                     s.httpflag = HTTPFLAG_OK;
243                 } else if(strncmp(cptr, http_301, sizeof(http_301) - 1) == 0 ||
244                     strncmp(cptr, http_302, sizeof(http_302) - 1) == 0) {
245                     /* 301 Moved permanently or 302 Found. Location: header line
246                        will contain thw new location. */
247                     s.httpflag = HTTPFLAG_MOVED;
248                 } else {
249                     s.httpheaderline[s.httpheaderlineptr - 1] = 0;
250                 }
251             } else {
252                 uip_abort();
253                 webclient_aborted();
254                 return 0;
255             }
256
257             /* We're done parsing the status line, so we reset the pointer
258                and start parsing the HTTP headers.*/
259             s.httpheaderlineptr = 0;
260             s.state = WEBCLIENT_STATE_HEADERS;
261             break;
262         } else {
263             ++s.httpheaderlineptr;
264         }
265     }
266     return len;

```



```

267 }
268 /*-----*/
269 static char
270 casecmp(char *str1, const char *str2, char len)
271 {
272     static char c;
273
274     while(len > 0) {
275         c = *str1;
276         /* Force lower-case characters. */
277         if(c & 0x40) {
278             c |= 0x20;
279         }
280         if(*str2 != c) {
281             return 1;
282         }
283         ++str1;
284         ++str2;
285         --len;
286     }
287     return 0;
288 }
289 /*-----*/
290 static u16_t
291 parse_headers(u16_t len)
292 {
293     char *cptr;
294     static unsigned char i;
295
296     while(len > 0 && s.httpheaderlineptr < sizeof(s.httpheaderline)) {
297         s.httpheaderline[s.httpheaderlineptr] = *(char *)uip_appdata;
298         ++((char *)uip_appdata);
299         --len;
300         if(s.httpheaderline[s.httpheaderlineptr] == ISO_nl) {
301             /* We have an entire HTTP header line in s.httpheaderline, so
302              * we parse it. */
303             if(s.httpheaderline[0] == ISO_cr) {
304                 /* This was the last header line (i.e., and empty "\r\n"), so
305                  * we are done with the headers and proceed with the actual
306                  * data. */
307                 s.state = WEBCLIENT_STATE_DATA;
308                 return len;
309             }
310
311             s.httpheaderline[s.httpheaderlineptr - 1] = 0;
312             /* Check for specific HTTP header fields. */
313             if(casecmp(s.httpheaderline, http_content_type,
314                 sizeof(http_content_type) - 1) == 0) {
315                 /* Found Content-type field. */
316                 cptr = strchr(s.httpheaderline, ';');
317                 if(cptr != NULL) {
318                     *cptr = 0;
319                 }
320                 strncpy(s.mimetype, s.httpheaderline +
321                     sizeof(http_content_type) - 1, sizeof(s.mimetype));
322             } else if(casecmp(s.httpheaderline, http_location,
323                 sizeof(http_location) - 1) == 0) {
324                 cptr = s.httpheaderline +
325                     sizeof(http_location) - 1;
326
327                 if(strncmp(cptr, http_http, 7) == 0) {
328                     cptr += 7;
329                     for(i = 0; i < s.httpheaderlineptr - 7; ++i) {
330                         if(*cptr == 0 ||
331                             *cptr == '/' ||
332                             *cptr == ' ' ||
333                             *cptr == ':') {

```

```
334         s.host[i] = 0;
335         break;
336     }
337     s.host[i] = *cptr;
338     ++cptr;
339 }
340 }
341 strncpy(s.file, cptr, sizeof(s.file));
342 /*      s.file[s.httpheaderlineptr - i] = 0;*/
343 }
344
345
346     /* We're done parsing, so we reset the pointer and start the
347     next line. */
348     s.httpheaderlineptr = 0;
349 } else {
350     ++s.httpheaderlineptr;
351 }
352 }
353 return len;
354 }
355 /*-----*/
356 static void
357 newdata(void)
358 {
359     u16_t len;
360
361     len = uip_datalen();
362
363     if(s.state == WEBCLIENT_STATE_STATUSLINE) {
364         len = parse_statusline(len);
365     }
366
367     if(s.state == WEBCLIENT_STATE_HEADERS && len > 0) {
368         len = parse_headers(len);
369     }
370
371     if(len > 0 && s.state == WEBCLIENT_STATE_DATA &&
372        s.httpflag != HTTPFLAG_MOVED) {
373         webclient_datahandler((char *)uip_appdata, len);
374     }
375 }
376 /*-----*/
377 void
378 webclient_appcall(void)
379 {
380     if(uip_connected()) {
381         s.timer = 0;
382         s.state = WEBCLIENT_STATE_STATUSLINE;
383         senddata();
384         webclient_connected();
385         return;
386     }
387
388     if(s.state == WEBCLIENT_STATE_CLOSE) {
389         webclient_closed();
390         uip_abort();
391         return;
392     }
393
394     if(uip_aborted()) {
395         webclient_aborted();
396     }
397     if(uip_timedout()) {
398         webclient_timedout();
399     }
400 }
```

```
401
402     if(uiplib_acked()) {
403         s.timer = 0;
404         acked();
405     }
406     if(uiplib_newdata()) {
407         s.timer = 0;
408         newdata();
409     }
410     if(uiplib_rexmit() ||
411         uilib_newdata() ||
412         uilib_acked()) {
413         senddata();
414     } else if(uiplib_poll()) {
415         ++s.timer;
416         if(s.timer == WEBCLIENT_TIMEOUT) {
417             webclient_timedout();
418             uilib_abort();
419             return;
420         }
421         /*      senddata();*/
422     }
423
424     if(uiplib_closed()) {
425         if(s.httpflag != HTTPFLAG_MOVED) {
426             /* Send NULL data to signal EOF. */
427             webclient_datahandler(NULL, 0);
428         } else {
429             if(resolv_lookup(s.host) == NULL) {
430                 resolv_query(s.host);
431             }
432             webclient_get(s.host, s.port, s.file);
433         }
434     }
435 }
436 /*-----*/
437
438 /** @} */
439 /** @} */
```

9.16 webclient.h

```
1 /**
2  * \addtogroup webclient
3  * @{
4  */
5
6 /**
7  * \file
8  * Header file for the HTTP client.
9  * \author Adam Dunkels <adam@dunkels.com>
10 */
11
12 /*
13  * Copyright (c) 2002, Adam Dunkels.
14  * All rights reserved.
15  *
16  * Redistribution and use in source and binary forms, with or without
17  * modification, are permitted provided that the following conditions
18  * are met:
19  * 1. Redistributions of source code must retain the above copyright
20  *    notice, this list of conditions and the following disclaimer.
21  * 2. Redistributions in binary form must reproduce the above
22  *    copyright notice, this list of conditions and the following
23  *    disclaimer in the documentation and/or other materials provided
24  *    with the distribution.
25  * 3. The name of the author may not be used to endorse or promote
26  *    products derived from this software without specific prior
27  *    written permission.
28  *
29  * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
30  * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
31  * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
32  * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
33  * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
34  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
35  * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
36  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
37  * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
38  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
39  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
40  *
41  * This file is part of the uIP TCP/IP stack.
42  *
43  * $Id: webclient.h,v 1.2 2006/06/11 21:46:37 adam Exp $
44  */
45
46 #ifndef __WEBCLIENT_H__
47 #define __WEBCLIENT_H__
48
49
50 #include "webclient-strings.h"
51 #include "uiptopt.h"
52
53 #define WEBCLIENT_CONF_MAX_URLLEN 100
54
55 struct webclient_state {
56     u8_t timer;
57     u8_t state;
58     u8_t httpflag;
59
60     u16_t port;
61     char host[40];
62     char file[WEBCLIENT_CONF_MAX_URLLEN];
63     u16_t getrequestptr;
64     u16_t getrequestleft;
65 }
```

```
66  char httpheaderline[200];
67  u16_t httpheaderlineptr;
68
69  char mimetype[32];
70 };
71
72 typedef struct webclient_state uip_tcp_appstate_t;
73 #define UIP_APPCALL webclient_appcall
74
75 /**
76  * Callback function that is called from the webclient code when HTTP
77  * data has been received.
78  *
79  * This function must be implemented by the module that uses the
80  * webclient code. The function is called from the webclient module
81  * when HTTP data has been received. The function is not called when
82  * HTTP headers are received, only for the actual data.
83  *
84  * \note This function is called many times, repeatedly, when data is
85  * being received, and not once when all data has been received.
86  *
87  * \param data A pointer to the data that has been received.
88  * \param len The length of the data that has been received.
89  */
90 void webclient_datahandler(char *data, u16_t len);
91
92 /**
93  * Callback function that is called from the webclient code when the
94  * HTTP connection has been connected to the web server.
95  *
96  * This function must be implemented by the module that uses the
97  * webclient code.
98  */
99 void webclient_connected(void);
100
101 /**
102  * Callback function that is called from the webclient code if the
103  * HTTP connection to the web server has timed out.
104  *
105  * This function must be implemented by the module that uses the
106  * webclient code.
107  */
108 void webclient_timedout(void);
109
110 /**
111  * Callback function that is called from the webclient code if the
112  * HTTP connection to the web server has been aborted by the web
113  * server.
114  *
115  * This function must be implemented by the module that uses the
116  * webclient code.
117  */
118 void webclient_aborted(void);
119
120 /**
121  * Callback function that is called from the webclient code when the
122  * HTTP connection to the web server has been closed.
123  *
124  * This function must be implemented by the module that uses the
125  * webclient code.
126  */
127 void webclient_closed(void);
128
129
130
131 /**
132  * Initialize the webclient module.
```

```
133 */
134 void webclient_init(void);
135
136 /**
137  * Open an HTTP connection to a web server and ask for a file using
138  * the GET method.
139  *
140  * This function opens an HTTP connection to the specified web server
141  * and requests the specified file using the GET method. When the HTTP
142  * connection has been connected, the webclient_connected() callback
143  * function is called and when the HTTP data arrives the
144  * webclient_datahandler() callback function is called.
145  *
146  * The callback function webclient_timedout() is called if the web
147  * server could not be contacted, and the webclient_aborted() callback
148  * function is called if the HTTP connection is aborted by the web
149  * server.
150  *
151  * When the HTTP request has been completed and the HTTP connection is
152  * closed, the webclient_closed() callback function will be called.
153  *
154  * \note If the function is passed a host name, it must already be in
155  * the resolver cache in order for the function to connect to the web
156  * server. It is therefore up to the calling module to implement the
157  * resolver calls and the signal handler used for reporting a resolv
158  * query answer.
159  *
160  * \param host A pointer to a string containing either a host name or
161  * a numerical IP address in dotted decimal notation (e.g., 192.168.23.1).
162  *
163  * \param port The port number to which to connect, in host byte order.
164  *
165  * \param file A pointer to the name of the file to get.
166  *
167  * \retval 0 if the host name could not be found in the cache, or
168  * if a TCP connection could not be created.
169  *
170  * \retval 1 if the connection was initiated.
171  */
172 unsigned char webclient_get(char *host, u16_t port, char *file);
173
174 /**
175  * Close the currently open HTTP connection.
176  */
177 void webclient_close(void);
178 void webclient_appcall(void);
179
180 /**
181  * Obtain the MIME type of the current HTTP data stream.
182  *
183  * \return A pointer to a string containing the MIME type. The string
184  * may be empty if no MIME type was reported by the web server.
185  */
186 char *webclient_mimetype(void);
187
188 /**
189  * Obtain the filename of the current HTTP data stream.
190  *
191  * The filename of an HTTP request may be changed by the web server,
192  * and may therefore not be the same as when the original GET request
193  * was made with webclient_get(). This function is used for obtaining
194  * the current filename.
195  *
196  * \return A pointer to the current filename.
197  */
198 char *webclient_filename(void);
199
```

```
200 /**
201  * Obtain the hostname of the current HTTP data stream.
202  *
203  * The hostname of the web server of an HTTP request may be changed
204  * by the web server, and may therefore not be the same as when the
205  * original GET request was made with webclient_get(). This function
206  * is used for obtaining the current hostname.
207  *
208  * \return A pointer to the current hostname.
209  */
210 char *webclient_hostname(void);
211
212 /**
213  * Obtain the port number of the current HTTP data stream.
214  *
215  * The port number of an HTTP request may be changed by the web
216  * server, and may therefore not be the same as when the original GET
217  * request was made with webclient_get(). This function is used for
218  * obtaining the current port number.
219  *
220  * \return The port number of the current HTTP data stream, in host byte order.
221  */
222 unsigned short webclient_port(void);
223
224
225
226 #endif /* __WEBCCLIENT_H__ */
227
228 /** @} */
```

Index

- Applications, 36
- apps/hello-world/hello-world.c, 147
- apps/hello-world/hello-world.h, 148
- apps/resolv/resolv.c, 149
- apps/resolv/resolv.h, 151
- apps/smtp/smtp.c, 153
- apps/smtp/smtp.h, 154
- apps/telnetd/shell.c, 156
- apps/telnetd/shell.h, 157
- apps/telnetd/telnetd.c, 158
- apps/telnetd/telnetd.h, 159
- apps/webclient/webclient.c, 160
- apps/webclient/webclient.h, 162
- apps/webserver/httpd-cgi.c, 164
- apps/webserver/httpd-cgi.h, 165
- apps/webserver/httpd.c, 166
- Architecture specific uIP functions, 74
- clock
 - clock_init, 94
 - clock_time, 94
- Clock interface, 94
- clock_init
 - clock, 94
- clock_time
 - clock, 94
- Configuration options for uIP, 79
- dhcpc_state, 123
- DNS resolver, 105
- Hello, world, 113
- hello_world_state, 124
- HTONS
 - uipconvfunc, 56
- htons
 - uip, 67
 - uipconvfunc, 60
- httpd
 - HTTPD_CGI_CALL, 121
 - httpd_init, 121
- HTTPD_CGI_CALL
 - httpd, 121
- httpd_cgi_call, 125
- httpd_init
 - httpd, 121
- httpd_state, 126
- lib/memb.c, 167
- lib/memb.h, 168
- Local continuations, 90
- MEMB
 - memb, 103
- memb
 - MEMB, 103
 - memb_alloc, 103
 - memb_free, 103
 - memb_init, 104
- memb_alloc
 - memb, 103
- memb_blocks, 127
- memb_free
 - memb, 103
- memb_init
 - memb, 104
- Memory block management functions, 102
- Protosockets library, 95
- Protothreads, 27
- psock, 128
 - PSOCK_BEGIN, 96
 - PSOCK_CLOSE, 97
 - PSOCK_CLOSE_EXIT, 97
 - PSOCK_DATALEN, 97
 - PSOCK_END, 97
 - PSOCK_EXIT, 98
 - PSOCK_GENERATOR_SEND, 98
 - PSOCK_INIT, 98
 - PSOCK_NEWDATA, 99
 - PSOCK_READBUF, 99
 - PSOCK_READTO, 99
 - PSOCK_SEND, 99
 - PSOCK_SEND_STR, 100
 - PSOCK_WAIT_UNTIL, 100
- PSOCK_BEGIN
 - psock, 96
- psock_buf, 129
- PSOCK_CLOSE
 - psock, 97

- PSOCK_CLOSE_EXIT
 - psock, 97
- PSOCK_DATALEN
 - psock, 97
- PSOCK_END
 - psock, 97
- PSOCK_EXIT
 - psock, 98
- PSOCK_GENERATOR_SEND
 - psock, 98
- PSOCK_INIT
 - psock, 98
- PSOCK_NEWDATA
 - psock, 99
- PSOCK_READBUF
 - psock, 99
- PSOCK_READTO
 - psock, 99
- PSOCK_SEND
 - psock, 99
- PSOCK_SEND_STR
 - psock, 100
- PSOCK_WAIT_UNTIL
 - psock, 100
- pt, 130
 - PT_BEGIN, 31
 - PT_END, 31
 - PT_EXIT, 32
 - PT_INIT, 32
 - PT_RESTART, 32
 - PT_SCHEDULE, 33
 - PT_SPAWN, 33
 - PT_THREAD, 33
 - PT_WAIT_THREAD, 33
 - PT_WAIT_UNTIL, 34
 - PT_WAIT_WHILE, 34
 - PT_YIELD, 34
 - PT_YIELD_UNTIL, 35
- PT_BEGIN
 - pt, 31
- PT_END
 - pt, 31
- PT_EXIT
 - pt, 32
- PT_INIT
 - pt, 32
- PT_RESTART
 - pt, 32
- PT_SCHEDULE
 - pt, 33
- PT_SPAWN
 - pt, 33
- PT_THREAD
 - pt, 33
- PT_WAIT_THREAD
 - pt, 33
- PT_WAIT_UNTIL
 - pt, 34
- PT_WAIT_WHILE
 - pt, 34
- PT_YIELD
 - pt, 34
- PT_YIELD_UNTIL
 - pt, 35
- resolv
 - resolv_conf, 106
 - resolv_found, 106
 - resolv_getserver, 106
 - resolv_lookup, 106
 - resolv_query, 107
- resolv_conf
 - resolv, 106
- resolv_found
 - resolv, 106
- resolv_getserver
 - resolv, 106
- resolv_lookup
 - resolv, 106
- resolv_query
 - resolv, 107
- shell_init
 - telnetd, 111
- shell_input
 - telnetd, 111
- shell_output
 - telnetd, 111
- shell_prompt
 - telnetd, 112
- shell_start
 - telnetd, 112
- smtp
 - smtp_configure, 109
 - smtp_done, 109
 - smtp_send, 109
- SMTP E-mail sender, 108
- smtp_configure
 - smtp, 109
- smtp_done
 - smtp, 109
- smtp_send
 - smtp, 109
- smtp_state, 131
- Telnet server, 110
- telnetd
 - shell_init, 111

- shell_input, 111
 - shell_output, 111
 - shell_prompt, 112
 - shell_start, 112
- telnetd_state, 132
- The uIP TCP/IP stack, 62
- timer, 133
 - timer_expired, 92
 - timer_reset, 92
 - timer_restart, 92
 - timer_set, 93
- Timer library, 91
- timer_expired
 - timer, 92
- timer_reset
 - timer, 92
- timer_restart
 - timer, 92
- timer_set
 - timer, 93
- u16_t
 - uiptopt, 87
- u8_t
 - uiptopt, 87
- uip
 - htons, 67
 - uip_add32, 67
 - uip_appdata, 71
 - UIP_APPDATA_SIZE, 66
 - uip_buf, 71
 - uip_chksum, 67
 - uip_conn, 72
 - uip_connect, 67
 - uip_init, 68
 - uip_ipchksum, 68
 - uip_len, 72
 - uip_listen, 68
 - uip_send, 69
 - uip_setipid, 69
 - uip_stat, 72
 - uip_tcpchksum, 69
 - uip_udp_new, 70
 - uip_udpchksum, 70
 - uip_unlisten, 70
- uIP Address Resolution Protocol, 76
- uIP application functions, 46
- uIP configuration functions, 37
- uIP conversion functions, 55
- uIP device driver functions, 41
- uIP initialization functions, 40
- uIP TCP throughput booster hack, 89
- uip/lc-addrlabels.h, 169
- uip/lc-switch.h, 170
- uip/lc.h, 171
- uip/psock.h, 172
- uip/pt.h, 174
- uip/timer.c, 176
- uip/timer.h, 177
- uip/uip-neighbor.c, 178
- uip/uip-neighbor.h, 179
- uip/uip-split.h, 180
- uip/uip.c, 181
- uip/uip.h, 184
- uip/uip_arch.h, 190
- uip/uip_arp.c, 191
- uip/uip_arp.h, 192
- uip/uipopt.h, 193
- uip_abort
 - uipappfunc, 47
- uip_aborted
 - uipappfunc, 48
- uip_acked
 - uipappfunc, 48
- UIP_ACTIVE_OPEN
 - uiptopt, 83
- uip_add32
 - uip, 67
 - uiparch, 74
- uip_appdata
 - uip, 71
- UIP_APPDATA_SIZE
 - uip, 66
- uip_arp_arpin
 - uiparp, 77
- UIP_ARP_MAXAGE
 - uiptopt, 83
- uip_arp_out
 - uiparp, 77
- uip_arp_timer
 - uiparp, 78
- UIP_ARPTAB_SIZE
 - uiptopt, 83
- UIP_BROADCAST
 - uiptopt, 83
- uip_buf
 - uip, 71
 - uipdevfunc, 44
- UIP_BUFSIZE
 - uiptopt, 83
- UIP_BYTE_ORDER
 - uiptopt, 84
- uip_chksum
 - uip, 67
 - uiparch, 75
- uip_close
 - uipappfunc, 48
- uip_closed

- uipappfunc, 48
- uip_conn, 134
 - uip, 72
- uip_connect
 - uip, 67
 - uipappfunc, 52
- uip_connected
 - uipappfunc, 48
- UIP_CONNS
 - uiptopt, 84
- uip_datalen
 - uipappfunc, 49
- uip_eth_addr, 136
- uip_eth_hdr, 137
- UIP_FIXEDADDR
 - uiptopt, 84
- UIP_FIXEETHADDR
 - uiptopt, 84
- uip_getdraddr
 - uipconffunc, 37
- uip_gethostaddr
 - uipconffunc, 37
- uip_getnetmask
 - uipconffunc, 38
- uip_icmpip_hdr, 138
- uip_init
 - uip, 68
 - uipinit, 40
- uip_input
 - uipdevfunc, 41
- uip_ip6addr
 - uipconvfunc, 56
- uip_ipaddr
 - uipconvfunc, 56
- uip_ipaddr1
 - uipconvfunc, 56
- uip_ipaddr2
 - uipconvfunc, 57
- uip_ipaddr3
 - uipconvfunc, 57
- uip_ipaddr4
 - uipconvfunc, 58
- uip_ipaddr_cmp
 - uipconvfunc, 58
- uip_ipaddr_copy
 - uipconvfunc, 58
- uip_ipaddr_mask
 - uipconvfunc, 59
- uip_ipaddr_maskcmp
 - uipconvfunc, 59
- uip_ipchksum
 - uip, 68
 - uiparch, 75
- uip_len
 - uip, 72
 - uipdrivervars, 61
- uip_listen
 - uip, 68
 - uipappfunc, 52
- UIP_LISTENPORTS
 - uiptopt, 84
- UIP_LLH_LEN
 - uiptopt, 85
- uip_log
 - uiptopt, 88
- UIP_LOGGING
 - uiptopt, 85
- UIP_MAXRTX
 - uiptopt, 85
- UIP_MAXSYNRTX
 - uiptopt, 85
- uip_mss
 - uipappfunc, 49
- uip_neighbor_addr, 139
- uip_newdata
 - uipappfunc, 49
- uip_periodic
 - uipdevfunc, 42
- uip_periodic_conn
 - uipdevfunc, 43
- UIP_PINGADDRCONF
 - uiptopt, 85
- uip_poll
 - uipappfunc, 49
- uip_poll_conn
 - uipdevfunc, 43
- UIP_REASSEMBLY
 - uiptopt, 85
- UIP_RECEIVE_WINDOW
 - uiptopt, 86
- uip_restart
 - uipappfunc, 50
- uip_rexmit
 - uipappfunc, 50
- UIP_RTO
 - uiptopt, 86
- uip_send
 - uip, 69
 - uipappfunc, 53
- uip_setdraddr
 - uipconffunc, 38
- uip_setethaddr
 - uipconffunc, 38
- uip_sethostaddr
 - uipconffunc, 38
- uip_setipid
 - uip, 69
 - uipinit, 40

- uip_setnetmask
 - uipconffunc, 39
- uip_split_output
 - uipsplit, 89
- uip_stat
 - uip, 72
- UIP_STATISTICS
 - uiptopt, 86
- uip_stats, 140
- uip_stats_t
 - uiptopt, 87
- uip_stop
 - uipappfunc, 50
- uip_tcp_appstate_t
 - uiptopt, 87
- UIP_TCP_MSS
 - uiptopt, 86
- uip_tcpchksum
 - uip, 69
 - uiparch, 75
- uip_tcpip_hdr, 142
- UIP_TIME_WAIT_TIMEOUT
 - uiptopt, 86
- uip_timedout
 - uipappfunc, 50
- UIP_TTL
 - uiptopt, 86
- uip_udp_appstate_t
 - uiptopt, 88
- uip_udp_bind
 - uipappfunc, 50
- UIP_UDP_CHECKSUMS
 - uiptopt, 87
- uip_udp_conn, 143
- uip_udp_new
 - uip, 70
 - uipappfunc, 53
- uip_udp_periodic
 - uipdevfunc, 43
- uip_udp_periodic_conn
 - uipdevfunc, 44
- uip_udp_remove
 - uipappfunc, 51
- uip_udp_send
 - uipappfunc, 51
- uip_udpchksum
 - uip, 70
- uip_udpconnection
 - uipappfunc, 51
- uip_udpip_hdr, 144
- uip_unlisten
 - uip, 70
 - uipappfunc, 54
- UIP_URGDATA
 - uiptopt, 87
- uip_urgdatalen
 - uipappfunc, 51
- uipappfunc
 - uip_abort, 47
 - uip_aborted, 48
 - uip_acked, 48
 - uip_close, 48
 - uip_closed, 48
 - uip_connect, 52
 - uip_connected, 48
 - uip_datalen, 49
 - uip_listen, 52
 - uip_mss, 49
 - uip_newdata, 49
 - uip_poll, 49
 - uip_restart, 50
 - uip_rexmit, 50
 - uip_send, 53
 - uip_stop, 50
 - uip_timedout, 50
 - uip_udp_bind, 50
 - uip_udp_new, 53
 - uip_udp_remove, 51
 - uip_udp_send, 51
 - uip_udpconnection, 51
 - uip_unlisten, 54
 - uip_urgdatalen, 51
- uiparch
 - uip_add32, 74
 - uip_chksum, 75
 - uip_ipchksum, 75
 - uip_tcpchksum, 75
- uiparp
 - uip_arp_arpin, 77
 - uip_arp_out, 77
 - uip_arp_timer, 78
- uipconffunc
 - uip_getdraddr, 37
 - uip_gethostaddr, 37
 - uip_getnetmask, 37
 - uip_setdraddr, 38
 - uip_setethaddr, 38
 - uip_sethostaddr, 38
 - uip_setnetmask, 39
- uipconvfunc
 - HTONS, 56
 - htons, 60
 - uip_ip6addr, 56
 - uip_ipaddr, 56
 - uip_ipaddr1, 56
 - uip_ipaddr2, 57
 - uip_ipaddr3, 57
 - uip_ipaddr4, 58

- uip_ipaddr_cmp, 58
- uip_ipaddr_copy, 58
- uip_ipaddr_mask, 59
- uip_ipaddr_maskcmp, 59
- uipdevfunc
 - uip_buf, 44
 - uip_input, 41
 - uip_periodic, 42
 - uip_periodic_conn, 43
 - uip_poll_conn, 43
 - uip_udp_periodic, 43
 - uip_udp_periodic_conn, 44
- uipdrivervars
 - uip_len, 61
- uipinit
 - uip_init, 40
 - uip_setipid, 40
- uipopt
 - u16_t, 87
 - u8_t, 87
 - UIP_ACTIVE_OPEN, 83
 - UIP_ARP_MAXAGE, 83
 - UIP_ARPTAB_SIZE, 83
 - UIP_BROADCAST, 83
 - UIP_BUFSIZE, 83
 - UIP_BYTE_ORDER, 84
 - UIP_CONNS, 84
 - UIP_FIXEDADDR, 84
 - UIP_FIXEDETHADDR, 84
 - UIP_LISTENPORTS, 84
 - UIP_LLH_LEN, 85
 - uip_log, 88
 - UIP_LOGGING, 85
 - UIP_MAXRTX, 85
 - UIP_MAXSYNRTX, 85
 - UIP_PINGADDRCONF, 85
 - UIP_REASSEMBLY, 85
 - UIP_RECEIVE_WINDOW, 86
 - UIP_RTO, 86
 - UIP_STATISTICS, 86
 - uip_stats_t, 87
 - uip_tcp_appstate_t, 87
 - UIP_TCP_MSS, 86
 - UIP_TIME_WAIT_TIMEOUT, 86
 - UIP_TTL, 86
 - uip_udp_appstate_t, 88
 - UIP_UDP_CHECKSUMS, 87
 - UIP_URGDATA, 87
- uipsplit
 - uip_split_output, 89
- unix/uip-conf.h, 196
- Web client, 114
- Web server, 120
- webclient
 - webclient_aborted, 115
 - webclient_closed, 116
 - webclient_connected, 116
 - webclient_datahandler, 116
 - webclient_filename, 116
 - webclient_get, 117
 - webclient_hostname, 117
 - webclient_mimetype, 118
 - webclient_port, 118
 - webclient_timedout, 118
- webclient_aborted
 - webclient, 115
- webclient_closed
 - webclient, 116
- webclient_connected
 - webclient, 116
- webclient_datahandler
 - webclient, 116
- webclient_filename
 - webclient, 116
- webclient_get
 - webclient, 117
- webclient_hostname
 - webclient, 117
- webclient_mimetype
 - webclient, 118
- webclient_port
 - webclient, 118
- webclient_state, 145
- webclient_timedout
 - webclient, 118

Variables used in uIP device drivers, 61