Contiki 2.x Reference Manual

Generated by Doxygen 1.4.1

Mon Jul 2 14:14:41 2007

Contents

1 The Contiki Operating System 2.x

Author:
Adam Dunkels <adam@sics.se>

Contiki is an open source, highly portable, multi-tasking operating system for memory-constrained networked embedded systems written by Adam Dunkels at the Networked Embedded Systems group at the Swedish Institute of Computer Science.

Contiki is designed for embedded systems with small amounts of memory. A typical Contiki configuration is 2 kilobytes of RAM and 40 kilobytes of ROM. Contiki consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime. Contiki processes use light-weight protothreads that provide a linear, thread-like programming style on top of the event-driven kernel. Contiki also supports per-process optional preemptive multi-threading, interprocess communication using message passing through events, as well as an optional GUI subsystem with either direct graphic support for locally connected terminals or networked virtual display with VNC or over Telnet.

Contiki contains two communication stacks: uIP and Rime. uIP is a small RFC-compliant TCP/IP stack that makes it possible for Contiki to communicate over the Internet. Rime is a lightweight communication stack designed for low-power radios. Rime provides a wide range of communication primitives, from best-effort local area broadcast, to reliable multi-hop bulk data flooding.

Contiki runs on a variety of platform ranging from embedded microcontrollers such as the MSP430 and the AVR to old homecomputers. Code footprint is on the order of kilobytes and memory usage can be configured to be as low as tens of bytes.

Contiki is written in the C programming language and is freely available as open source under a BSD-style license. More information about Contiki can be found at the Contiki home page: http://www.sics.se/contiki/

1.1 TCP/IP

Contiki includes the uIP TCP/IP stack (`http://www.sics.se/~adam/uip/`) that provides Contiki with TCP/IP networking support. uIP provides the protocols TCP, UDP, IP, and ARP.

See also:

The uIP TCP/IP stack documentation
The Contiki/uIP interface
Protosockets library

1.2 Rime

Rime is a lightweight communication stacks designed for low-power radios. Rime provides a wide range of communication primitives suitable for implementing communication-bound applications or network protocols.

See also:

The Rime Communication Stack

1.3 Multi-threading and protothreads

Contiki is based on an event-driven kernel but provides support for both multi-threading and a lightweight stackless thread-like construct called protothreads.

See also:

Contiki processes
Protothreads
Event timers
Optional multi-threading

1.4 Libraries

Contiki provides a set of convenience libraries for memory management and linked list operations.

See also:

Simple timer library
Memory block management
Linked list library

1.5 Getting started with Contiki

Contiki is designed to run on many different platforms. It is also possible to compile and build both the Contiki system and Contiki applications on many different development platforms.

See Getting started with Contiki for the ESB platform

1.6 Building the Contiki system and its applications

The Contiki build system is designed to make it easy to compile Contiki applications for either to a hardware platform or into a simulation platform by simply supplying different parameters to the `make` command, without having to edit makefiles or modify the application code.

See The Contiki build system

2 Contiki 2.x Module Index

2.1 Contiki 2.x Modules

Here is a list of all modules:

3 Contiki 2.x Directory Hierarchy

3.1 Contiki 2.x Directories

This directory hierarchy is sorted roughly, but not completely, alphabetically:

4 Contiki 2.x Data Structure Index

4.1 Contiki 2.x Data Structures

Here are the data structures with brief descriptions:

5 Contiki 2.x File Index

5.1 Contiki 2.x File List

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

6 Contiki 2.x Module Documentation

6.1 Communication stacks

Modules

- The uIP TCP/IP stack
- The Rime communication stack

6.2 Device driver APIs

Modules

- EEPROM API

  The EEPROM API defines a common interface for EEPROM access on Contiki platforms.

- Radio API

  The radio API module defines a set of functions that a radio device driver must implement.

6.3 Memory functions

Modules

- Memory block management functions

  The memory block allocation routines provide a simple yet powerful set of functions for managing a set of memory blocks of fixed size.

- Managed memory allocator

  The managed memory allocator is a fragmentation-free memory manager.

6.4 Contiki system

Modules

- Contiki processes

  A process in Contiki consists of a single protothread.

- Event timers

  Event timers provides a way to generate timed events.

- Argument buffer
- The Contiki program loader

  The Contiki program loader is an abstract interface for loading and starting programs.

- Clock library

  The clock library is the interface between Contiki and the platform specific clock functionality.

- Multi-threading library

  The event driven Contiki kernel does not provide multi-threading by itself - instead, preemptive multi-threading is implemented as a library that optionally can be linked with applications.

- Protothreads

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

- The Contiki file system interface

The Contiki file system interface (CFS) defines an abstract API for reading directories and for reading and writing files.

- Timer library

  The Contiki kernel does not provide support for timed events.

6.5 Libraries

Modules

- Linked list library

  The linked list library provides a set of functions for manipulating linked lists.

- Table-driven Manchester encoding and decoding

  Manchester encoding is a bit encoding scheme which translates each bit into two bits: the original bit and the inverted bit.

- Cyclic Redundancy Check 16 (CRC16) calculcation

  The Cyclic Redundancy Check 16 is a hash function that produces a checksum that is used to detect errors in transmissions.

6.6 Contiki platforms

6.6.1 Detailed Description

*

*

Modules

- The Tmote Sky Board

  The Tmote Sky platform is a wireless sensor board from Moteiv.

- The ESB Embedded Sensor Board

  The ESB (Embedded Sensor Board) is a prototype wireless sensor network device developed at Freie Universität Berlin.

- Microsoft Windows

  It is possible to run an entire Contiki system as a program under Microsoft Windows.

6.7 The uIP TCP/IP stack

6.7.1 Detailed Description

The uIP TCP/IP stack provides Internet communication abilities to Contiki.

6.7.2 uIP introduction

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:

The Contiki/uIP interface
uIP Compile-time configuration options
uIP Run-time configuration functions
uIP initialization functions
uIP device driver interface and uIP variables used by device drivers
uIP functions called from application programs (see below) and the protosockets API and their underlying protothreads

6.7.3 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.

6.7.4 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.

6.7.5 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 called to send out the packets that may have been produced.

6.7.6 Architecture Specific Functions

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

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

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.

6.7.6.2 32-bit Arithmetic The TCP protocol uses 32-bit sequence numbers, and a TCP implementation will have to do a number of 32-bit additions as part of the normal protocol processing. Since 32-bit arithmetic is not natively available on many of the platforms for which uIP is intended, uIP leaves the 32-bit additions to be implemented by the architecture specific module and does not make use of any 32-bit arithmetic in the main code base.

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

6.7.7 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.

6.7.8 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 nore low-level than protosockets but uses less memory.

See also:
Protosockets library
Protothreads

6.7.8.1 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 retransmissions. Other TCP/IP stacks buffer the transmitted data in memory until the data is known to be successfully delivered to the remote end of the connection. If the data needs to be retransmitted, the stack takes care of the retransmission without notifying the application. With this approach, the data has to be buffered in memory while waiting for an acknowledgment even if the application might be able to quickly regenerate the data if a retransmission has to be made.

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

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).

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.

Receiving Data If the uIP test function uip_newdata() is non-zero, the remote host of the connection has sent new data. The uip_appdata pointer point to the actual data. The size of the data is obtained through the uIP function uip_datalen(). The data is not buffered by uIP, but will be overwritten after the application function returns, and the application will therefor have to either act directly on the incoming data, or by itself copy the incoming data into a buffer for later processing.

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.

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

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

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.

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.

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.

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.

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));
}
```

6.7.9 Examples

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

6.7.9.1 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().

6.7.9.2 A More Advanced Application This second example is slightly more advanced than the previous one, and shows how the application state field in the uip_conn structure is used.

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

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

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

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

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

The implementation of this application is seen below. This configuration settings for the application is follows after its implementation.

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

void example2_init(void) {
   uip_listen(HTONS(2345));
```

```
}

void example2_app(void) {
   struct example2_state *s;

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

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

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

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

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

The configuration for the application:

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

6.7.9.3 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;
   }
}
```

6.7.9.4 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().

6.7.9.5 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().

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

The uIP event handler function is shown below.

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

```
      uip_acked() ||
      uip_connected() ||
      uip_poll()) {
    senddata();
  }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

```
      uip_close();
      break;
    }
  }
}

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

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

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

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

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

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

The acked() function is called whenever data that previously was sent has been acknowleged by the receiving host. This acked() function first reduces the amount of data that is left to send, by subtracting the length of the previously sent data (obtained from "uip_conn $\rightarrow$ 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.

6.7.10 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.

6.7.10.1 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.

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.

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.

6.7.10.2 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 itis 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.

6.7.10.3 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.

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.

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.

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.

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.

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.

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.

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.

6.7.11 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.

6.7.11.1 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 acknowledgmen t 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.

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
- uIP initialization functions
- uIP device driver functions
- uIP application functions
- uIP conversion functions
- Variables used in uIP device drivers
- Configuration options for uIP
- uIP Address Resolution Protocol
- uIP TCP throughput booster hack
- uIP packet forwarding
- uIP hostname resolver functions
- Protosockets library
- The Contiki/uIP interface
- Uiparch

Data Structures

- union uip_ip4addr_t

  Representation of an IP address.

- 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_eth_addr

  Representation of a 48-bit Ethernet address.

Defines

- #define UIP_APPDATA_SIZE

  The buffer size available for user data in the uip_buf buffer.

Typedefs

- typedef uip_ip4addr_t uip_ip4addr_t

  Representation of an IP address.

Functions

- 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_init (void)

  uIP initialization function.

- uip_udp_conn ∗ uip_udp_new (const 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

- CCIF void * uip_appdata

  Pointer to the application data in the packet buffer.

- CCIF struct uip_conn * uip_conn

  Pointer to the current TCP connection.

- uip_udp_conn * uip_udp_conn

  The current UDP connection.

- uip_stats uip_stat

  The uIP TCP/IP statistics.

- 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_udp_conn * uip_udp_conn

  The current UDP connection.

- u8_t uip_acc32 [4]

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

6.7.12 Define Documentation

6.7.12.1 #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 1552 of file uip.h.

6.7.13 Function Documentation

6.7.13.1 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 1866 of file uip.c.

References HTONS.

Referenced by uip_chksum(), uip_ipchksum(), and uip_udp_new().

6.7.13.2 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.

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 296 of file uip.c.

References htons().

6.7.13.3 void uip_init (void)

uIP initialization function.

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

Definition at line 364 of file uip.c.

References uip_udp_conn::lport, uip_conn::tcpstateflags, UIP_CONNS, UIP_LISTENPORTS, and UIP_-UDP_CONNS.

6.7.13.4 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 303 of file uip.c.

References htons(), uip_buf, and UIP_LLH_LEN.

Referenced by uip_split_output().

6.7.13.5 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 514 of file uip.c.

References UIP_LISTENPORTS.

Referenced by tcp_listen().

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

Note:

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

Parameters:

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

len The maximum amount of data bytes to be sent.

Definition at line 1878 of file uip.c.

6.7.13.7 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 166 of file uip.c.

6.7.13.8 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 349 of file uip.c.

Referenced by uip_split_output().

6.7.13.9 struct uip_udp_conn* uip_udp_new (const 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 458 of file uip.c.

References HTONS, htons(), uip_udp_conn::lport, uip_udp_conn::ripaddr, uip_udp_conn::rport, uip_udp_conn::ttl, uip_ipaddr_copy, UIP_TTL, uip_udp_conn, and UIP_UDP_CONNS.

Referenced by udp_new().

6.7.13.10 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.

6.7.13.11 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 503 of file uip.c.

References UIP_LISTENPORTS.

Referenced by tcp_unlisten().

6.7.14 Variable Documentation

6.7.14.1 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 128 of file uip.c.

Referenced by uip_arp_out(), uip_fw_forward(), and uip_split_output().

6.7.14.2 CCIF 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 128 of file uip.c.

Referenced by uip_arp_out(), uip_fw_forward(), and uip_split_output().

6.7.14.3 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 124 of file uip.c.

Referenced by uip_arp_out(), uip_fw_forward(), and uip_ipchksum().

6.7.14.4 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 148 of file uip.c.

6.7.14.5 CCIF 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 148 of file uip.c.

6.7.14.6 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 140 of file uip.c.

Referenced by tcpip_input(), uip_arp_arpin(), uip_arp_out(), uip_fw_forward(), uip_fw_output(), and uip_split_output().

6.7.14.7 struct uip_stats uip_stat

The uIP TCP/IP statistics.

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

6.8 The Rime communication stack

6.8.1 Detailed Description

The Rime communication stack provides a set of lightweight communication primitives ranging from best-effort anonymous local area broadcast to reliable network flooding.

Files

- file rime.h

  Header file for the Rime stack.

Modules

- Anonymous best-effort local area broadcast
- Callback timer
- Identified best-effort local area broadcast

- Mesh routing
- Best-effort multihop forwarding
- Rime neighbor management
- Best-effort network flooding
- Rime queue buffer management
- Rime addresses
- Rime buffer management
- Rime route discovery protocol
- Rime route table
- Stubborn anonymous best-effort local area broadcast
- Stubborn identified broadcast
- Stubborn unicast
- Tree-based hop-by-hop reliable data collection
- Reliable single-source multi-hop flooding
- Unique anonymous best effort local area broadcast
- Single-hop unicast
- Unique identified best effort local area broadcast
- Single-hop reliable bulk data transfer
- Multi-hop reliable bulk data transfer

Functions

- void rime_init (const struct mac_driver ∗)

  Initialize Rime.

- void rime_input (void)

  Send an incoming packet to Rime.

- void rime_driver_send (void)

  Rime calls this function to send out a packet.

6.8.2 Function Documentation

6.8.2.1 void rime_driver_send (void)

Rime calls this function to send out a packet.

This function must be implemented by the driver running below Rime. It is called by abRime to send out a packet. The packet is consecutive in the rimebuf. A pointer to the first byte of the packet is obtained with the rimebuf_hdrptr() function. The length of the packet to send is obtained with the rimebuf_totlen() function.

The driver, which typically is a MAC protocol, may queue the packet by using the queuebuf functions.

6.8.2.2 void rime_init (const struct mac_driver ∗)

Initialize Rime.

This function should be called from the system boot up code to initialize Rime.

6.8.2.3 void rime_input (void)

Send an incoming packet to Rime.

This function should be called by the network driver to hand over a packet to Rime for furhter processing. The packet should be placed in the rimebuf (with rimebuf_copyfrom()) before calling this function.

6.9 The Contiki build system

The Contiki build system is designed to make it easy to compile Contiki applications for either to a hardware platform or into a simulation platform by simply supplying different parameters to the `make` command, without having to edit makefiles or modify the application code.

The file example project in examples/hello-world/ shows how the Contiki build system works. The `hello-world.c` application can be built into a complete Contiki system by running `make` in the examples/hello-world/ directory. Running `make` without parameters will build a Contiki system using the `native` target. The `native` target is a special Contiki platform that builds an entire Contiki system as a program that runs on the development system. After compiling the application for the `native` target it is possible to run the Contiki system with the application by running the file `hello-world.native`. To compile the application and a Contiki system for the ESB platform" the command `make TARGET=esb` is used. This produces a hello-world.esb file that can be loaded into an ESB board.

To compile the hello-world application into a stand-alone executable that can be loaded into a running Contiki system, the command `make hello-world.ce` is used. To build an executable file for the ESB platform, `make TARGET=esb hello-world.ce` is run.

To avoid having to type `TARGET=` every time `make` is run, it is possible to run `make TARGET=esb savetarget` to save the selected target as the default target platform for subsequent invocations of `make`. A file called `Makefile.target` containing the currently saved target is saved in the project's directory.

6.9.1 Makefiles used in the Contiki build system

The Contiki build system is composed of a number of Makefiles. These are:

- `Makefile`: the project's makefile, located in the project directory.
- `Makefile.include`: the system-wide Contiki makefile, located in the root of the Contiki source tree.
- `Makefile.$(TARGET)` (where $(TARGET) is the name of the platform that is currently being built): rules for the specific platform, located in the platform's subdirectory in the platform/ directory.
- `Makefile.$(CPU)` (where $(CPU) is the name of the CPU or microcontroller architecture used on the platform for which Contiki is built): rules for the CPU architecture, located in the CPU architecture's subdirectory in the cpu/ directory.
- `Makefile.$(APP)` (where $(APP) is the name of an application in the apps/ directory): rules for applications in the apps/ directories. Each application has its own makefile.

The Makefile in the project's directory is intentionally simple. It specifies where the Contiki source code resides in the system and includes the system-wide Makefile, `Makefile.include`. The project's makefile can also define in the `APPS` variable a list of applications from the apps/ directory that should be included in the Contiki system. The Makefile used in the hello-world example project looks like this:

```
CONTIKI = ../..
all: hello-world
include $(CONTIKI)/Makefile.include
```

First, the location of the Contiki source code tree is given by defining the `CONTIKI` variable. Next, the name of the application is defined. Finally, the system-wide `Makefile.include` is included.

The `Makefile.include` contains definitions of the C files of the core Contiki system. `Makefile.include` always reside in the root of the Contiki source tree. When `make` is run, `Makefile.include` includes the `Makefile.$(TARGET)` as well as all makefiles for the applications in the `APPS` list (which is specified by the project's `Makefile`).

`Makefile.$(TARGET)`, which is located in the platform/$(TARGET)/ directory, contains the list of C files that the platform adds to the Contiki system. This list is defined by the `CONTIKI_TARGET_-SOURCEFILES` variable. The `Makefile.$(TARGET)` also includes the `Makefile.$(CPU)` from the cpu/$(CPU)/ directory.

The `Makefile.$(CPU)` typically contains definitions for the C compiler used for the particular CPU. If multiple C compilers are used, the `Makefile.$(CPU)` can either contain a conditional expression that allows different C compilers to be defined, or it can be completely overridden by the platform specific makefile `Makefile.$(TARGET)`.

6.10 Contiki processes

6.10.1 Detailed Description

A process in Contiki consists of a single protothread.

Files

- file process.c

    Implementation of the Contiki process kernel.

- file process.h

    Header file for the Contiki process interface.

Return values

- #define PROCESS_ERR_OK 0

    Return value indicating that an operation was successful.

- #define PROCESS_ERR_FULL 1

    Return value indicating that the event queue was full.

Process protothread functions

- #define PROCESS_BEGIN()

    Define the beginning of a process.

- #define PROCESS_END()

    Define the end of a process.

- #define PROCESS_WAIT_EVENT()

Wait for an event to be posted to the process.

- #define PROCESS_WAIT_EVENT_UNTIL(c)

  Wait for an event to be posted to the process, with an extra condition.

- #define PROCESS_YIELD()

  Yield the currently running process.

- #define PROCESS_YIELD_UNTIL(c)

  Yield the currently running process until a condition occurs.

- #define PROCESS_WAIT_UNTIL(c)

  Wait for a condition to occur.

- #define PROCESS_EXIT()

  Exit the currently running process.

- #define PROCESS_PT_SPAWN(pt, thread)

  Spawn a protothread from the process.

- #define PROCESS_PAUSE()

  Yield the process for a short while.

Poll and exit handlers

- #define PROCESS_POLLHANDLER(handler)

  Specify an action when a process is polled.

- #define PROCESS_EXITHANDLER(handler)

  Specify an action when a process exits.

Process declaration and definion

- #define PROCESS_THREAD(name, ev, data)

  Define the body of a process.

- #define PROCESS_NAME(name)

  Declare the name of a process.

- #define PROCESS(name, strname)

  Declare a process.

Functions called from application programs

- #define PROCESS_CURRENT()

  Get a pointer to the currently running process.
- #define PROCESS_CONTEXT_BEGIN(p)

  Switch context to another process.
- #define PROCESS_CONTEXT_END(p) process_current = tmp_current; }

  End a context switch.
- process_event_t process_alloc_event (void)

  Allocate a global event number.
- void process_start (struct process ∗p, char ∗arg)

  Start a process.
- void process_exit (struct process ∗p)

  Cause a process to exit.
- int process_post (struct process ∗p, process_event_t ev, process_data_t data)

  Post an asynchronous event.
- void process_post_synch (struct process ∗p, process_event_t ev, process_data_t data)

  Post a synchronous event to a process.

Functions called by the system and boot-up code

- void process_init (void)

  Initialize the process module.
- int process_run (void)

  Run the system once - call poll handlers and process one event.
- int process_nevents (void)

  Number of events waiting to be processed.

Functions called from device drivers

- void process_poll (struct process ∗p)

  Request a process to be polled.

6.10.2 Define Documentation

6.10.2.1 #define PROCESS(name, strname)

Declare a process.

This macro declares a process. The process has two names: the variable of the process structure, which is used by the C program, and a human readable string name, which is used when debugging.

Parameters:

name The variable name of the process structure.

strname The string repressentation of the process' name.

Examples:

example-packet-drv.c, example-pollhandler.c, example-program.c, example-psock-server.c, test-abc.c, test-meshroute.c, test-rudolph0.c, test-rudolph1.c, test-treeroute.c, and test-trickle.c.

Definition at line 312 of file process.h.

6.10.2.2 #define PROCESS_BEGIN()

Define the beginning of a process.

This macro defines the beginning of a process, and must always appear in a PROCESS_THREAD() definition. The PROCESS_END() macro must come at the end of the process.

Examples:

example-packet-drv.c, example-pollhandler.c, example-program.c, example-psock-server.c, and test-treeroute.c.

Definition at line 121 of file process.h.

6.10.2.3 #define PROCESS_CONTEXT_BEGIN(p)

Value:

```
{\
struct process *tmp_current = PROCESS_CURRENT();\
process_current = p
```

Switch context to another process.

This function switch context to the specified process and executes the code as if run by that process. Typical use of this function is to switch context in services, called by other processes. Each PROCESS_CONTEXT_BEGIN() must be followed by the PROCESS_CONTEXT_END() macro to end the context switch.

Example:

```
 PROCESS_CONTEXT_BEGIN(&test_process);
 etimer_set(&timer, CLOCK_SECOND);
 PROCESS_CONTEXT_END(&test_process);
```

Parameters:

p The process to use as context

See also:

PROCESS_CONTEXT_END()
PROCESS_CURRENT()

Definition at line 424 of file process.h.

6.10.2.4 #define PROCESS_CONTEXT_END(p) process_current = tmp_current; }

End a context switch.

This function ends a context switch and changes back to the previous process.

Parameters:

p The process used in the context switch

See also:

PROCESS_CONTEXT_START()

Definition at line 438 of file process.h.

6.10.2.5 #define PROCESS_CURRENT()

Get a pointer to the currently running process.

This macro get a pointer to the currently running process. Typically, this macro is used to post an event to the current process with process_post().

Definition at line 400 of file process.h.

Referenced by ctk_desktop_redraw(), process_exit(), tcp_attach(), tcp_connect(), tcp_listen(), tcp_unlisten(), udp_attach(), and udp_new().

6.10.2.6 #define PROCESS_END()

Define the end of a process.

This macro defines the end of a process. It must appear in a PROCESS_THREAD() definition and must always be included. The process exits when the PROCESS_END() macro is reached.

Examples:

example-packet-drv.c, example-pollhandler.c, example-program.c, example-psock-server.c, test-abc.c, test-meshroute.c, test-rudolph0.c, test-rudolph1.c, test-treeroute.c, and test-trickle.c.

Definition at line 132 of file process.h.

6.10.2.7 #define PROCESS_ERR_FULL 1

Return value indicating that the event queue was full.

This value is returned from process_post() to indicate that the event queue was full and that an event could not be posted.

Definition at line 83 of file process.h.

Referenced by process_post().

6.10.2.8 #define PROCESS_ERR_OK 0

Return value indicating that an operation was successful.

This value is returned to indicate that an operation was successful.

Definition at line 75 of file process.h.

Referenced by process_post().

6.10.2.9 #define PROCESS_EXITHANDLER(handler)

Specify an action when a process exits.

Note:
This declaration must come immediately before the PROCESS_BEGIN() macro.

Parameters:
handler The action to be performed.

Examples:
example-packet-drv.c, example-pollhandler.c, test-abc.c, test-meshroute.c, test-rudolph0.c, test-rudolph1.c, and test-trickle.c.

Definition at line 255 of file process.h.

6.10.2.10 #define PROCESS_NAME(name)

Declare the name of a process.

This macro is typically used in header files to declare the name of a process that is implemented in the C file.

Definition at line 294 of file process.h.

6.10.2.11 #define PROCESS_PAUSE()

Yield the process for a short while.

This macro yields the currently running process for a short while, thus letting other processes run before the process continues.

Definition at line 222 of file process.h.

6.10.2.12 #define PROCESS_POLLHANDLER(handler)

Specify an action when a process is polled.

Note:
This declaration must come immediately before the PROCESS_BEGIN() macro.

Parameters:
handler The action to be performed.

Examples:
example-packet-drv.c, and example-pollhandler.c.

Definition at line 243 of file process.h.

6.10.2.13 #define PROCESS_PT_SPAWN(pt, thread)

Spawn a protothread from the process.

Parameters:

pt The protothread state (struct pt) for the new protothread

thread The call to the protothread function.

See also:

PT_SPAWN()

Definition at line 212 of file process.h.

6.10.2.14 #define PROCESS_THREAD(name, ev, data)

Define the body of a process.

This macro is used to define the body (protothread) of a process. The process is called whenever an event occurs in the system, A process always start with the PROCESS_BEGIN() macro and end with the PROCESS_END() macro.

Examples:

example-packet-drv.c, example-pollhandler.c, example-program.c, example-psock-server.c, test-abc.c, test-meshroute.c, test-rudolph0.c, test-rudolph1.c, test-treeroute.c, and test-trickle.c.

Definition at line 274 of file process.h.

6.10.2.15 #define PROCESS_WAIT_EVENT()

Wait for an event to be posted to the process.

This macro blocks the currently running process until the process receives an event.

Examples:

example-pollhandler.c, and test-treeroute.c.

Definition at line 142 of file process.h.

6.10.2.16 #define PROCESS_WAIT_EVENT_UNTIL(c)

Wait for an event to be posted to the process, with an extra condition.

This macro is similar to PROCESS_WAIT_EVENT() in that it blocks the currently running process until the process receives an event. But PROCESS_WAIT_EVENT_UNTIL() takes an extra condition which must be true for the process to continue.

Parameters:

c The condition that must be true for the process to continue.

See also:

PT_WAIT_UNTIL()

Examples:

example-packet-drv.c, example-program.c, example-psock-server.c, test-abc.c, test-meshroute.c, test-rudolph0.c, test-rudolph1.c, and test-trickle.c.

Definition at line 158 of file process.h.

6.10.2.17 #define PROCESS_WAIT_UNTIL(c)

Wait for a condition to occur.

This macro does not guarantee that the process yields, and should therefore be used with care. In most cases, PROCESS_WAIT_EVENT(), PROCESS_WAIT_EVENT_UNTIL(), PROCESS_YIELD() or PROCESS_YIELD_UNTIL() should be used instead.

Parameters:

c The condition to wait for.

Examples:

test-treeroute.c.

Definition at line 193 of file process.h.

6.10.2.18 #define PROCESS_YIELD_UNTIL(c)

Yield the currently running process until a condition occurs.

This macro is different from PROCESS_WAIT_UNTIL() in that PROCESS_YIELD_UNTIL() is guaranteed to always yield at least once. This ensures that the process does not end up in an infinite loop and monopolizing the CPU.

Parameters:

c The condition to wait for.

Definition at line 179 of file process.h.

6.10.3 Function Documentation

6.10.3.1 process_event_t process_alloc_event (void)

Allocate a global event number.

Returns:

The allocated event number

In Contiki, event numbers above 128 are global and may be posted from one process to another. This function allocates one such event number.

Note:

There currently is no way to deallocate an allocated event number.

Definition at line 96 of file process.c.

6.10.3.2 CCIF void process_exit (struct process * *p*)

Cause a process to exit.

Parameters:

p The process that is to be exited

This function causes a process to exit. The process can either be the currently executing process, or another process that is currently running.

See also:
PROCESS_CURRENT()

Definition at line 209 of file process.c.

References PROCESS_CURRENT.

6.10.3.3 void process_init (void)

Initialize the process module.

This function initializes the process module and should be called by the system boot-up code.

Definition at line 215 of file process.c.

6.10.3.4 int process_nevents (void)

Number of events waiting to be processed.

Returns:
The number of events that are currently waiting to be processed.

Definition at line 362 of file process.c.

6.10.3.5 CCIF void process_poll (struct process ∗ *p*)

Request a process to be polled.

This function typically is called from an interrupt handler to cause a process to be polled.

Parameters:
p A pointer to the process' process structure.

Examples:
example-packet-drv.c.

Definition at line 406 of file process.c.

Referenced by etimer_request_poll().

6.10.3.6 CCIF int process_post (struct process ∗ *p*, process_event_t *ev*, process_data_t *data*)

Post an asynchronous event.

This function posts an asynchronous event to one or more processes. The handing of the event is deferred until the target process is scheduled by the kernel. An event can be broadcast to all processes, in which case all processes in the system will be scheduled to handle the event.

Parameters:
ev The event to be posted.
data The auxillary data to be sent with the event

p The process to which the event should be posted, or PROCESS_BROADCAST if the event should be posted to all processes.

Return values:

PROCESS_ERR_OK The event could be posted.

PROCESS_ERR_FULL The event queue was full and the event could not be posted.

Definition at line 372 of file process.c.

References PROCESS_ERR_FULL, and PROCESS_ERR_OK.

Referenced by process_start(), program_handler_load(), resolv_conf(), tcpip_poll_tcp(), and tcpip_poll_-udp().

6.10.3.7 void process_post_synch (struct process ∗ *p*, process_event_t *ev*, process_data_t *data*)

Post a synchronous event to a process.

Parameters:

p A pointer to the process' process structure.

ev The event to be posted.

data A pointer to additional data that is posted together with the event.

Definition at line 397 of file process.c.

Referenced by tcpip_input().

6.10.3.8 int process_run (void)

Run the system once - call poll handlers and process one event.

This function should be called repeatedly from the main() program to actually run the Contiki system. It calls the necessary poll handlers, and processes one event. The function returns the number of events that are waiting in the event queue so that the caller may choose to put the CPU to sleep when there are no pending events.

Returns:

The number of events that are currently waiting in the event queue.

Definition at line 348 of file process.c.

6.10.3.9 void process_start (struct process ∗ *p*, char ∗ *arg*)

Start a process.

Parameters:

p A pointer to a process structure.

arg An argument pointer that can be passed to the new process

Definition at line 102 of file process.c.

References process_post(), and PT_INIT.

6.11 Event timers

6.11.1 Detailed Description

Event timers provides a way to generate timed events.

An event timer will post an event to the process that set the timer when the event timer expires.

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

See also:

Simple timer library
Clock library (used by the timer library)

Files

- file etimer.c

  Event timer library implementation.

- file etimer.h

  Event timer header file.

Data Structures

- struct etimer

  A timer.

Functions called from timer interrupts, by the system

- void etimer_request_poll (void)

  Make the event timer aware that the clock has changed.

- int etimer_pending (void)

  Check if there are any non-expired event timers.

- clock_time_t etimer_next_expiration_time (void)

  Get next event timer expiration time.

Functions called from application programs

- void etimer_set (struct etimer ∗et, clock_time_t interval)

  Set an event timer.

- void etimer_reset (struct etimer ∗et)

  Reset an event timer with the same interval as was previously set.

- void etimer_restart (struct etimer ∗et)

Restart an event timer from the current point in time.

- void etimer_adjust (struct etimer ∗et, int timediff)

  Adjust the expiration time for an event timer.

- int etimer_expired (struct etimer ∗et)

  Check if an event timer has expired.

- clock_time_t etimer_expiration_time (struct etimer ∗et)

  Get the expiration time for the event timer.

- clock_time_t etimer_start_time (struct etimer ∗et)

  Get the start time for the event timer.

- void etimer_stop (struct etimer ∗et)

  Stop a pending event timer.

6.11.2 Function Documentation

6.11.2.1 void etimer_adjust (struct etimer ∗ *et*, int *td*)

Adjust the expiration time for an event timer.

Parameters:

et A pointer to the event timer.

td The time difference to adjust the expiration time with.

This function is used to adjust the time the event timer will expire. It can be used to synchronize periodic timers without the need to restart the timer or change the timer interval.

Note:

This function should only be used for small adjustments. For large adjustments use etimer_set() instead.
A periodic timer will drift unless the etimer_reset() function is used.

See also:

etimer_set()
etimer_reset()

Definition at line 199 of file etimer.c.

References timer::start, and timer.

6.11.2.2 clock_time_t etimer_expiration_time (struct etimer ∗ *et*)

Get the expiration time for the event timer.

Parameters:

et A pointer to the event timer

Returns:

The expiration time for the event timer.

This function returns the expiration time for an event timer.

Definition at line 212 of file etimer.c.

References timer::interval, and timer::start.

6.11.2.3 CCIF int etimer_expired (struct etimer * *et*)

Check if an event timer has expired.

Parameters:

et A pointer to the event timer

Returns:

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

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

Examples:

example-program.c, test-abc.c, test-meshroute.c, and test-treeroute.c.

Definition at line 206 of file etimer.c.

6.11.2.4 clock_time_t etimer_next_expiration_time (void)

Get next event timer expiration time.

Returns:

Next expiration time of all pending event timers. If there are no pending event timers this function returns 0.

This functions returns next expiration time of all pending event timers.

Definition at line 230 of file etimer.c.

References etimer_pending().

6.11.2.5 int etimer_pending (void)

Check if there are any non-expired event timers.

Returns:

True if there are active event timers, false if there are no active timers.

This function checks if there are any active event timers that have not expired.

Definition at line 224 of file etimer.c.

Referenced by etimer_next_expiration_time().

6.11.2.6 void etimer_request_poll (void)

Make the event timer aware that the clock has changed.

This function is used to inform the event timer module that the system clock has been updated. Typically, this function would be called from the timer interrupt handler when the clock has ticked.

Definition at line 146 of file etimer.c.

References process_poll().

6.11.2.7 void etimer_reset (struct etimer * *et*)

Reset an event timer with the same interval as was previously set.

Parameters:

et A pointer to the event timer.

This function resets the event timer with the same interval that was given to the event timer with the etimer_set() function. The start point of the interval is the exact time that the event timer last expired. Therefore, this function will cause the timer to be stable over time, unlike the etimer_restart() function.

See also:

etimer_restart()

Definition at line 185 of file etimer.c.

References timer_reset().

6.11.2.8 void etimer_restart (struct etimer * *et*)

Restart an event timer from the current point in time.

Parameters:

et A pointer to the event timer.

This function restarts the event timer with the same interval that was given to the etimer_set() function. The event timer will start at the current time.

Note:

A periodic timer will drift if this function is used to reset it. For periodic timers, use the etimer_reset() function instead.

See also:

etimer_reset()

Definition at line 192 of file etimer.c.

References timer_restart().

6.11.2.9 CCIF void etimer_set (struct etimer * *et*, clock_time_t *interval*)

Set an event timer.

Parameters:

et A pointer to the event timer

interval The interval before the timer expires.

This function is used to set an event timer for a time sometime in the future. When the event timer expires, the event PROCESS_EVENT_TIMER will be posted to the process that called the etimer_set() function.

Examples:
example-program.c, test-abc.c, and test-treeroute.c.

Definition at line 178 of file etimer.c.

References timer_set().

6.11.2.10 clock_time_t etimer_start_time (struct etimer ∗ *et*)

Get the start time for the event timer.

Parameters:
et A pointer to the event timer

Returns:
The start time for the event timer.

This function returns the start time (when the timer was last set) for an event timer.

Definition at line 218 of file etimer.c.

References timer::start.

6.11.2.11 void etimer_stop (struct etimer ∗ *et*)

Stop a pending event timer.

Parameters:
et A pointer to the pending event timer.

This function stops an event timer that has previously been set with etimer_set() or etimer_reset(). After this function has been called, the event timer will not emit any event when it expires.

Definition at line 236 of file etimer.c.

References next, and p.

6.12 Argument buffer

6.12.1 Detailed Description

The argument buffer can be used when passing an argument from an exiting process to a process that has not been created yet. Since the exiting process will have exited when the new process is started, the argument cannot be passed in any of the processes' addres spaces. In such situations, the argument buffer can be used.

The argument buffer is statically allocated in memory and is globally accessible to all processes.

An argument buffer is allocated with the arg_alloc() function and deallocated with the arg_free() function. The arg_free() function is designed so that it can take any pointer, not just an argument buffer pointer. If the pointer to arg_free() is not an argument buffer, the function does nothing.

Functions

- char * arg_alloc (char size)
  Allocates an argument buffer.
- void arg_free (char *arg)
  Deallocates an argument buffer.

6.12.2 Function Documentation

6.12.2.1 char* arg_alloc (char *size*)

Allocates an argument buffer.

Parameters:
size The requested size of the buffer, in bytes.

Returns:
Pointer to allocated buffer, or NULL if no buffer could be allocated.

Note:
It currently is not possible to allocate argument buffers of any other size than 128 bytes.

Definition at line 105 of file arg.c.

6.12.2.2 void arg_free (char * *arg*)

Deallocates an argument buffer.

This function deallocates the argument buffer pointed to by the parameter, but only if the buffer actually is an argument buffer and is allocated. It is perfectly safe to call this function with any pointer.

Parameters:
arg A pointer.

Definition at line 126 of file arg.c.

6.13 The Contiki program loader

6.13.1 Detailed Description

The Contiki program loader is an abstract interface for loading and starting programs.

Files

- file loader.h
  Default definitions and error values for the Contiki program loader.

Modules

- The Contiki ELF loader

  The Contiki ELF loader links, relocates, and loads ELF (Executable Linkable Format) object files into a running Contiki system.

Data Structures

- struct dsc

  The DSC program description structure.

Defines

- #define DSC(dscname, description, prgname, process, icon) CLIF const struct dsc dscname = {description, prgname, icon}

  Intantiating macro for the DSC structure.

- #define LOADER_OK 0

  No error.

- #define LOADER_ERR_READ 1

  Read error.

- #define LOADER_ERR_HDR 2

  Header error.

- #define LOADER_ERR_OS 3

  Wrong OS.

- #define LOADER_ERR_FMT 4

  Data format error.

- #define LOADER_ERR_MEM 5

  Not enough memory.

- #define LOADER_ERR_OPEN 6

  Could not open file.

- #define LOADER_ERR_ARCH 7

  Wrong architecture.

- #define LOADER_ERR_VERSION 8

  Wrong OS version.

- #define LOADER_ERR_NOLOADER 9

  Program loading not supported.

- #define LOADER_LOAD(name, arg) LOADER_ERR_NOLOADER

Load and execute a program.

- #define LOADER_UNLOAD()

  Unload a program from memory.

- #define LOADER_LOAD_DSC(name) NULL

  Load a DSC (program description).

- #define LOADER_UNLOAD_DSC(dsc)

  Unload a DSC (program description).

6.13.1.1 The program description structure

The Contiki DSC structure is used for describing programs. It includes a string describing the program, the name of the program file on disk (or a pointer to the programs initialization function for systems without disk support), a bitmap icon and a text version of the same icon.

The DSC is saved into a file which can be loaded by programs such as the "Directory" application which reads all DSC files on disk and presents the icons and descriptions in a window.

6.13.2 Define Documentation

6.13.2.1 #define DSC(dscname, description, prgname, process, icon) CLIF const struct dsc dscname = {description, prgname, icon}

Intantiating macro for the DSC structure.

Parameters:

dscname The name of the C variable which is to contain the DSC.

description A one-line text describing the program.

prgname The name of the program on disk.

initfunc A pointer to the initialization function of the program.

icon A pointer to the CTK icon.

Definition at line 112 of file dsc.h.

6.13.2.2 #define LOADER_LOAD(name, arg) LOADER_ERR_NOLOADER

Load and execute a program.

This macro is used for loading and executing a program, and requires support from the architecture dependant code. The actual program loading is made by architecture specific functions.

Note:

A program loaded with LOADER_LOAD() must call the LOADER_UNLOAD() function to unload itself.

Parameters:

name The name of the program to be loaded.

arg A pointer argument that is passed to the program.

Returns:

A loader error, or LOADER_OK if loading was successful.

Definition at line 92 of file loader.h.

6.13.2.3 #define LOADER_LOAD_DSC(name) NULL

Load a DSC (program description).

Loads a DSC (program description) into memory and returns a pointer to the dsc.

Returns:

A pointer to the DSC or NULL if it could not be loaded.

Definition at line 116 of file loader.h.

6.13.2.4 #define LOADER_UNLOAD()

Unload a program from memory.

This macro is used for unloading a program and deallocating any memory that was allocated during the loading of the program. This function must be called by the program itself.

Definition at line 104 of file loader.h.

6.13.2.5 #define LOADER_UNLOAD_DSC(dsc)

Unload a DSC (program description).

Unload a DSC from memory and deallocate any memory that was allocated when it was loaded.

Definition at line 126 of file loader.h.

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_INIT(lc)
  Initialize a local continuation.
- #define LC_SET(lc)
  Set a local continuation.
- #define LC_RESUME(lc)
  Resume a local continuation.
- #define LC_END(lc)
  Mark the end of local continuation usage.

Typedefs

- typedef unsigned short lc_t
  The local continuation type.

6.14.2 Define Documentation

6.14.2.1 #define LC_END(lc)

Mark the end of local continuation usage.

The end operation signifies that local continuations should not be used any more in the function. This operation is not needed for most implementations of local continuation, but is required by a few implementations.

Definition at line 108 of file lc.h.

6.14.2.2 #define LC_INIT(lc)

Initialize a local continuation.

This operation initializes the local continuation, thereby unsetting any previously set continuation state.

Definition at line 71 of file lc.h.

6.14.2.3 #define LC_RESUME(lc)

Resume a local continuation.

The resume operation resumes a previously set local continuation, thus restoring the state in which the function was when the local continuation was set. If the local continuation has not been previously set, the resume operation does nothing.

Definition at line 96 of file lc.h.

6.14.2.4 #define LC_SET(lc)

Set a local continuation.

The set operation saves the state of the function at the point where the operation is executed. As far as the set operation is concerned, the state of the function does **not** include the call-stack or local (automatic) variables, but only the program counter and such CPU registers that needs to be saved.

Definition at line 84 of file lc.h.

6.15 Protothread semaphores

6.15.1 Detailed Description

This module implements counting semaphores on top of protothreads. Semaphores are a synchronization primitive that provide two operations: "wait" and "signal". The "wait" operation checks the semaphore counter and blocks the thread if the counter is zero. The "signal" operation increases the semaphore counter but does not block. If another thread has blocked waiting for the semaphore that is signalled, the blocked thread will become runnable again.

Semaphores can be used to implement other, more structured, synchronization primitives such as monitors and message queues/bounded buffers (see below).

The following example shows how the producer-consumer problem, also known as the bounded buffer problem, can be solved using protothreads and semaphores. Notes on the program follow after the example.

```
#include "pt-sem.h"

#define NUM_ITEMS 32
#define BUFSIZE 8

static struct pt_sem mutex, full, empty;

PT_THREAD(producer(struct pt *pt))
{
  static int produced;

  PT_BEGIN(pt);

  for(produced = 0; produced < NUM_ITEMS; ++produced) {

    PT_SEM_WAIT(pt, &full);

    PT_SEM_WAIT(pt, &mutex);
    add_to_buffer(produce_item());
    PT_SEM_SIGNAL(pt, &mutex);

    PT_SEM_SIGNAL(pt, &empty);
  }

  PT_END(pt);
}

PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;

  PT_BEGIN(pt);

  for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {

    PT_SEM_WAIT(pt, &empty);

    PT_SEM_WAIT(pt, &mutex);
    consume_item(get_from_buffer());
    PT_SEM_SIGNAL(pt, &mutex);
```

```
    PT_SEM_SIGNAL(pt, &full);
  }

  PT_END(pt);
}

PT_THREAD(driver_thread(struct pt *pt))
{
  static struct pt pt_producer, pt_consumer;

  PT_BEGIN(pt);

  PT_SEM_INIT(&empty, 0);
  PT_SEM_INIT(&full, BUFSIZE);
  PT_SEM_INIT(&mutex, 1);

  PT_INIT(&pt_producer);
  PT_INIT(&pt_consumer);

  PT_WAIT_THREAD(pt, producer(&pt_producer) &
                     consumer(&pt_consumer));

  PT_END(pt);
}
```

The program uses three protothreads: one protothread that implements the consumer, one thread that implements the producer, and one protothread that drives the two other protothreads. The program uses three semaphores: "full", "empty" and "mutex". The "mutex" semaphore is used to provide mutual exclusion for the buffer, the "empty" semaphore is used to block the consumer is the buffer is empty, and the "full" semaphore is used to block the producer is the buffer is full.

The "driver_thread" holds two protothread state variables, "pt_producer" and "pt_consumer". It is important to note that both these variables are declared as *static*. If the static keyword is not used, both variables are stored on the stack. Since protothreads do not store the stack, these variables may be overwritten during a protothread wait operation. Similarly, both the "consumer" and "producer" protothreads declare their local variables as static, to avoid them being stored on the stack.

Files

- file pt-sem.h

  Counting semaphores implemented on protothreads.

Defines

- #define PT_SEM_INIT(s, c)

  Initialize a semaphore.

- #define PT_SEM_WAIT(pt, s)

  Wait for a semaphore.

- #define PT_SEM_SIGNAL(pt, s)

  Signal a semaphore.

6.15.2 Define Documentation

6.15.2.1 #define PT_SEM_INIT(s, c)

Initialize a semaphore.

This macro initializes a semaphore with a value for the counter. Internally, the semaphores use an "unsigned int" to represent the counter, and therefore the "count" argument should be within range of an unsigned int.

Parameters:

s (struct pt_sem ∗) A pointer to the pt_sem struct representing the semaphore

c (unsigned int) The initial count of the semaphore.

Definition at line 183 of file pt-sem.h.

6.15.2.2 #define PT_SEM_SIGNAL(pt, s)

Signal a semaphore.

This macro carries out the "signal" operation on the semaphore. The signal operation increments the counter inside the semaphore, which eventually will cause waiting protothreads to continue executing.

Parameters:

pt (struct pt ∗) A pointer to the protothread (struct pt) in which the operation is executed.

s (struct pt_sem ∗) A pointer to the pt_sem struct representing the semaphore

Definition at line 222 of file pt-sem.h.

6.15.2.3 #define PT_SEM_WAIT(pt, s)

Wait for a semaphore.

This macro carries out the "wait" operation on the semaphore. The wait operation causes the protothread to block while the counter is zero. When the counter reaches a value larger than zero, the protothread will continue.

Parameters:

pt (struct pt ∗) A pointer to the protothread (struct pt) in which the operation is executed.

s (struct pt_sem ∗) A pointer to the pt_sem struct representing the semaphore

Definition at line 201 of file pt-sem.h.

6.16 Clock library

6.16.1 Detailed Description

The clock library is the interface between Contiki and the platform specific clock functionality.

The clock library performs a single function: measuring time. Additionally, the clock library provides a macro, CLOCK_SECOND, which corresponds to one second of system time.

Note:

The clock library need in many cases not be used directly. Rather, the timer library or the event timers should be used.

See also:
Timer library
Event timers

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 Multi-threading library

6.17.1 Detailed Description

The event driven Contiki kernel does not provide multi-threading by itself - instead, preemptive multi-threading is implemented as a library that optionally can be linked with applications.

This library constists of two parts: a platform independent part, which is the same for all platforms on which Contiki runs, and a platform specific part, which must be implemented specifically for the platform that the multi-threading library should run.

Modules

- Architecture support for multi-threading

Defines

- #define MT_OK
  No error.

Functions

- void mt_init (void)
  Initializes the multithreading library.
- void mt_remove (void)
  Uninstalls library and cleans up.
- void mt_start (struct mt_thread *thread, void(*function)(void *), void *data)
  Starts a multithreading thread.
- void mt_exec (struct mt_thread *thread)
  Execute parts of a thread.
- void mt_yield (void)
  Voluntarily give up the processor.
- void mt_exit (void)
  Exit a thread.
- void mt_stop (struct mt_thread *thread)
  Stop a thread.

6.17.2 Function Documentation

6.17.2.1 void mt_exec (struct mt_thread * *thread*)

Execute parts of a thread.

This function is called by a Contiki process and runs a thread. The function does not return until the thread has yielded, or is preempted.

Note:
The thread library must first be initialized with the mt_init() function.

Parameters:
thread A pointer to a struct mt_thread block that must be allocated by the caller.

Definition at line 82 of file mt.c.

References mtarch_exec().

6.17.2.2 void mt_exit (void)

Exit a thread.

This function is called from within an executing thread in order to exit the thread. The function never returns.

Definition at line 110 of file mt.c.

References mtarch_yield().

6.17.2.3 void mt_start (struct mt_thread ∗ *thread*, void(∗)(void ∗) *function*, void ∗ *data*)

Starts a multithreading thread.

Parameters:

thread Pointer to an mt_thread struct that must have been previously allocated by the caller.

function A pointer to the entry function of the thread that is to be set up.

data A pointer that will be passed to the entry function.

Definition at line 72 of file mt.c.

References mtarch_start().

6.17.2.4 void mt_stop (struct mt_thread ∗ *thread*)

Stop a thread.

This function is called by a Contiki process in order to clean up a thread. The struct mt_thread block may then be discarded by the caller.

Parameters:

thread A pointer to a struct mt_thread block that must be allocated by the caller.

Definition at line 118 of file mt.c.

References mtarch_stop().

6.17.2.5 void mt_yield (void)

Voluntarily give up the processor.

This function is called by a running thread in order to give up control of the CPU.

Definition at line 96 of file mt.c.

References mtarch_yield().

6.18 Architecture support for multi-threading

6.18.1 Detailed Description

The Contiki multi-threading library requires some architecture specific support for seting up and switching stacks. This support requires four stack manipulation functions to be implemented: mtarch_start(), which sets up the stack frame for a new thread, mtarch_exec(), which switches in the stack of a thread, mtarch_-yield(), which restores the kernel stack from a thread's stack and mtarch_stop(), which cleans up the stack of a thread. Additionally, two functions for controlling the preemption (if any) must be implemented: mtarch_pstart() and mtarch_pstop(). If no preemption is used, these functions can be implemented as empty functions. Finally, the function mtarch_init() is called by mt_init(), and can be used for initalization of timer interrupts, or any other mechanisms required for correct operation of the architecture specific support funcions while mtarch_remove() is called by mt_remove() to clean up those resources.

Files

- file mt.h

  Header file for the preemptive multitasking library for Contiki.

Functions

- void mtarch_init (void)

  Initialize the architecture specific support functions for the multi-thread library.

- void mtarch_remove (void)

  Uninstall library and clean up.

- void mtarch_start (struct mtarch_thread ∗thread, void(∗function)(void ∗data), void ∗data)

  Setup the stack frame for a thread that is being started.

- void mtarch_exec (struct mtarch_thread ∗thread)

  Start executing a thread.

- void mtarch_yield (void)

  Yield the processor.

- void mtarch_stop (struct mtarch_thread ∗thread)

  Clean up the stack of a thread.

6.18.2 Function Documentation

6.18.2.1 void mtarch_exec (struct mtarch_thread ∗ *thread*)

Start executing a thread.

This function is called from mt_exec() and the purpose of the function is to start execution of the thread. The function should switch in the stack of the thread, and does not return until the thread has explicitly yielded (using mt_yield()) or until it is preempted.

Parameters:

thread A pointer to a struct mtarch_thread for the thread to be executed.

Referenced by mt_exec().

6.18.2.2 struct void mtarch_init (void)

Initialize the architecture specific support functions for the multi-thread library.

This function is implemented by the architecture specific functions for the multi-thread library and is called by the mt_init() function as part of the initialization of the library. The mtarch_init() function can be used for, e.g., starting preemtion timers or other architecture specific mechanisms required for the operation of the library.

Referenced by mt_init().

6.18.2.3 void mtarch_start (struct mtarch_thread * *thread*, void(*)(void *data) *function*, void * *data*)

Setup the stack frame for a thread that is being started.

This function is called by the mt_start() function in order to set up the architecture specific stack of the thread to be started.

Parameters:

thread A pointer to a struct mtarch_thread for the thread to be started.

function A pointer to the function that the thread will start executing the first time it is scheduled to run.

data A pointer to the argument that the function should be passed.

Referenced by mt_start().

6.18.2.4 void mtarch_stop (struct mtarch_thread * *thread*)

Clean up the stack of a thread.

This function is called by the mt_stop() function in order to clean up the architecture specific stack of the thread to be stopped.

Note:

If the stack is wholly contained in struct mtarch_thread this function may very well be empty.

Parameters:

thread A pointer to a struct mtarch_thread for the thread to be stopped.

Referenced by mt_stop().

6.18.2.5 void mtarch_yield (void)

Yield the processor.

This function is called by the mt_yield() function, which is called from the running thread in order to give up the processor.

Referenced by mt_exit(), and mt_yield().

6.19 EEPROM API

6.19.1 Detailed Description

The EEPROM API defines a common interface for EEPROM access on Contiki platforms.

A platform with EEPROM support must implement this API.

Files

- file eeprom.h

  EEPROM functions.

Functions

- void eeprom_write (eeprom_addr_t addr, unsigned char ∗buf, int size)

  Write a buffer into EEPROM.

- void eeprom_read (eeprom_addr_t addr, unsigned char ∗buf, int size)

  Read data from the EEPROM.

- void eeprom_init (void)

  Initialize the EEPROM module.

6.19.2 Function Documentation

6.19.2.1 void eeprom_init (void)

Initialize the EEPROM module.

This function initializes the EEPROM module and is called from the bootup code.

6.19.2.2 void eeprom_read (eeprom_addr_t *addr*, unsigned char ∗ *buf*, int *size*)

Read data from the EEPROM.

This function reads a number of bytes from the specified address in EEPROM and into a buffer in memory.

Parameters:

addr The address in EEPROM from which the data should be read.

buf A pointer to the buffer to which the data should be stored.

size The number of bytes to read.

6.19.2.3 void eeprom_write (eeprom_addr_t *addr*, unsigned char ∗ *buf*, int *size*)

Write a buffer into EEPROM.

This function writes a buffer of the specified size into EEPROM.

Parameters:

addr The address in EEPROM to which the buffer should be written.

buf A pointer to the buffer from which data is to be read.

size The number of bytes to write into EEPROM.

6.20 Radio API

6.20.1 Detailed Description

The radio API module defines a set of functions that a radio device driver must implement.

Files

- file radio.h

  Header file for the radio API.

Data Structures

- struct radio_driver

  The structure of a device driver for a radio in Contiki.

6.21 The Contiki ELF loader

6.21.1 Detailed Description

The Contiki ELF loader links, relocates, and loads ELF (Executable Linkable Format) object files into a running Contiki system.

ELF is a standard format for relocatable object code and executable files. ELF is the standard program format for Linux, Solaris, and other operating systems.

An ELF file contains either a standalone executable program or a program module. The file contains both the program code, the program data, as well as information about how to link, relocate, and load the program into a running system.

The ELF file is composed of a set of sections. The sections contain program code, data, or relocation information, but can also contain debugging information.

To link and relocate an ELF file, the Contiki ELF loader first parses the ELF file structure to find the appropriate ELF sections. It then allocates memory for the program code and data in ROM and RAM, respectively. After allocating memory, the Contiki ELF loader starts relocating the code found in the ELF file.

Files

- file elfloader.h

  Header file for the Contiki ELF loader.

Modules

- Architecture specific functionality for the ELF loader.

  The architecture specific functionality for the Contiki ELF loader has to be implemented for each processor type Contiki runs on.

Defines

- #define ELFLOADER_OK 0

    Return value from elfloader_load() indicating that loading worked.

- #define ELFLOADER_BAD_ELF_HEADER 1

    Return value from elfloader_load() indicating that the ELF file had a bad header.

- #define ELFLOADER_NO_SYMTAB 2

    Return value from elfloader_load() indicating that no symbol table could be find in the ELF file.

- #define ELFLOADER_NO_STRTAB 3

    Return value from elfloader_load() indicating that no string table could be find in the ELF file.

- #define ELFLOADER_NO_TEXT 4

    Return value from elfloader_load() indicating that the size of the .text segment was zero.

- #define ELFLOADER_SYMBOL_NOT_FOUND 5

    Return value from elfloader_load() indicating that a symbol specific symbol could not be found.

- #define ELFLOADER_SEGMENT_NOT_FOUND 6

    Return value from elfloader_load() indicating that one of the required segments (.data, .bss, or .text) could not be found.

- #define ELFLOADER_NO_STARTPOINT 7

    Return value from elfloader_load() indicating that no starting point could be found in the loaded module.

Functions

- void elfloader_init (void)

    elfloader initialization function.

- int elfloader_load (int fd)

    Load and relocate an ELF file.

Variables

- process ** elfloader_autostart_processes

    A pointer to the processes loaded with elfloader_load().

- char elfloader_unknown [30]

    If elfloader_load() could not find a specific symbol, it is copied into this array.

6.21.2 Define Documentation

6.21.2.1 #define ELFLOADER_SYMBOL_NOT_FOUND 5

Return value from elfloader_load() indicating that a symbol specific symbol could not be found.

If this value is returned from elfloader_load(), the symbol has been copied into the elfloader_unknown[] array.

Definition at line 111 of file elfloader.h.

6.21.3 Function Documentation

6.21.3.1 void elfloader_init (void)

elfloader initialization function.

This function should be called at boot up to initilize the elfloader.

6.21.3.2 int elfloader_load (int *fd*)

Load and relocate an ELF file.

Parameters:

fd An open CFS file descriptor.

Returns:

ELFLOADER_OK if loading and relocation worked. Otherwise an error value.

This function loads and relocates an ELF file. The ELF file must have been opened with cfs_open() prior to calling this function.

If the function is able to load the ELF file, a pointer to the process structure in the model is stored in the elfloader_loaded_process variable.

Note:

This function modifies the ELF file opened with cfs_open()! If the contents of the file is required to be intact, the file must be backed up first.

6.22 Architecture specific functionality for the ELF loader.

6.22.1 Detailed Description

The architecture specific functionality for the Contiki ELF loader has to be implemented for each processor type Contiki runs on.

Since the ELF format is slightly different for different processor types, the Contiki ELF loader is divided into two parts: the generic ELF loader module (The Contiki ELF loader) and the architecture specific part (this module). The architecture specific part deals with memory allocation, code and data relocation, and writing the relocated ELF code into program memory.

To port the Contiki ELF loader to a new processor type, this module has to be implemented for the new processor type.

Files

- file elfloader-arch.h

  Header file for the architecture specific parts of the Contiki ELF loader.

Functions

- void ∗ elfloader_arch_allocate_ram (int size)

  Allocate RAM for a new module.

- void ∗ elfloader_arch_allocate_rom (int size)

  Allocate program memory for a new module.

- void elfloader_arch_relocate (int fd, unsigned int sectionoffset, char ∗sectionaddr, struct elf32_rela ∗rela, char ∗addr)

  Perform a relocation.

- void elfloader_arch_write_rom (int fd, unsigned short textoff, unsigned int size, char ∗mem)

  Write to read-only memory (for example the text segment).

6.22.2 Function Documentation

6.22.2.1 void∗ elfloader_arch_allocate_ram (int *size*)

Allocate RAM for a new module.

Parameters:

size The size of the requested memory.

Returns:

A pointer to the allocated RAM

This function is called from the Contiki ELF loader to allocate RAM for the module to be loaded into.

6.22.2.2 void∗ elfloader_arch_allocate_rom (int *size*)

Allocate program memory for a new module.

Parameters:

size The size of the requested memory.

Returns:

A pointer to the allocated program memory

This function is called from the Contiki ELF loader to allocate program memory (typically ROM) for the module to be loaded into.

6.22.2.3 void elfloader_arch_relocate (int *fd*, unsigned int *sectionoffset*, char ∗ *sectionaddr*, struct elf32_rela ∗ *rela*, char ∗ *addr*)

Perform a relocation.

Parameters:

fd The file descriptor for the ELF file.

sectionoffset The file offset at which the relocation can be found.

sectionaddr The section start address (absolute runtime).

rela A pointer to an ELF32 rela structure (struct elf32_rela).

addr The relocated address.

This function is called from the Contiki ELF loader to perform a relocation on a piece of code or data. The relocated address is calculated by the Contiki ELF loader, based on information in the ELF file, and it is the responsibility of this function to patch the executable code. The Contiki ELF loader passes a pointer to an ELF32 rela structure (struct elf32_rela) that contains information about how to patch the code. This information is different from processor to processor.

6.22.2.4 void elfloader_arch_write_rom (int *fd*, unsigned short *textoff*, unsigned int *size*, char ∗ *mem*)

Write to read-only memory (for example the text segment).

Parameters:

fd The file descriptor for the ELF file.

textoff Offset of text segment relative start of file.

size The size of the text segment.

mem A pointer to the where the text segment should be flashed

This function is called from the Contiki ELF loader to write the program code (text segment) of a loaded module into memory. The function is called when all relocations have been performed.

6.23 Protothreads

6.23.1 Detailed Description

Protothreads are a type of lightweight stackless threads designed for severly 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 a 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 do 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.

6.23.2 Authors

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

6.23.3 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.

6.23.4 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!

6.23.5 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.

6.23.6 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
- Protothread semaphores

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.

6.23.7 Define Documentation

6.23.7.1 #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() invokation will be executed each time the protothread is scheduled.

Parameters:

pt A pointer to the protothread control structure.

Definition at line 115 of file pt.h.

6.23.7.2 #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.

Definition at line 127 of file pt.h.

6.23.7.3 #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.

6.23.7.4 #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()

Definition at line 80 of file pt.h.

Referenced by process_start().

6.23.7.5 #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.

Definition at line 229 of file pt.h.

6.23.7.6 #define PT_SCHEDULE(f)

Schedule a protothread.

This function shedules 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.

6.23.7.7 #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.

6.23.7.8 #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:

example-psock-server.c.

Definition at line 100 of file pt.h.

6.23.7.9 #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.

6.23.7.10 #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.

Definition at line 148 of file pt.h.

6.23.7.11 #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.

6.23.7.12 #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.

Definition at line 290 of file pt.h.

6.23.7.13 #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.

6.24 The Contiki file system interface

6.24.1 Detailed Description

The Contiki file system interface (CFS) defines an abstract API for reading directories and for reading and writing files.

The CFS API is intentionally simple. The CFS API is modeled after the POSIX file API, and slightly simplified.

Files

- file cfs.h
  CFS header file.

Defines

- #define CFS_READ 1
  Specify that cfs_open() should open a file for reading.
- #define CFS_WRITE 2
  Specify that cfs_open() should open a file for writing.
- #define CFS_APPEND 4
  Specify that cfs_open() should append written data to the file rather than overwriting it.

Functions

- CCIF int cfs_open (const char ∗name, int flags)
  Open a file.
- CCIF void cfs_close (int fd)
  Close an open file.
- CCIF int cfs_read (int fd, char ∗buf, unsigned int len)
  Read data from an open file.
- CCIF int cfs_write (int fd, char ∗buf, unsigned int len)
  Write data to an open file.
- CCIF int cfs_seek (int fd, unsigned int offset)
  Seek to a specified position in an open file.
- CCIF int cfs_opendir (struct cfs_dir ∗dirp, const char ∗name)
  Open a directory for reading directory entries.
- CCIF int cfs_readdir (struct cfs_dir ∗dirp, struct cfs_dirent ∗dirent)
  Read a directory entry.

- CCIF int cfs_closedir (struct cfs_dir ∗dirp)

  Close a directory opened with cfs_opendir().

6.24.2 Define Documentation

6.24.2.1 #define CFS_APPEND 4

Specify that cfs_open() should append written data to the file rather than overwriting it.

This constant indicates to cfs_open() that a file that should be opened for writing gets written data appended to the end of the file. The default behaviour (without CFS_APPEND) is that the file is overwritten with the new data.

See also:

cfs_open()

Definition at line 107 of file cfs.h.

6.24.2.2 #define CFS_READ 1

Specify that cfs_open() should open a file for reading.

This constant indicates to cfs_open() that a file should be opened for reading. CFS_WRITE should be used if the file is opened for writing, and CFS_READ + CFS_WRITE indicates that the file is opened for both reading and writing.

See also:

cfs_open()

Definition at line 83 of file cfs.h.

6.24.2.3 #define CFS_WRITE 2

Specify that cfs_open() should open a file for writing.

This constant indicates to cfs_open() that a file should be opened for writing. CFS_READ should be used if the file is opened for reading, and CFS_READ + CFS_WRITE indicates that the file is opened for both reading and writing.

See also:

cfs_open()

Definition at line 95 of file cfs.h.

6.24.3 Function Documentation

6.24.3.1 CCIF void cfs_close (int *fd*)

Close an open file.

Parameters:

fd The file descriptor of the open file.

This function closes a file that has previously been opened with cfs_open().

Examples:

test-rudolph0.c, and test-rudolph1.c.

6.24.3.2 CCIF int cfs_closedir (struct cfs_dir * *dirp*)

Close a directory opened with cfs_opendir().

Parameters:

dirp A pointer to a struct cfs_dir that has been opened with cfs_opendir().

See also:

cfs_opendir()
cfs_readdir()

6.24.3.3 CCIF int cfs_open (const char * *name*, int *flags*)

Open a file.

Parameters:

name The name of the file.

flags CFS_READ, or CFS_WRITE, or both.

Returns:

A file descriptor, if the file could be opened, or -1 if the file could not be opened.

This function opens a file and returns a file descriptor for the opened file. If the file could not be opened, the function returns -1. The function can open a file for reading or writing, or both.

An opened file must be closed with cfs_close().

See also:

CFS_READ
CFS_WRITE
cfs_close()

Examples:

test-rudolph0.c, and test-rudolph1.c.

6.24.3.4 CCIF int cfs_opendir (struct cfs_dir * *dirp*, const char * *name*)

Open a directory for reading directory entries.

Parameters:

dirp A pointer to a struct cfs_dir that is filled in by the function.

name The name of the directory.

Returns:

0 or -1 if the directory could not be opened.

See also:

cfs_readdir()
cfs_closedir()

6.24.3.5 CCIF int cfs_read (int *fd*, char ∗ *buf*, unsigned int *len*)

Read data from an open file.

Parameters:

fd The file descriptor of the open file.

buf The buffer in which data should be read from the file.

len The number of bytes that should be read.

Returns:

The number of bytes that was actually read from the file.

This function reads data from an open file into a buffer. The file must have first been opened with cfs_open() and the CFS_READ flag.

Examples:

test-rudolph0.c, and test-rudolph1.c.

6.24.3.6 CCIF int cfs_readdir (struct cfs_dir ∗ *dirp*, struct cfs_dirent ∗ *dirent*)

Read a directory entry.

Parameters:

dirp A pointer to a struct cfs_dir that has been opened with cfs_opendir().

dirent A pointer to a struct cfs_dirent that is filled in by cfs_readdir()

Return values:

0 If a directory entry was read.

0 If no more directory entries can be read.

See also:

cfs_opendir()
cfs_closedir()

6.24.3.7 CCIF int cfs_seek (int *fd*, unsigned int *offset*)

Seek to a specified position in an open file.

Parameters:

fd The file descriptor of the open file.

offset The position in the file.

Returns:

The new position in the file.

This function moves the file position to the specified position in the file. The next byte that is read from or written to the file will be at the position given by the offset parameter.

Examples:

test-rudolph0.c, and test-rudolph1.c.

6.24.3.8 CCIF int cfs_write (int *fd*, char $*$ *buf*, unsigned int *len*)

Write data to an open file.

Parameters:

fd The file descriptor of the open file.

buf The buffer from which data should be written to the file.

len The number of bytes that should be written.

Returns:

The number of bytes that was actually written to the file.

This function reads writes data from a memory buffer to an open file. The file must have been opened with cfs_open() and the CFS_WRITE flag.

Examples:

test-rudolph0.c, and test-rudolph1.c.

6.25 CTK application functions

6.25.1 Detailed Description

The CTK functions used by an application program.

Defines

- #define CTK_SEPARATOR(x, y, w) NULL, NULL, x, y, CTK_WIDGET_SEPARATOR, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0)

  Instantiating macro for the ctk_separator widget.

- #define CTK_BUTTON(x, y, w, text) NULL, NULL, x, y, CTK_WIDGET_BUTTON, w, 1, CTK_-WIDGET_FLAG_INITIALIZER(0) text

  Instantiating macro for the ctk_button widget.

- #define CTK_LABEL(x, y, w, h, text) NULL, NULL, x, y, CTK_WIDGET_LABEL, w, h, CTK_-WIDGET_FLAG_INITIALIZER(0) text,

  Instantiating macro for the ctk_label widget.

- #define CTK_HYPERLINK(x, y, w, text, url) NULL, NULL, x, y, CTK_WIDGET_HYPERLINK, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0) text, url

  Instantiating macro for the ctk_hyperlink widget.

- #define CTK_TEXTENTRY_CLEAR(e)

  Clears a text entry widget and sets the cursor to the start of the text line.

- #define CTK_TEXTENTRY(x, y, w, h, text, len)

  Instantiating macro for the ctk_textentry widget.

- #define CTK_ICON(title, bitmap, textmap)

  Instantiating macro for the ctk_icon widget.

- #define CTK_ICON_ADD(icon, p) ctk_icon_add((struct ctk_widget ∗)icon, p)

  Add an icon to the desktop.

- #define CTK_WIDGET_ADD(win, widg) ctk_widget_add(win, (struct ctk_widget ∗)widg)

  Add a widget to a window.

- #define CTK_WIDGET_FOCUS(win, widg) (win) → focused = (struct ctk_widget ∗)(widg)

  Set focus to a widget.

- #define CTK_WIDGET_REDRAW(widg) ctk_widget_redraw((struct ctk_widget ∗)widg)

  Add a widget to the redraw queue.

- #define CTK_WIDGET_TYPE(w) ((w) → type)

  Obtain the type of a widget.

- #define CTK_WIDGET_SET_WIDTH(widget, width)

  Sets the width of a widget.

- #define CTK_WIDGET_XPOS(w) (((struct ctk_widget ∗)(w)) → x)

  Retrieves the x position of a widget, relative to the window in which the widget is contained.

- #define CTK_WIDGET_SET_XPOS(w, xpos) ((struct ctk_widget ∗)(w)) → x = (xpos)

  Sets the x position of a widget, relative to the window in which the widget is contained.

- #define CTK_WIDGET_YPOS(w) (((struct ctk_widget ∗)(w)) → y)

  Retrieves the y position of a widget, relative to the window in which the widget is contained.

- #define CTK_WIDGET_SET_YPOS(w, ypos) ((struct ctk_widget ∗)(w)) → y = (ypos)

  Sets the y position of a widget, relative to the window in which the widget is contained.

- #define ctk_label_set_height(w, height) (w) → widget.label.h = (height)

  Set the height of a label.

- #define ctk_label_set_text(l, t) (l) → text = (t)

  Set the text of a label.

- #define ctk_button_set_text(b, t) (b) → text = (t)

  Set the text of a button.

Functions

- CCIF void ctk_widget_redraw (struct ctk_widget ∗w)

  Redraws a widget.

- void ctk_desktop_redraw (struct ctk_desktop ∗d)

  Redraw the entire desktop.

- CCIF unsigned char ctk_desktop_width (struct ctk_desktop ∗d)

Gets the width of the desktop.

- unsigned char ctk_desktop_height (struct ctk_desktop ∗d)

  Gets the height of the desktop.

- void ctk_mode_set (unsigned char m)

  Sets the current CTK mode.

- unsigned char ctk_mode_get (void)

  Retrieves the current CTK mode.

- void ctk_icon_add (CC_REGISTER_ARG struct ctk_widget ∗icon, struct process ∗p)

  Add an icon to the desktop.

- void ctk_dialog_open (struct ctk_window ∗d)

  Open a dialog box.

- void ctk_dialog_close (void)

  Close the dialog box, if one is open.

- void ctk_window_open (CC_REGISTER_ARG struct ctk_window ∗w)

  Open a window, or bring window to front if already open.

- void ctk_window_close (struct ctk_window ∗w)

  Close a window if it is open.

- void ctk_window_clear (struct ctk_window ∗w)

  Remove all widgets from a window.

- void ctk_menu_add (struct ctk_menu ∗menu)

  Add a menu to the menu bar.

- void ctk_menu_remove (struct ctk_menu ∗menu)

  Remove a menu from the menu bar.

- void ctk_window_redraw (struct ctk_window ∗w)

  Redraw a window.

- void ctk_window_new (struct ctk_window ∗window, unsigned char w, unsigned char h, char ∗title)

  Create a new window.

- void ctk_dialog_new (CC_REGISTER_ARG struct ctk_window ∗dialog, unsigned char w, unsigned char h)

  Creates a new dialog.

- void ctk_menu_new (CC_REGISTER_ARG struct ctk_menu ∗menu, char ∗title)

  Creates a new menu.

- unsigned char ctk_menuitem_add (CC_REGISTER_ARG struct ctk_menu ∗menu, char ∗name)

  Adds a menu item to a menu.

- void CC_FASTCALL ctk_widget_add (CC_REGISTER_ARG struct ctk_window ∗window, CC_-REGISTER_ARG struct ctk_widget ∗widget)

  Adds a widget to a window.

Variables

- CCIF process_event_t ctk_signal_keypress

  Emitted for every key being pressed.

- CCIF process_event_t ctk_signal_widget_activate

  Emitted when a widget is activated (pressed).

- CCIF process_event_t ctk_signal_widget_select

  Emitted when a widget is selected.

- CCIF process_event_t ctk_signal_menu_activate

  Emitted when a menu item is activated.

- CCIF process_event_t ctk_signal_window_close

  Emitted when a window is closed.

- CCIF process_event_t ctk_signal_pointer_move

  Emitted when the mouse pointer is moved.

- CCIF process_event_t ctk_signal_pointer_button

  Emitted when a mouse button is pressed.

- CCIF process_event_t ctk_signal_button_activate

  Same as ctk_signal_widget_activate.

- CCIF process_event_t ctk_signal_button_hover

  Same as ctk_signal_widget_select.

- CCIF process_event_t ctk_signal_hyperlink_activate

  Emitted when a hyperlink is activated.

- CCIF process_event_t ctk_signal_hyperlink_hover

  Same as ctk_signal_widget_select.

6.25.2 Define Documentation

6.25.2.1 #define CTK_BUTTON(x, y, w, text) NULL, NULL, x, y, CTK_WIDGET_BUTTON, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0) text

Instantiating macro for the ctk_button widget.

This macro is used when instantiating a ctk_button widget and is intended to be used together with a struct assignment like this:

```
struct ctk_button but =
       {CTK_BUTTON(0, 0, 2, "Ok")};
```

Parameters:

- ***x*** The x position of the widget, relative to the widget's window.
- ***y*** The y position of the widget, relative to the widget's window.
- ***w*** The widget's width.
- ***text*** The button text.

Definition at line 141 of file ctk.h.

6.25.2.2 #define ctk_button_set_text(b, t) (b) → text = (t)

Set the text of a button.

Parameters:

- ***b*** The CTK button widget.
- ***t*** The new text of the button.

Definition at line 832 of file ctk.h.

6.25.2.3 #define CTK_HYPERLINK(x, y, w, text, url) NULL, NULL, x, y, CTK_WIDGET_-HYPERLINK, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0) text, url

Instantiating macro for the ctk_hyperlink widget.

This macro is used when instantiating a ctk_hyperlink widget and is intended to be used together with a struct assignment like this:

```
struct ctk_hyperlink hlink =
       {CTK_HYPERLINK(0, 0, 7, "Contiki", "http://dunkels.com/adam/contiki/")};
```

Parameters:

- ***x*** The x position of the widget, relative to the widget's window.
- ***y*** The y position of the widget, relative to the widget's window.
- ***w*** The widget's width.
- ***text*** The hyperlink text.
- ***url*** The hyperlink URL.

Definition at line 203 of file ctk.h.

6.25.2.4 #define CTK_ICON(title, bitmap, textmap)

Value:

```
NULL, NULL, 0, 0, CTK_WIDGET_ICON, 2, 4, CTK_WIDGET_FLAG_INITIALIZER(0) \
 title, PROCESS_NONE, \
 CTK_ICON_BITMAP(bitmap), CTK_ICON_TEXTMAP(textmap)
```

Instantiating macro for the ctk_icon widget.

This macro is used when instantiating a ctk_icon widget and is intended to be used together with a struct assignment like this:

```
struct ctk_icon icon =
       {CTK_ICON("An icon", bitmapptr, textmapptr)};
```

Parameters:

title The icon's text.

bitmap A pointer to the icon's bitmap image.

textmap A pointer to the icon's text version of the bitmap.

Definition at line 313 of file ctk.h.

6.25.2.5 #define CTK_ICON_ADD(icon, p) ctk_icon_add((struct ctk_widget ∗)icon, p)

Add an icon to the desktop.

Parameters:

icon The icon to be added.

p The process ID of the process that owns the icon.

Definition at line 716 of file ctk.h.

Referenced by program_handler_add().

6.25.2.6 #define CTK_LABEL(x, y, w, h, text) NULL, NULL, x, y, CTK_WIDGET_LABEL, w, h, CTK_WIDGET_FLAG_INITIALIZER(0) text,

Instantiating macro for the ctk_label widget.

This macro is used when instantiating a ctk_label widget and is intended to be used together with a struct assignment like this:

```
struct ctk_label lab =
       {CTK_LABEL(0, 0, 5, 1, "Label")};
```

Parameters:

x The x position of the widget, relative to the widget's window.

y The y position of the widget, relative to the widget's window.

w The widget's width.

h The height of the label.

text The label text.

Definition at line 172 of file ctk.h.

6.25.2.7 #define ctk_label_set_height(w, height) (w) → widget.label.h = (height)

Set the height of a label.

Parameters:

w The CTK label widget.

height The new height of the label.

Definition at line 815 of file ctk.h.

6.25.2.8 #define ctk_label_set_text(l, t) (l) → text = (t)

Set the text of a label.

Parameters:

- ***l*** The CTK label widget.
- ***t*** The new text of the label.

Definition at line 824 of file ctk.h.

Referenced by program_handler_load().

6.25.2.9 #define CTK_SEPARATOR(x, y, w) NULL, NULL, x, y, CTK_WIDGET_SEPARATOR, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0)

Instantiating macro for the ctk_separator widget.

This macro is used when instantiating a ctk_separator widget and is intended to be used together with a struct assignment like this:

```
struct ctk_separator sep =
       {CTK_SEPARATOR(0, 0, 23)};
```

Parameters:

- ***x*** The x position of the widget, relative to the widget's window.
- ***y*** The y position of the widget, relative to the widget's window.
- ***w*** The widget's width.

Definition at line 112 of file ctk.h.

6.25.2.10 #define CTK_TEXTENTRY(x, y, w, h, text, len)

Value:

```
NULL, NULL, x, y, CTK_WIDGET_TEXTENTRY, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0) text, len, \
  CTK_TEXTENTRY_NORMAL, 0, 0, NULL
```

Instantiating macro for the ctk_textentry widget.

This macro is used when instantiating a ctk_textentry widget and is intended to be used together with a struct assignment like this:

```
struct ctk_textentry tentry =
       {CTK_TEXTENTRY(0, 0, 30, 1, textbuffer, 50)};
```

Note:

The height of the text entry widget is obsolete and not intended to be used.

Parameters:

- ***x*** The x position of the widget, relative to the widget's window.
- ***y*** The y position of the widget, relative to the widget's window.
- ***w*** The widget's width.
- ***h*** The text entry height (obsolete).
- ***text*** A pointer to the buffer that should be edited.
- ***len*** The length of the text buffer

Definition at line 265 of file ctk.h.

6.25.2.11 #define CTK_TEXTENTRY_CLEAR(e)

Value:

```
do { memset((e)->text, 0, (e)->h * ((e)->len + 1)); \
             (e)->xpos = 0; (e)->ypos = 0; } while(0)
```

Clears a text entry widget and sets the cursor to the start of the text line.

Parameters:

e The text entry widget to be cleared.

Definition at line 230 of file ctk.h.

6.25.2.12 #define CTK_WIDGET_ADD(win, widg) ctk_widget_add(win, (struct ctk_widget ∗)widg)

Add a widget to a window.

Parameters:

win The window to which the widget should be added.

widg The widget to be added.

Definition at line 727 of file ctk.h.

Referenced by ctk_textedit_add().

6.25.2.13 #define CTK_WIDGET_FOCUS(win, widg) (win) → focused = (struct ctk_widget ∗)(widg)

Set focus to a widget.

Parameters:

win The widget's window.

widg The widget

Definition at line 738 of file ctk.h.

Referenced by ctk_textedit_eventhandler().

6.25.2.14 #define CTK_WIDGET_REDRAW(widg) ctk_widget_redraw((struct ctk_widget ∗)widg)

Add a widget to the redraw queue.

Parameters:

widg The widget to be redrawn.

Definition at line 746 of file ctk.h.

Referenced by ctk_textedit_eventhandler().

6.25.2.15 #define CTK_WIDGET_SET_WIDTH(widget, width)

Value:

```
do { \
    ((struct ctk_widget *)(widget))->w = (width); } while(0)
```

Sets the width of a widget.

Parameters:

widget The widget.

width The width of the widget, in characters.

Definition at line 764 of file ctk.h.

6.25.2.16 #define CTK_WIDGET_SET_XPOS(w, xpos) ((struct ctk_widget ∗)(w)) → x = (xpos)

Sets the x position of a widget, relative to the window in which the widget is contained.

Parameters:

w The widget.

xpos The x position of the widget.

Definition at line 783 of file ctk.h.

6.25.2.17 #define CTK_WIDGET_SET_YPOS(w, ypos) ((struct ctk_widget ∗)(w)) → y = (ypos)

Sets the y position of a widget, relative to the window in which the widget is contained.

Parameters:

w The widget.

ypos The y position of the widget.

Definition at line 801 of file ctk.h.

6.25.2.18 #define CTK_WIDGET_TYPE(w) ((w) → type)

Obtain the type of a widget.

Parameters:

w The widget.

Definition at line 755 of file ctk.h.

6.25.2.19 #define CTK_WIDGET_XPOS(w) (((struct ctk_widget ∗)(w)) → x)

Retrieves the x position of a widget, relative to the window in which the widget is contained.

Parameters:

w The widget.

Returns:

The x position of the widget.

Definition at line 774 of file ctk.h.

6.25.2.20 #define CTK_WIDGET_YPOS(w) (((struct ctk_widget *)(w)) → y)

Retrieves the y position of a widget, relative to the window in which the widget is contained.

Parameters:
w The widget.

Returns:
The y position of the widget.

Definition at line 792 of file ctk.h.

6.25.3 Function Documentation

6.25.3.1 unsigned char ctk_desktop_height (struct ctk_desktop * *d*)

Gets the height of the desktop.

Parameters:
d The desktop.

Returns:
The height of the desktop, in characters.

Note:
The d parameter is currently unused and must be set to NULL.

Definition at line 939 of file ctk.c.

6.25.3.2 void ctk_desktop_redraw (struct ctk_desktop * *d*)

Redraw the entire desktop.

Parameters:
d The desktop to be redrawn.

Note:
Currently the parameter d is not used, but must be set to NULL.

Definition at line 602 of file ctk.c.

References PROCESS_CURRENT.

6.25.3.3 unsigned char ctk_desktop_width (struct ctk_desktop * *d*)

Gets the width of the desktop.

Parameters:
d The desktop.

Returns:
The width of the desktop, in characters.

Note:
The d parameter is currently unused and must be set to NULL.

Definition at line 924 of file ctk.c.

6.25.3.4 void ctk_dialog_new (CC_REGISTER_ARG struct ctk_window * *dialog*, unsigned char *w*, unsigned char *h*)

Creates a new dialog.

This function only sets up the internal structure of the ctk_window struct but does not open the dialog. The dialog must be explicitly opened by calling the ctk_dialog_open() function.

Parameters:

dialog The dialog to be created.

w The width of the dialog.

h The height of the dialog.

Definition at line 729 of file ctk.c.

6.25.3.5 void ctk_dialog_open (struct ctk_window * *d*)

Open a dialog box.

Parameters:

d The dialog to be opened.

Definition at line 313 of file ctk.c.

Referenced by program_handler_load().

6.25.3.6 void ctk_icon_add (CC_REGISTER_ARG struct ctk_widget * *icon*, struct process * *p*)

Add an icon to the desktop.

Parameters:

icon The icon to be added.

p The process that owns the icon.

Definition at line 288 of file ctk.c.

References ctk_widget_add(), ctk_widget::icon, and ctk_widget::widget.

6.25.3.7 void ctk_menu_add (struct ctk_menu * *menu*)

Add a menu to the menu bar.

Parameters:

menu The menu to be added.

Note:

Do not call this function multiple times for the same menu, as no check is made to see if the menu already is in the menu bar.

Definition at line 488 of file ctk.c.

References ctk_menus::menus, and ctk_menu::next.

6.25.3.8 void ctk_menu_new (CC_REGISTER_ARG struct ctk_menu ∗ *menu*, char ∗ *title*)

Creates a new menu.

This function sets up the internal structure of the menu, but does not add it to the menubar. Use the function ctk_menu_add() for that purpose.

Parameters:

menu The menu to be created.

title The title of the menu.

Definition at line 747 of file ctk.c.

References ctk_menu::active, ctk_menu::next, ctk_menu::nitems, ctk_menu::title, and ctk_menu::titlelen.

6.25.3.9 void ctk_menu_remove (struct ctk_menu ∗ *menu*)

Remove a menu from the menu bar.

Parameters:

menu The menu to be removed.

Definition at line 516 of file ctk.c.

References ctk_menus::menus, and ctk_menu::next.

6.25.3.10 unsigned char ctk_menuitem_add (CC_REGISTER_ARG struct ctk_menu ∗ *menu*, char ∗ *name*)

Adds a menu item to a menu.

In CTK, each menu item is identified by a number which is unique within each menu. When a menu item is selected, a ctk_menuitem_activated signal is emitted and the menu item number is passed as signal data with the signal.

Parameters:

menu The menu to which the menu item should be added.

name The name of the menu item.

Returns:

The number of the menu item.

Definition at line 773 of file ctk.c.

References ctk_menu::items, ctk_menuitem::title, and ctk_menuitem::titlelen.

Referenced by program_handler_add().

6.25.3.11 unsigned char ctk_mode_get (void)

Retrieves the current CTK mode.

Returns:

The current CTK mode.

Definition at line 275 of file ctk.c.

6.25.3.12 void ctk_mode_set (unsigned char *m*)

Sets the current CTK mode.

The CTK mode can be either CTK_MODE_NORMAL, CTK_MODE_SCREENSAVER or CTK_MODE_EXTERNAL. CTK_MODE_NORMAL is the normal mode, in which keypresses and mouse pointer movements are processed and the screen is redrawn. In CTK_MODE_SCREENSAVER, no screen redraws are performed and the first key press or pointer movement will cause the ctk_signal_screensaver_stop to be emitted. In the CTK_MODE_EXTERNAL mode, key presses and pointer movements are ignored and no screen redraws are made.

Parameters:

m The mode.

Definition at line 264 of file ctk.c.

6.25.3.13 void CC_FASTCALL ctk_widget_add (CC_REGISTER_ARG struct ctk_window * *window*, CC_REGISTER_ARG struct ctk_widget * *widget*)

Adds a widget to a window.

This function adds a widget to a window. The order of which the widgets are added is important, as it sets the order to which widgets are cycled with the widget selection keys.

Parameters:

window The window to which the widhet should be added.

widget The widget to be added.

Definition at line 896 of file ctk.c.

References ctk_window::active, CTK_WIDGET_LABEL, CTK_WIDGET_SEPARATOR, ctk_window::inactive, ctk_window::next, and ctk_widget::window.

Referenced by ctk_icon_add().

6.25.3.14 void ctk_widget_redraw (struct ctk_widget * *widget*)

Redraws a widget.

This function will set a flag which causes the widget to be redrawn next time the CTK process is scheduled.

Parameters:

widget The widget that is to be redrawn.

Note:

This function should usually not be called directly since it requires typecasting of the widget parameter. The wrapper macro CTK_WIDGET_REDRAW() does the required typecast and should be used instead.

Definition at line 873 of file ctk.c.

6.25.3.15 void ctk_window_clear (struct ctk_window * *w*)

Remove all widgets from a window.

Parameters:

w The window to be cleared.

Definition at line 471 of file ctk.c.

References ctk_window::active.

6.25.3.16 void ctk_window_close (struct ctk_window * *w*)

Close a window if it is open.

If the window is not open, this function does nothing.

Parameters:

w The window to be closed.

Definition at line 387 of file ctk.c.

References ctk_window::next, and ctk_window::prev.

6.25.3.17 void ctk_window_new (struct ctk_window * *window*, unsigned char *w*, unsigned char *h*, char * *title*)

Create a new window.

Creates a new window. The memory for the window structure must already be allocated by the caller, and is usually done with a static declaration.

This function sets up the internal structure of the ctk_window struct and creates the move and close buttons, but it does not open the window. The window must be explicitly opened by calling the ctk_window_open() function.

Parameters:

window The window to be created.

w The width of the new window.

h The height of the new window.

title The title of the new window.

Definition at line 707 of file ctk.c.

6.25.3.18 void ctk_window_open (CC_REGISTER_ARG struct ctk_window * *w*)

Open a window, or bring window to front if already open.

Parameters:

w The window to be opened.

Definition at line 338 of file ctk.c.

References ctk_window::next, and ctk_window::prev.

6.25.3.19 void ctk_window_redraw (struct ctk_window * w)

Redraw a window.

This function redraws the window, but only if it is the foremost one on the desktop.

Parameters:

w The window to be redrawn.

Definition at line 628 of file ctk.c.

References ctk_draw_dialog(), ctk_draw_window(), CTK_FOCUS_WINDOW, and ctk_menus::open.

6.25.4 Variable Documentation

6.25.4.1 CCIF process_event_t ctk_signal_hyperlink_activate

Emitted when a hyperlink is activated.

The signal is broadcast to all listeners.

Definition at line 115 of file ctk.c.

6.25.4.2 CCIF process_event_t ctk_signal_keypress

Emitted for every key being pressed.

The key is passed as signal data.

Definition at line 115 of file ctk.c.

Referenced by ctk_textedit_eventhandler().

6.25.4.3 CCIF process_event_t ctk_signal_menu_activate

Emitted when a menu item is activated.

The number of the menu item is passed as signal data.

Definition at line 115 of file ctk.c.

6.25.4.4 CCIF process_event_t ctk_signal_pointer_button

Emitted when a mouse button is pressed.

The button is passed as signal data to the listening process.

Definition at line 115 of file ctk.c.

6.25.4.5 CCIF process_event_t ctk_signal_pointer_move

Emitted when the mouse pointer is moved.

A NULL pointer is passed as signal data and it is up to the listening process to check the position of the mouse using the CTK mouse API.

Definition at line 115 of file ctk.c.

6.25.4.6 CCIF process_event_t ctk_signal_widget_activate

Emitted when a widget is activated (pressed).

A pointer to the widget is passed as signal data.

Definition at line 115 of file ctk.c.

Referenced by ctk_textedit_eventhandler().

6.25.4.7 CCIF process_event_t ctk_signal_widget_select

Emitted when a widget is selected.

A pointer to the widget is passed as signal data.

Definition at line 115 of file ctk.c.

6.25.4.8 CCIF process_event_t ctk_signal_window_close

Emitted when a window is closed.

A pointer to the window is passed as signal data.

Definition at line 115 of file ctk.c.

6.26 CTK graphical user interface

6.26.1 Detailed Description

The Contiki Toolkit (CTK) provides the graphical user interface for the Contiki system.

Files

- file ctk.h

  CTK header file.

- file ctk.c

  The Contiki Toolkit CTK, the Contiki GUI.

- file ctk-draw.h

  CTK screen drawing module interface, ctk-draw.

Modules

- CTK application functions

  The CTK functions used by an application program.

- CTK events
- CTK device driver functions

Functions

- void ctk_mode_set (unsigned char mode)

  Sets the current CTK mode.
- unsigned char ctk_mode_get (void)

  Retrieves the current CTK mode.
- CCIF void ctk_window_new (struct ctk_window ∗window, unsigned char w, unsigned char h, char ∗title)

  Create a new window.
- CCIF void ctk_window_clear (struct ctk_window ∗w)

  Remove all widgets from a window.
- CCIF void ctk_window_close (struct ctk_window ∗w)

  Close a window if it is open.
- CCIF void ctk_window_redraw (struct ctk_window ∗w)

  Redraw a window.
- CCIF void ctk_dialog_open (struct ctk_window ∗d)

  Open a dialog box.
- CCIF void ctk_dialog_close (void)

  Close the dialog box, if one is open.
- CCIF void ctk_menu_add (struct ctk_menu ∗menu)

  Add a menu to the menu bar.
- CCIF void ctk_menu_remove (struct ctk_menu ∗menu)

  Remove a menu from the menu bar.

6.26.2 Function Documentation

6.26.2.1 CCIF void ctk_dialog_open (struct ctk_window ∗ *d*)

Open a dialog box.

Parameters:

d The dialog to be opened.

Definition at line 313 of file ctk.c.

Referenced by program_handler_load().

6.26.2.2 CCIF void ctk_menu_add (struct ctk_menu * *menu*)

Add a menu to the menu bar.

Parameters:

menu The menu to be added.

Note:

Do not call this function multiple times for the same menu, as no check is made to see if the menu already is in the menu bar.

Definition at line 488 of file ctk.c.

References ctk_menus::menus, and ctk_menu::next.

6.26.2.3 CCIF void ctk_menu_remove (struct ctk_menu * *menu*)

Remove a menu from the menu bar.

Parameters:

menu The menu to be removed.

Definition at line 516 of file ctk.c.

References ctk_menus::menus, and ctk_menu::next.

6.26.2.4 unsigned char ctk_mode_get (void)

Retrieves the current CTK mode.

Returns:

The current CTK mode.

Definition at line 275 of file ctk.c.

6.26.2.5 void ctk_mode_set (unsigned char *m*)

Sets the current CTK mode.

The CTK mode can be either CTK_MODE_NORMAL, CTK_MODE_SCREENSAVER or CTK_-MODE_EXTERNAL. CTK_MODE_NORMAL is the normal mode, in which keypresses and mouse pointer movements are processed and the screen is redrawn. In CTK_MODE_SCREENSAVER, no screen redraws are performed and the first key press or pointer movement will cause the ctk_signal_screensaver_-stop to be emitted. In the CTK_MODE_EXTERNAL mode, key presses and pointer movements are ignored and no screen redraws are made.

Parameters:

m The mode.

Definition at line 264 of file ctk.c.

6.26.2.6 CCIF void ctk_window_clear (struct ctk_window * *w*)

Remove all widgets from a window.

Parameters:

w The window to be cleared.

Definition at line 471 of file ctk.c.

References ctk_window::active.

6.26.2.7 CCIF void ctk_window_close (struct ctk_window * *w*)

Close a window if it is open.

If the window is not open, this function does nothing.

Parameters:

w The window to be closed.

Definition at line 387 of file ctk.c.

References ctk_window::next, and ctk_window::prev.

6.26.2.8 CCIF void ctk_window_new (struct ctk_window * *window*, unsigned char *w*, unsigned char *h*, char * *title*)

Create a new window.

Creates a new window. The memory for the window structure must already be allocated by the caller, and is usually done with a static declaration.

This function sets up the internal structure of the ctk_window struct and creates the move and close buttons, but it does not open the window. The window must be explicitly opened by calling the ctk_window_open() function.

Parameters:

window The window to be created.

w The width of the new window.

h The height of the new window.

title The title of the new window.

Definition at line 707 of file ctk.c.

6.26.2.9 CCIF void ctk_window_redraw (struct ctk_window * *w*)

Redraw a window.

This function redraws the window, but only if it is the foremost one on the desktop.

Parameters:

w The window to be redrawn.

Definition at line 628 of file ctk.c.

References ctk_draw_dialog(), ctk_draw_window(), CTK_FOCUS_WINDOW, and ctk_menus::open.

6.27 CTK events

Variables

- process_event_t ctk_signal_keypress
  Emitted for every key being pressed.
- process_event_t ctk_signal_widget_activate
  Emitted when a widget is activated (pressed).
- process_event_t ctk_signal_button_activate
  Same as ctk_signal_widget_activate.
- process_event_t ctk_signal_widget_select
  Emitted when a widget is selected.
- process_event_t ctk_signal_button_hover
  Same as ctk_signal_widget_select.
- process_event_t ctk_signal_hyperlink_activate
  Emitted when a hyperlink is activated.
- process_event_t ctk_signal_hyperlink_hover
  Same as ctk_signal_widget_select.
- process_event_t ctk_signal_menu_activate
  Emitted when a menu item is activated.
- process_event_t ctk_signal_window_close
  Emitted when a window is closed.
- process_event_t ctk_signal_pointer_move
  Emitted when the mouse pointer is moved.
- process_event_t ctk_signal_pointer_button
  Emitted when a mouse button is pressed.

6.27.1 Variable Documentation

6.27.1.1 process_event_t ctk_signal_hyperlink_activate

Emitted when a hyperlink is activated.

The signal is broadcast to all listeners.

Definition at line 115 of file ctk.c.

6.27.1.2 process_event_t ctk_signal_keypress

Emitted for every key being pressed.

The key is passed as signal data.

Definition at line 115 of file ctk.c.

Referenced by ctk_textedit_eventhandler().

6.27.1.3 process_event_t ctk_signal_menu_activate

Emitted when a menu item is activated.

The number of the menu item is passed as signal data.

Definition at line 115 of file ctk.c.

6.27.1.4 process_event_t ctk_signal_pointer_button

Emitted when a mouse button is pressed.

The button is passed as signal data to the listening process.

Definition at line 115 of file ctk.c.

6.27.1.5 process_event_t ctk_signal_pointer_move

Emitted when the mouse pointer is moved.

A NULL pointer is passed as signal data and it is up to the listening process to check the position of the mouse using the CTK mouse API.

Definition at line 115 of file ctk.c.

6.27.1.6 process_event_t ctk_signal_widget_activate

Emitted when a widget is activated (pressed).

A pointer to the widget is passed as signal data.

Definition at line 115 of file ctk.c.

Referenced by ctk_textedit_eventhandler().

6.27.1.7 process_event_t ctk_signal_widget_select

Emitted when a widget is selected.

A pointer to the widget is passed as signal data.

Definition at line 115 of file ctk.c.

6.27.1.8 process_event_t ctk_signal_window_close

Emitted when a window is closed.

A pointer to the window is passed as signal data.

Definition at line 115 of file ctk.c.

6.28 CTK device driver functions

6.28.1 Detailed Description

The CTK device driver functions are divided into two modules, the ctk-draw module and the ctk-arch module. The purpose of the ctk-arch and the ctk-draw modules is to act as an interface between the CTK and the actual hardware of the system on which Contiki is run. The ctk-arch takes care of the keyboard input from the user, and the ctk-draw is responsible for drawing the CTK desktop, windows and user interface widgets onto the actual screen.

More information about the ctk-draw and the ctk-arch modules can be found in the sections The ctk-draw module and The ctk-arch module.

Data Structures

- struct ctk_widget

  The generic CTK widget structure that contains all other widget structures.

- struct ctk_window

  Representation of a CTK window.

- struct ctk_menuitem

  Representation of an individual menu item.

- struct ctk_menu

  Representation of an individual menu.

- struct ctk_menus

  Representation of the menu bar.

Defines

- #define CTK_WIDGET_SEPARATOR 1

  Widget number: The CTK separator widget.

- #define CTK_WIDGET_LABEL 2

  Widget number: The CTK label widget.

- #define CTK_WIDGET_BUTTON 3

  Widget number: The CTK button widget.

- #define CTK_WIDGET_HYPERLINK 4

  Widget number: The CTK hyperlink widget.

- #define CTK_WIDGET_TEXTENTRY 5

  Widget number: The CTK textentry widget.

- #define CTK_WIDGET_BITMAP 6

  Widget number: The CTK bitmap widget.

- #define CTK_WIDGET_ICON 7

  Widget number: The CTK icon widget.

- #define CTK_FOCUS_NONE 0

  Widget focus flag: no focus.

- #define CTK_FOCUS_WIDGET 1

  Widget focus flag: widget has focus.

- #define CTK_FOCUS_WINDOW 2

  Widget focus flag: widget's window is the foremost one.

- #define CTK_FOCUS_DIALOG 4

  Widget focus flag: widget is in a dialog.

Typedefs

- typedef char ctk_arch_key_t

  The keyboard character type of the system.

Functions

- void ctk_draw_init (void)

  The initialization function.

- void ctk_draw_clear (unsigned char clipy1, unsigned char clipy2)

  Clear the screen between the clip bounds.

- void ctk_draw_clear_window (struct ctk_window ∗window, unsigned char focus, unsigned char clipy1, unsigned char clipy2)

  Draw the window background.

- void ctk_draw_window (struct ctk_window ∗window, unsigned char focus, unsigned char clipy1, unsigned char clipy2, unsigned char draw_borders)

  Draw a window onto the screen.

- void ctk_draw_dialog (struct ctk_window ∗dialog)

  Draw a dialog onto the screen.

- void ctk_draw_widget (struct ctk_widget ∗w, unsigned char focus, unsigned char clipy1, unsigned char clipy2)

  Draw a widget on a window.

6.28.1.1 The ctk-draw module

In order to work efficiently even on limited systems, CTK uses a simple coordinate system, where the screen is addressed using character coordinates instead of pixel coordinates. This makes it trivial to implement the coordinate system on a text-based screen, and significantly reduces complexity for pixel based screen systems.

The top left of the screen is (0,0) with x and y coordinates growing downwards and to the right.

It is the responsibility of the ctk-draw module to keep track of the screen size and must implement the two functions ctk_draw_width() and ctk_draw_height(), which are used by the CTK for querying the screen size. The functions must return the width and the height of the ctk-draw screen in character coordinates.

The ctk-draw module is responsible for drawing CTK windows onto the screen through the function ctk_-draw_window().. A pseudo-code implementation of this function might look like this:

```
ctk_draw_window(window, focus, clipy1, clipy2, draw_borders) {
   if(draw_borders) {
      draw_window_borders(window, focus, clipy1, clipy2);
   }
   foreach(widget, window->inactive) {
      ctk_draw_widget(widget, focus, clipy1, clipy2);
   }
   foreach(widget, window->active) {
      if(widget == window->focused) {
         ctk_draw_widget(widget, focus | CTK_FOCUS_WIDGET,
                         clipy1, clipy2);
      } else {
         ctk_draw_widget(widget, focus, clipy1, clipy2);
      }
   }
}
```

Where draw_window_borders() draws the window borders (also between clipy1 and clipy2). The ctk_-draw_widget() function is explained below. Notice how the clipy1 and clipy2 parameters are passed to all other functions; every function needs to know the boundaries within which they are allowed to draw.

In order to aid in implementing a ctk-draw module, a text-based ctk-draw called ctk-conio has already been implemented. It conforms to the Borland conio C library, and a skeleton implementation of said library exists in lib/libconio.c. If a more machine specific ctk-draw module is to be implemented, the instructions in this file should be followed.

6.28.1.2 The ctk-arch module

The ctk-arch module deals with keyboard input from the underlying target system on which Contiki is running. The ctk-arch manages a keyboard input queue that is queried using the two functions ctk_arch_keyavail() and ctk_arch_getkey().

6.28.2 Typedef Documentation

6.28.2.1 typedef char ctk_arch_key_t

The keyboard character type of the system.

The ctk_arch_key_t is usually typedef'd to the char type, but some systems (such as VNC) have a 16-bit key type.

Definition at line 237 of file ctk.h.

6.28.3 Function Documentation

6.28.3.1 void ctk_draw_clear (unsigned char *clipy1*, unsigned char *clipy2*)

Clear the screen between the clip bounds.

This function should clear the screen between the y coordinates "clipy1" and "clipy2", including the line at y coordinate "clipy1", but not the line at y coordinate "clipy2".

Note:

This function may be used to draw a background image (wallpaper) on the desktop; it does not necessarily "clear" the screen.

Parameters:

clipy1 The lower y coordinate of the clip region.

clipy2 The upper y coordinate of the clip region.

6.28.3.2 void ctk_draw_clear_window (struct ctk_window * *window*, unsigned char *focus*, unsigned char *clipy1*, unsigned char *clipy2*)

Draw the window background.

This function will be called by the CTK before a window will be completely redrawn.The function is supposed to draw the window background, excluding window borders as these should be drawn by the function that actually draws the window, between "clipy1" and "clipy2".

Note:

This function does not necessarily have to clear the window - it can be used for drawing a background pattern in the window as well.

Parameters:

window The window for which the background should be drawn.

focus The focus of the window, either CTK_FOCUS_NONE for a background window, or CTK_-FOCUS_WINDOW for the foreground window.

clipy1 The lower y coordinate of the clip region.

clipy2 The upper y coordinate of the clip region.

6.28.3.3 void ctk_draw_dialog (struct ctk_window * *dialog*)

Draw a dialog onto the screen.

In CTK, a dialog is similar to a window, with the only exception being that they are drawn in a different style. Also, since dialogs always are drawn on top of everything else, they do not need to be drawn within any special boundaries.

Note:

This function can usually be implemented so that it uses the same widget drawing code as the ctk_-draw_window() function.

Parameters:

dialog The dialog that is to be drawn.

Referenced by ctk_window_redraw().

6.28.3.4 void ctk_draw_init (void)

The initialization function.

This function is supposed to get the screen ready for drawing, and may be called at more than one time during the operation of the system.

6.28.3.5 void ctk_draw_widget (struct ctk_widget ∗ *w*, unsigned char *focus*, unsigned char *clipy1*, unsigned char *clipy2*)

Draw a widget on a window.

This function is used for drawing a CTK widgets onto the screem is likely to be the most complex function in the ctk-draw module. Still, it is straightforward to implement as it can be written in an incremental fashion, starting with a single widget type and adding more widget types, one at a time.

The ctk-draw module may exploit how the CTK focus constants are defined in order to use a look-up table for the colors. The CTK focus constants are defined in the file ctk/ctk.h as follows:

```
#define CTK_FOCUS_NONE     0
#define CTK_FOCUS_WIDGET   1
#define CTK_FOCUS_WINDOW   2
#define CTK_FOCUS_DIALOG   4
```

This gives the following table:

```
0: CTK_FOCUS_NONE      (Background window, non-focused widget)
1: CTK_FOCUS_WIDGET    (Background window, focused widget)
2: CTK_FOCUS_WINDOW    (Foreground window, non-focused widget)
3: CTK_FOCUS_WINDOW | CTK_FOCUS_WIDGET
                       (Foreground window, focused widget)
4: CTK_FOCUS_DIALOG    (Dialog, non-focused widget)
5: CTK_FOCUS_DIALOG | CTK_FOCUS_WIDGET
                       (Dialog, focused widget)
```

Parameters:

w The widget to be drawn.

focus The focus of the widget.

clipy1 The lower y coordinate of the clip region.

clipy2 The upper y coordinate of the clip region.

6.28.3.6 void ctk_draw_window (struct ctk_window ∗ *window*, unsigned char *focus*, unsigned char *clipy1*, unsigned char *clipy2*, unsigned char *draw_borders*)

Draw a window onto the screen.

This function is called by the CTK when a window should be drawn on the screen. The ctk-draw layer is free to choose how the window will appear on screen; with or without window borders and the style of the borders, with or without transparent window background and how the background shall look, etc.

Parameters:

window The window which is to be drawn.

focus Specifies if the window should be drawn in foreground or background colors and can be either CTK_FOCUS_NONE or CTK_FOCUS_WINDOW. Windows with a focus of CTK_FOCUS_-WINDOW is usually drawn in a brighter color than those with CTK_FOCUS_NONE.

clipy1 Specifies the first lines on screen that actually should be drawn, in screen coordinates (line 1 is the first line below the menus).

clipy2 Specifies the last + 1 line on screen that should be drawn, in screen coordinates (line 1 is the first line below the menus)

Referenced by ctk_window_redraw().

6.29 Timer library

6.29.1 Detailed Description

The Contiki kernel does not provide support for timed events.

Rather, an application that wants to use timers needs to explicitly use the timer library.

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 is not able to post events when a timer expires. The Event timers should be used for this purpose.
The timer library uses the Clock library to measure time. Intervals should be specified in the format used by the clock library.

See also:

Event timers

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.

6.29.2 Function Documentation

6.29.2.1 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.

Definition at line 122 of file timer.c.

References clock_time().

6.29.2.2 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_restart() function.

Parameters:

t A pointer to the timer.

See also:

timer_restart()

Definition at line 85 of file timer.c.

Referenced by etimer_reset().

6.29.2.3 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 105 of file timer.c.

References clock_time().

Referenced by etimer_restart().

6.29.2.4 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.

Definition at line 65 of file timer.c.

References clock_time().

Referenced by etimer_set().

6.30 uIP configuration functions

6.30.1 Detailed Description

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

Defines

- #define uip_sethostaddr(addr)

  Set the IP address of this host.

- #define uip_gethostaddr(addr)

  Get the IP address of this host.

- #define uip_setdraddr(addr)

  Set the default router's IP address.

- #define uip_setnetmask(addr)

  Set the netmask.

- #define uip_getdraddr(addr)

  Get the default router's IP address.

- #define uip_getnetmask(addr)

Get the netmask.

- #define uip_setethaddr(eaddr)

  Specifiy the Ethernet MAC address.

6.30.2 Define Documentation

6.30.2.1 #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 174 of file uip.h.

6.30.2.2 #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 139 of file uip.h.

6.30.2.3 #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 184 of file uip.h.

6.30.2.4 #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 151 of file uip.h.

6.30.2.5 #define uip_setethaddr(eaddr)

Specifiy the Ethernet MAC address.

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

Note:

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

Parameters:

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

Definition at line 134 of file uip_arp.h.

6.30.2.6 #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()

Definition at line 119 of file uip.h.

6.30.2.7 #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 163 of file uip.h.

6.31 uIP initialization functions

6.31.1 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.

6.31.2 Function Documentation

6.31.2.1 void uip_init (void)

uIP initialization function.

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

Definition at line 364 of file uip.c.

References uip_udp_conn::lport, uip_conn::tcpstateflags, UIP_CONNS, UIP_LISTENPORTS, and UIP_-UDP_CONNS.

6.31.2.2 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 166 of file uip.c.

6.32 uIP device driver functions

6.32.1 Detailed Description

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

Defines

- #define uip_input()
  Process an incoming packet.
- #define uip_periodic(conn)
  Periodic processing for a connection identified by its number.
- #define uip_periodic_conn(conn)
  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

- CCIF u8_t uip_buf [UIP_BUFSIZE+2]

  The uIP packet buffer.

6.32.2 Define Documentation

6.32.2.1 #define uip_input()

Process an incoming packet.

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

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

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

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

Note:

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

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

Definition at line 270 of file uip.h.

6.32.2.2 #define uip_periodic(conn)

Periodic processing for a connection identified by its number.

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

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

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

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

Note:

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

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

Parameters:

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

Definition at line 314 of file uip.h.

6.32.2.3 #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 336 of file uip.h.

6.32.2.4 #define uip_poll_conn(conn)

Reuqest 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 350 of file uip.h.

6.32.2.5 #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.

Definition at line 386 of file uip.h.

6.32.2.6 #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 403 of file uip.h.

6.32.3 Variable Documentation

6.32.3.1 CCIF 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 124 of file uip.c.

Referenced by uip_arp_out(), uip_fw_forward(), and uip_ipchksum().

6.33 uIP application functions

6.33.1 Detailed Description

Functions used by an application running of top of uIP.

Defines

- #define uip_datalen()
  The length of any incoming data that is currently avaliable (if avaliable) in the uip_appdata buffer.
- #define uip_urgdatalen()
  The length of any out-of-band data (urgent data) that has arrived on the connection.
- #define uip_close()
  Close the current connection.
- #define uip_abort()
  Abort the current connection.
- #define uip_stop()
  Tell the sending host to stop sending data.
- #define uip_stopped(conn)
  Find out if the current connection has been previously stopped with uip_stop().
- #define uip_restart()
  Restart the current connection, if is has previously been stopped with uip_stop().
- #define uip_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.

- CCIF void uip_send (const void ∗data, int len)

  Send data on the current connection.

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

  Set up a new UDP connection.

6.33.2 Define Documentation

6.33.2.1 #define uip_abort()

Abort the current connection.

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

Definition at line 594 of file uip.h.

6.33.2.2 #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:

example-psock-server.c.

Definition at line 693 of file uip.h.

6.33.2.3 #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.

Definition at line 661 of file uip.h.

6.33.2.4 #define uip_close()

Close the current connection.

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

Definition at line 583 of file uip.h.

6.33.2.5 #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:

example-psock-server.c.

Definition at line 683 of file uip.h.

6.33.2.6 #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:
example-psock-server.c.

Definition at line 673 of file uip.h.

6.33.2.7 #define uip_datalen()

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

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

Definition at line 563 of file uip.h.

6.33.2.8 #define uip_mss()

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

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

Definition at line 750 of file uip.h.

6.33.2.9 #define uip_newdata()

Is new incoming data available?

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

Definition at line 650 of file uip.h.

6.33.2.10 #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.

Definition at line 729 of file uip.h.

6.33.2.11 #define uip_restart()

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

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

Definition at line 623 of file uip.h.

6.33.2.12 #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.

Definition at line 715 of file uip.h.

6.33.2.13 #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 604 of file uip.h.

6.33.2.14 #define uip_timedout()

Has the connection timed out?

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

Examples:
example-psock-server.c.

Definition at line 703 of file uip.h.

6.33.2.15 #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.

Definition at line 800 of file uip.h.

6.33.2.16 #define uip_udp_remove(conn)

Removed a UDP connection.

Parameters:
conn A pointer to the uip_udp_conn structure for the connection.

Definition at line 788 of file uip.h.

6.33.2.17 #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.

Definition at line 813 of file uip.h.

6.33.2.18 #define uip_udpconnection()

Is the current connection a UDP connection?

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

Definition at line 639 of file uip.h.

6.33.2.19 #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 574 of file uip.h.

6.33.3 Function Documentation

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

Note:

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

Since this function requires the port number to be in network byte order, a 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 hot.

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

Returns:

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

Referenced by tcp_connect().

6.33.3.2 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.

Definition at line 514 of file uip.c.

References UIP_LISTENPORTS.

Referenced by tcp_listen().

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

Note:

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

Parameters:

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

len The maximum amount of data bytes to be sent.

Examples:

example-program.c.

Definition at line 1878 of file uip.c.

6.33.3.4 struct uip_udp_conn∗ uip_udp_new (const 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 458 of file uip.c.

References htons(), HTONS, uip_udp_conn::lport, uip_udp_conn::ripaddr, uip_udp_conn::rport, uip_udp_conn::ttl, uip_ipaddr_copy, UIP_TTL, uip_udp_conn, and UIP_UDP_CONNS.

Referenced by udp_new().

6.33.3.5 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 503 of file uip.c.

References UIP_LISTENPORTS.

Referenced by tcp_unlisten().

6.34 uIP conversion functions

6.34.1 Detailed Description

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

Defines

- #define uip_ipaddr_to_quad(a)

  Convert an IP address to four bytes separated by commas.

- #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.

Functions

- CCIF u16_t htons (u16_t val)

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

- CCIF unsigned char uiplib_ipaddrconv (char ∗addrstr, unsigned char ∗addr)

  Convert a textual representation of an IP address to a numerical representation.

6.34.2 Define Documentation

6.34.2.1 #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:

example-program.c, and example-psock-server.c.

Definition at line 1107 of file uip.h.

Referenced by htons(), uip_arp_arpin(), uip_arp_out(), uip_fw_forward(), and uip_udp_new().

6.34.2.2 #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 881 of file uip.h.

6.34.2.3 #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.

Definition at line 867 of file uip.h.

Referenced by udp_broadcast_new().

6.34.2.4 #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.

Definition at line 1034 of file uip.h.

6.34.2.5 #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.

Definition at line 1054 of file uip.h.

6.34.2.6 #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.

Definition at line 1074 of file uip.h.

6.34.2.7 #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.

Definition at line 1094 of file uip.h.

6.34.2.8 #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 933 of file uip.h.

Referenced by uip_arp_arpin(), uip_arp_out(), uip_arp_timer(), uip_fw_forward(), and uip_fw_output().

6.34.2.9 #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.

Definition at line 910 of file uip.h.

Referenced by resolv_conf(), uip_arp_out(), and uip_udp_new().

6.34.2.10 #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 1011 of file uip.h.

6.34.2.11 #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 963 of file uip.h.

Referenced by uip_arp_out().

6.34.2.12 #define uip_ipaddr_to_quad(a)

Convert an IP address to four bytes separated by commas.

Example:

```
uip_ipaddr_t ipaddr;
printf("ipaddr=%d.%d.%d.%d\n", uip_ipaddr_to_quad(&ipaddr));
```

Parameters:

a A pointer to a uip_ipaddr_t.

Definition at line 839 of file uip.h.

6.34.3 Function Documentation

6.34.3.1 CCIF 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 1866 of file uip.c.

References HTONS.

Referenced by uip_chksum(), uip_ipchksum(), and uip_udp_new().

6.34.3.2 CCIF unsigned char uiplib_ipaddrconv (char ∗ *addrstr*, unsigned char ∗ *addr*)

Convert a textual representation of an IP address to a numerical representation.

This function takes a textual representation of an IP address in the form a.b.c.d and converts it into a 4-byte array that can be used by other uIP functions.

Parameters:

addrstr A pointer to a string containing the IP address in textual form.

addr A pointer to a 4-byte array that will be filled in with the numerical representation of the address.

Return values:

0 If the IP address could not be parsed.

Non-zero If the IP address was parsed.

Definition at line 43 of file uiplib.c.

6.35 Variables used in uIP device drivers

6.35.1 Detailed Description

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

Variables

- CCIF u16_t uip_len

  The length of the packet in the uip_buf buffer.

6.35.2 Variable Documentation

6.35.2.1 CCIF 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 140 of file uip.c.

Referenced by tcpip_input(), uip_arp_arpin(), uip_arp_out(), uip_fw_forward(), uip_fw_output(), and uip_split_output().

6.36 Configuration options for uIP

6.36.1 Detailed Description

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

Note:

Contiki does not use the uipopt.h file to configure uIP, but uses a per-port uip-conf.h file that should be edited instead.

Files

- file uipopt.h

  Configuration options for uIP.

Modules

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

6.37 Static configuration options

6.37.1 Detailed Description

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

All of these should be changed to suit your project.

Defines

- #define UIP_FIXEDADDR

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

- #define UIP_PINGADDRCONF

  Ping IP address asignment.

- #define UIP_FIXEDETHADDR

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

6.37.2 Define Documentation

6.37.2.1 #define UIP_FIXEDADDR

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

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

Definition at line 102 of file uipopt.h.

6.37.2.2 #define UIP_FIXEDETHADDR

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

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

Definition at line 132 of file uipopt.h.

6.37.2.3 #define UIP_PINGADDRCONF

Ping IP address asignment.

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

Note:

This works only if UIP_FIXEDADDR is 0.

Definition at line 119 of file uipopt.h.

6.38 IP configuration options

Defines

- #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.

6.38.1 Define Documentation

6.38.1.1 #define UIP_REASSEMBLY

Turn on support for IP packet reassembly.

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

Note:

IP packet reassembly is not heavily tested.

Definition at line 161 of file uipopt.h.

6.38.1.2 #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 146 of file uipopt.h.

Referenced by uip_udp_new().

6.39 UDP configuration options

6.39.1 Detailed Description

Note:

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

Defines

- #define UIP_UDP

  Toggles wether UDP support should be compiled in or not.

- #define UIP_UDP_CHECKSUMS

  Toggles if UDP checksums should be used or not.

- #define UIP_UDP_CONNS

  The maximum amount of concurrent UDP connections.

6.39.2 Define Documentation

6.39.2.1 #define UIP_UDP_CHECKSUMS

Toggles if UDP checksums should be used or not.

Note:

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

Definition at line 205 of file uipopt.h.

6.40 TCP configuration options

Defines

- #define UIP_ACTIVE_OPEN

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

- #define UIP_CONNS

  The maximum number of simultaneously open TCP connections.

- #define UIP_LISTENPORTS

  The maximum number of simultaneously listening TCP ports.

- #define UIP_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.

6.40.1 Define Documentation

6.40.1.1 #define UIP_ACTIVE_OPEN

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

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

Definition at line 243 of file uipopt.h.

6.40.1.2 #define UIP_CONNS

The maximum number of simultaneously open TCP connections.

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

Definition at line 255 of file uipopt.h.

Referenced by uip_init().

6.40.1.3 #define UIP_LISTENPORTS

The maximum number of simultaneously listening TCP ports.

Each listening TCP port requires 2 bytes of memory.

Definition at line 269 of file uipopt.h.

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

6.40.1.4 #define UIP_MAXRTX 8

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

This should not be changed.

Definition at line 298 of file uipopt.h.

6.40.1.5 #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 307 of file uipopt.h.

6.40.1.6 #define UIP_RECEIVE_WINDOW

The size of the advertised receiver's window.

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

Definition at line 327 of file uipopt.h.

6.40.1.7 #define UIP_RTO 3

The initial retransmission timeout counted in timer pulses.

This should not be changed.

Definition at line 290 of file uipopt.h.

6.40.1.8 #define UIP_TCP_MSS (UIP_BUFSIZE - UIP_LLH_LEN - UIP_TCPIP_HLEN)

The TCP maximum segment size.

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

Definition at line 315 of file uipopt.h.

6.40.1.9 #define UIP_TIME_WAIT_TIMEOUT 120

How long a connection should stay in the TIME_WAIT state.

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

Definition at line 338 of file uipopt.h.

6.40.1.10 #define UIP_URGDATA

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

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

Definition at line 283 of file uipopt.h.

6.41 ARP configuration options

Defines

- #define UIP_ARPTAB_SIZE

  The size of the ARP table.

- #define UIP_ARP_MAXAGE 120

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

6.41.1 Define Documentation

6.41.1.1 #define UIP_ARP_MAXAGE 120

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

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

Definition at line 368 of file uipopt.h.

Referenced by uip_arp_timer().

6.41.1.2 #define UIP_ARPTAB_SIZE

The size of the ARP table.

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

Definition at line 359 of file uipopt.h.

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

6.42 General configuration options

Defines

- #define UIP_BUFSIZE

  The size of the uIP packet buffer.

- #define UIP_STATISTICS

  Determines if statistics support should be compiled in.

- #define UIP_LOGGING

  Determines if logging of certain events should be compiled in.

- #define UIP_BROADCAST

  Broadcast support.

- #define UIP_LLH_LEN

  The link level header length.

Functions

- void uip_log (char ∗msg)
  Print out a uIP log message.

6.42.1 Define Documentation

6.42.1.1 #define UIP_BROADCAST

Broadcast support.

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

Definition at line 433 of file uipopt.h.

6.42.1.2 #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 389 of file uipopt.h.

Referenced by uip_split_output().

6.42.1.3 #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 458 of file uipopt.h.

Referenced by uip_arp_out(), uip_fw_forward(), uip_ipchksum(), and uip_split_output().

6.42.1.4 #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 418 of file uipopt.h.

6.42.1.5 #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 403 of file uipopt.h.

6.42.2 Function Documentation

6.42.2.1 void uip_log (char ∗ *msg*)

Print out a uIP log message.

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

6.43 CPU architecture configuration

6.43.1 Detailed Description

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

Defines

- #define UIP_BYTE_ORDER

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

6.43.2 Define Documentation

6.43.2.1 #define UIP_BYTE_ORDER

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

This option can be either UIP_BIG_ENDIAN (Motorola byte order) or UIP_LITTLE_ENDIAN (Intel byte order).

Definition at line 485 of file uipopt.h.

6.44 Appication specific configurations

6.44.1 Detailed Description

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

uIP applications can store the application state within the uip_conn structure by specifying the 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
```

Defines

- #define UIP_APPCALL tcpip_uipcall
  The name of the application function that uIP should call in response to TCP/IP events.

Typedefs

- typedef tcpip_uipstate uip_tcp_appstate_t
  The type of the application state that is to be stored in the uip_conn structure.
- typedef tcpip_uipstate uip_udp_appstate_t
  The type of the application state that is to be stored in the uip_conn structure.

6.44.2 Typedef Documentation

6.44.2.1 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.

Definition at line 82 of file tcpip.h.

6.44.2.2 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.

Definition at line 81 of file tcpip.h.

6.45 uIP Address Resolution Protocol

6.45.1 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.

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.

6.45.2 Function Documentation

6.45.2.1 void uip_arp_arpin (void)

ARP processing for incoming ARP packets.

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

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

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

Definition at line 283 of file uip_arp.c.

References uip_eth_addr::addr, HTONS, uip_ip4addr_t::u8, uip_ipaddr_cmp, and uip_len.

6.45.2.2 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.

Definition at line 366 of file uip_arp.c.

References uip_eth_addr::addr, HTONS, uip_appdata, UIP_ARPTAB_SIZE, uip_buf, uip_ipaddr_cmp, uip_ipaddr_copy, uip_ipaddr_maskcmp, uip_len, and UIP_LLH_LEN.

6.45.2.3 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.

Definition at line 150 of file uip_arp.c.

References UIP_ARP_MAXAGE, UIP_ARPTAB_SIZE, and uip_ipaddr_cmp.

6.46 uIP TCP throughput booster hack

6.46.1 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.

6.46.2 Function Documentation

6.46.2.1 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 uip_acc32, uip_appdata, UIP_BUFSIZE, uip_ipchksum(), uip_len, UIP_LLH_LEN, and uip_-tcpchksum().

6.47 uIP packet forwarding

Files

- file uip-fw.h

  uIP packet forwarding header file.

- file uip-fw.c

  uIP packet forwarding.

Data Structures

- struct uip_fw_netif

  Representation of a uIP network interface.

Defines

- #define UIP_FW_NETIF(ip1, ip2, ip3, ip4, nm1, nm2, nm3, nm4, outputfunc)

  Intantiating macro for a uIP network interface.

- #define uip_fw_setipaddr(netif, addr)

  Set the IP address of a network interface.

- #define uip_fw_setnetmask(netif, addr)

  Set the netmask of a network interface.

- #define UIP_FW_LOCAL

  A non-error message that indicates that a packet should be processed locally.

- #define UIP_FW_OK

  A non-error message that indicates that something went OK.

- #define UIP_FW_FORWARDED

  A non-error message that indicates that a packet was forwarded.

- #define UIP_FW_ZEROLEN

  A non-error message that indicates that a zero-length packet transmission was attempted, and that no packet was sent.

- #define UIP_FW_TOOLARGE

  An error message that indicates that a packet that was too large for the outbound network interface was detected.

- #define UIP_FW_NOROUTE

  An error message that indicates that no suitable interface could be found for an outbound packet.

- #define UIP_FW_DROPPED

  An error message that indicates that a packet that should be forwarded or output was dropped.

Functions

- void uip_fw_init (void)

  Initialize the uIP packet forwarding module.

- u8_t uip_fw_forward (void)

  Forward an IP packet in the uip_buf buffer.

- u8_t uip_fw_output (void)

  Output an IP packet on the correct network interface.

- void uip_fw_register (struct uip_fw_netif ∗netif)

  Register a network interface with the forwarding module.

- void uip_fw_default (struct uip_fw_netif ∗netif)

  Register a default network interface.

- void uip_fw_periodic (void)

  Perform periodic processing.

6.47.1 Define Documentation

6.47.1.1 #define UIP_FW_NETIF(ip1, ip2, ip3, ip4, nm1, nm2, nm3, nm4, outputfunc)

Intantiating macro for a uIP network interface.

Example:

```
struct uip_fw_netif slipnetif =
  {UIP_FW_NETIF(192,168,76,1, 255,255,255,0, slip_output)};
```

Parameters:

ip1,ip2,ip3,ip4 The IP address of the network interface.

nm1,nm2,nm3,nm4 The netmask of the network interface.

outputfunc A pointer to the output function of the network interface.

Definition at line 80 of file uip-fw.h.

6.47.1.2 #define uip_fw_setipaddr(netif, addr)

Set the IP address of a network interface.

Parameters:

netif A pointer to the uip_fw_netif structure for the network interface.

addr A pointer to an IP address.

Definition at line 95 of file uip-fw.h.

6.47.1.3 #define uip_fw_setnetmask(netif, addr)

Set the netmask of a network interface.

Parameters:

netif A pointer to the uip_fw_netif structure for the network interface.

addr A pointer to an IP address representing the netmask.

Definition at line 107 of file uip-fw.h.

6.47.2 Function Documentation

6.47.2.1 void uip_fw_default (struct uip_fw_netif * *netif*)

Register a default network interface.

All packets that don't go out on any of the other interfaces will be routed to the default interface.

Parameters:

netif A pointer to the network interface that is to be registered.

Definition at line 518 of file uip-fw.c.

6.47.2.2 u8_t uip_fw_forward (void)

Forward an IP packet in the uip_buf buffer.

Returns:

UIP_FW_FORWARDED if the packet was forwarded, UIP_FW_LOCAL if the packet should be processed locally.

Definition at line 407 of file uip-fw.c.

References HTONS, uip_appdata, uip_buf, UIP_FW_FORWARDED, UIP_FW_LOCAL, uip_fw_output(), uip_ipaddr_cmp, uip_len, and UIP_LLH_LEN.

6.47.2.3 u8_t uip_fw_output (void)

Output an IP packet on the correct network interface.

The IP packet should be present in the uip_buf buffer and its length in the global uip_len variable.

Return values:

UIP_FW_ZEROLEN Indicates that a zero-length packet transmission was attempted and that no packet was sent.

UIP_FW_NOROUTE No suitable network interface could be found for the outbound packet, and the packet was not sent.

Returns:

The return value from the actual network interface output function is passed unmodified as a return value.

Definition at line 360 of file uip-fw.c.

References uip_fw_netif::next, uip_fw_netif::output, UIP_FW_NOROUTE, UIP_FW_OK, UIP_FW_-ZEROLEN, uip_ipaddr_cmp, and uip_len.

Referenced by uip_fw_forward().

6.47.2.4 void uip_fw_register (struct uip_fw_netif * *netif*)

Register a network interface with the forwarding module.

Parameters:

netif A pointer to the network interface that is to be registered.

Definition at line 501 of file uip-fw.c.

References uip_fw_netif::next.

6.48 uIP hostname resolver functions

6.48.1 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.

The event resolv_event_found is posted when a hostname has been resolved. It is up to the receiving process to determine if the correct hostname has been found by calling the resolv_lookup() function with the hostname.

Files

- file resolv.c

    DNS host name to IP address resolver.

Functions

- void resolv_query (char *name)

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

- u16_t * resolv_lookup (char *name)

    Look up a hostname in the array of known hostnames.

- uip_ipaddr_t * resolv_getserver (void)

Obtain the currently configured DNS server.

- void resolv_conf (const uip_ipaddr_t *dnsserver)

  Configure a DNS server.

Variables

- process_event_t resolv_event_found

  Event that is broadcasted when a DNS name has been resolved.

6.48.2 Function Documentation

6.48.2.1 void resolv_conf (const uip_ipaddr_t * *dnsserver*)

Configure a DNS server.

Parameters:

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

Definition at line 479 of file resolv.c.

References process_post(), and uip_ipaddr_copy.

6.48.2.2 uip_ipaddr_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.

Definition at line 463 of file resolv.c.

References uip_udp_conn::ripaddr.

6.48.2.3 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.

Definition at line 437 of file resolv.c.

6.48.2.4 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.

Definition at line 389 of file resolv.c.

References tcpip_poll_udp().

6.49 Protosockets library

6.49.1 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 block. Automatic variables (stack variables) are not necessarily 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

  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.

6.49.2 Define Documentation

6.49.2.1 #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:

example-psock-server.c.

Definition at line 165 of file psock.h.

6.49.2.2 #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:

example-psock-server.c.

Definition at line 242 of file psock.h.

6.49.2.3 #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 315 of file psock.h.

6.49.2.4 #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.

Examples:

example-psock-server.c.

Definition at line 288 of file psock.h.

6.49.2.5 #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:

example-psock-server.c.

Definition at line 332 of file psock.h.

6.49.2.6 #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.

Definition at line 304 of file psock.h.

6.49.2.7 #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 226 of file psock.h.

6.49.2.8 #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.

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

Examples:

example-psock-server.c.

Definition at line 151 of file psock.h.

6.49.2.9 #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 346 of file psock.h.

6.49.2.10 #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 257 of file psock.h.

6.49.2.11 #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 specifieed 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:

example-psock-server.c.

Definition at line 275 of file psock.h.

6.49.2.12 #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:

example-psock-server.c.

Definition at line 185 of file psock.h.

6.49.2.13 #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:

example-psock-server.c.

Definition at line 198 of file psock.h.

6.49.2.14 #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_NEWDATA(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 379 of file psock.h.

6.50 The Contiki/uIP interface

6.50.1 Detailed Description

TCP/IP support in Contiki is implemented using the uIP TCP/IP stack. For sending and receiving data, Contiki uses the functions provided by the uIP module, but Contiki adds a set of functions for connection management. The connection management functions make sure that the uIP TCP/IP connections are connected to the correct process.

Contiki also includes an optional protosocket library that provides an API similar to the BSD socket API.

See also:

The uIP TCP/IP stack
Protosockets library

Files

- file tcpip.h

  Header for the Contiki/uIP interface.

TCP functions

- CCIF void tcp_attach (struct uip_conn ∗conn, void ∗appstate)

  Attach a TCP connection to the current process.

- CCIF void tcp_listen (u16_t port)

  Open a TCP port.

- CCIF void tcp_unlisten (u16_t port)

  Close a listening TCP port.

- CCIF struct uip_conn ∗ tcp_connect (uip_ipaddr_t ∗ripaddr, u16_t port, void ∗appstate)

  Open a TCP connection to the specified IP address and port.

- void tcpip_poll_tcp (struct uip_conn ∗conn)

  Cause a specified TCP connection to be polled.

UDP functions

- #define udp_bind(conn, port) uip_udp_bind(conn, port)

  Bind a UDP connection to a local port.

- void udp_attach (struct uip_udp_conn ∗conn, void ∗appstate)

  Attach the current process to a UDP connection.

- CCIF struct uip_udp_conn ∗ udp_new (const uip_ipaddr_t ∗ripaddr, u16_t port, void ∗appstate)

  Create a new UDP connection.

- uip_udp_conn ∗ udp_broadcast_new (u16_t port, void ∗appstate)

  Create a new UDP broadcast connection.

- CCIF void tcpip_poll_udp (struct uip_udp_conn ∗conn)

  Cause a specified UDP connection to be polled.

TCP/IP packet processing

- CCIF void tcpip_input (void)

  Deliver an incoming packet to the TCP/IP stack.

Variables

- CCIF process_event_t tcpip_event

  The uIP event.

6.50.2 Define Documentation

6.50.2.1 #define udp_bind(conn, port) uip_udp_bind(conn, port)

Bind a UDP connection to a local port.

This function binds a UDP conncetion to a specified local port.

When a connction is created with udp_new(), it gets a local port number assigned automatically. If the application needs to bind the connection to a specified local port, this function should be used.

Note:

The port number must be provided in network byte order so a conversion with HTONS() usually is necessary.

Parameters:

conn A pointer to the UDP connection that is to be bound.

port The port number in network byte order to which to bind the connection.

Definition at line 259 of file tcpip.h.

Referenced by udp_broadcast_new().

6.50.3 Function Documentation

6.50.3.1 CCIF void tcp_attach (struct uip_conn * *conn*, void * *appstate*)

Attach a TCP connection to the current process.

This function attaches the current process to a TCP connection. Each TCP connection must be attached to a process in order for the process to be able to receive and send data. Additionally, this function can add a pointer with connection state to the connection.

Parameters:

conn A pointer to the TCP connection.

appstate An opaque pointer that will be passed to the process whenever an event occurs on the connection.

Definition at line 161 of file tcpip.c.

References PROCESS_CURRENT.

6.50.3.2 CCIF struct uip_conn* tcp_connect (uip_ipaddr_t * *ripaddr*, u16_t *port*, void * *appstate*)

Open a TCP connection to the specified IP address and port.

This function opens a TCP connection to the specified port at the host specified with an IP address. Additionally, an opaque pointer can be attached to the connection. This pointer will be sent together with uIP events to the process.

Note:

The port number must be provided in network byte order so a conversion with HTONS() usually is necessary.
This function will only create the connection. The connection is not opened directly. uIP will try to open the connection the next time the uIP stack is scheduled by Contiki.

Parameters:

ripaddr Pointer to the IP address of the remote host.

port Port number in network byte order.

appstate Pointer to application defined data.

Returns:

A pointer to the newly created connection, or NULL if memory could not be allocated for the connection.

Definition at line 107 of file tcpip.c.

References uip_conn::appstate, PROCESS_CURRENT, tcpip_poll_tcp(), and uip_connect().

6.50.3.3 CCIF void tcp_listen (u16_t *port*)

Open a TCP port.

This function opens a TCP port for listening. When a TCP connection request occurs for the port, the process will be sent a tcpip_event with the new connection request.

Note:

Port numbers must always be given in network byte order. The functions HTONS() and htons() can be used to convert port numbers from host byte order to network byte order.

Parameters:

port The port number in network byte order.

Examples:

example-psock-server.c.

Definition at line 143 of file tcpip.c.

References PROCESS_CURRENT, uip_listen(), and UIP_LISTENPORTS.

6.50.3.4 CCIF void tcp_unlisten (u16_t *port*)

Close a listening TCP port.

This function closes a listening TCP port.

Note:

Port numbers must always be given in network byte order. The functions HTONS() and htons() can be used to convert port numbers from host byte order to network byte order.

Parameters:

port The port number in network byte order.

Definition at line 125 of file tcpip.c.

References PROCESS_CURRENT, UIP_LISTENPORTS, and uip_unlisten().

6.50.3.5 CCIF void tcpip_input (void)

Deliver an incoming packet to the TCP/IP stack.

This function is called by network device drivers to deliver an incoming packet to the TCP/IP stack. The incoming packet must be present in the uip_buf buffer, and the length of the packet must be in the global uip_len variable.

Examples:

example-packet-drv.c.

Definition at line 325 of file tcpip.c.

References process_post_synch(), and uip_len.

6.50.3.6 void tcpip_poll_tcp (struct uip_conn ∗ *conn*)

Cause a specified TCP connection to be polled.

This function causes uIP to poll the specified TCP connection. The function is used when the application has data that is to be sent immediately and do not wish to wait for the periodic uIP polling mechanism.

Parameters:

conn A pointer to the TCP connection that should be polled.

Definition at line 338 of file tcpip.c.

References process_post().

Referenced by tcp_connect().

6.50.3.7 CCIF void tcpip_poll_udp (struct uip_udp_conn ∗ *conn*)

Cause a specified UDP connection to be polled.

This function causes uIP to poll the specified UDP connection. The function is used when the application has data that is to be sent immediately and do not wish to wait for the periodic uIP polling mechanism.

Parameters:

conn A pointer to the UDP connection that should be polled.

Examples:

example-program.c.

Definition at line 332 of file tcpip.c.

References process_post().

Referenced by resolv_query().

6.50.3.8 struct void udp_attach (struct uip_udp_conn ∗ *conn*, void ∗ *appstate*)

Attach the current process to a UDP connection.

This function attaches the current process to a UDP connection. Each UDP connection must have a process attached to it in order for the process to be able to receive and send data over the connection. Additionally, this function can add a pointer with connection state to the connection.

Parameters:

conn A pointer to the UDP connection.

appstate An opaque pointer that will be passed to the process whenever an event occurs on the connection.

Definition at line 172 of file tcpip.c.

References PROCESS_CURRENT.

6.50.3.9 struct uip_udp_conn∗ udp_broadcast_new (u16_t *port*, void ∗ *appstate*)

Create a new UDP broadcast connection.

This function creates a new (link-local) broadcast UDP connection to a specified port.

Parameters:

port Port number in network byte order.

appstate Pointer to application defined data.

Returns:

A pointer to the newly created connection, or NULL if memory could not be allocated for the connection.

Examples:
example-program.c.

Definition at line 201 of file tcpip.c.

References udp_bind, udp_new(), and uip_ipaddr.

6.50.3.10 CCIF struct uip_udp_conn* udp_new (const uip_ipaddr_t * *ripaddr*, u16_t *port*, void * *appstate*)

Create a new UDP connection.

This function creates a new UDP connection with the specified remote endpoint.

Note:
The port number must be provided in network byte order so a conversion with HTONS() usually is necessary.

See also:
udp_bind()

Parameters:
ripaddr Pointer to the IP address of the remote host.

port Port number in network byte order.

appstate Pointer to application defined data.

Returns:
A pointer to the newly created connection, or NULL if memory could not be allocated for the connection.

Definition at line 183 of file tcpip.c.

References uip_udp_conn::appstate, PROCESS_CURRENT, and uip_udp_new().

Referenced by udp_broadcast_new().

6.50.4 Variable Documentation

6.50.4.1 CCIF process_event_t tcpip_event

The uIP event.

This event is posted to a process whenever a uIP event has occured.

Definition at line 42 of file tcpip.c.

6.51 Anonymous best-effort local area broadcast

6.51.1 Detailed Description

The abc module sends packets to all local area neighbors. The abc module adds no headers to outgoing packets.

6.51.2 Channels

The abc module uses 1 channel.

Files

- file abc.h

  Header file for the Rime module Anonymous BroadCast (abc).

- file abc.c

  Anonymous best-effort local area Broad Cast (abc).

Data Structures

- struct abc_callbacks

  Callback structure for abc.

- struct abc_callbacks

  Callback structure for abc.

Functions

- void abc_open (struct abc_conn ∗c, u16_t channel, const struct abc_callbacks ∗u)

  Set up an anonymous best-effort broadcast connection.

- void abc_close (struct abc_conn ∗c)

  Close an abc connection.

- int abc_send (struct abc_conn ∗c)

  Send an anonymous best-effort broadcast packet.

- void abc_input_packet (void)

  Internal Rime function: Pass a packet to the abc layer.

6.51.3 Function Documentation

6.51.3.1 void abc_close (struct abc_conn ∗ *c*)

Close an abc connection.

Parameters:

c A pointer to a struct abc_conn

This function closes an abc connection that has previously been opened with abc_open().

This function typically is called as an exit handler.

Examples:
test-abc.c.

Definition at line 78 of file abc.c.

References list_remove().

Referenced by ibc_close().

6.51.3.2 void abc_input_packet (void)

Internal Rime function: Pass a packet to the abc layer.

This function is used internally by Rime to pass packets to the abc layer. Should never be called directly.

Definition at line 99 of file abc.c.

References list_head(), rimeaddr_node_addr, rimebuf_dataptr(), and rimebuf_hdrreduce().

6.51.3.3 void abc_open (struct abc_conn * *c*, u16_t *channel*, const struct abc_callbacks * *u*)

Set up an anonymous best-effort broadcast connection.

Parameters:
- ***c*** A pointer to a struct abc_conn
- ***channel*** The channel on which the connection will operate
- ***u*** A struct abc_callbacks with function pointers to functions that will be called when a packet has been received

This function sets up an abc connection on the specified channel. The caller must have allocated the memory for the struct abc_conn, usually by declaring it as a static variable.

The struct abc_callbacks pointer must point to a structure containing a pointer to a function that will be called when a packet arrives on the channel.

Definition at line 68 of file abc.c.

References list_add().

Referenced by ibc_open(), and sabc_open().

6.51.3.4 int abc_send (struct abc_conn * *c*)

Send an anonymous best-effort broadcast packet.

Parameters:
- ***c*** The abc connection on which the packet should be sent

Return values:
- ***Non-zero*** if the packet could be sent, zero otherwise

This function sends an anonymous best-effort broadcast packet. The packet must be present in the rimebuf before this function is called.

The parameter c must point to an abc connection that must have previously been set up with abc_open().

Examples:
test-abc.c.

Definition at line 84 of file abc.c.

References rimeaddr_node_addr, rimebuf_hdralloc(), and rimebuf_hdrptr().

Referenced by ibc_send().

6.52 Callback timer

6.52.1 Detailed Description

The ctimer module provides a timer mechanism that calls a specified C function when a ctimer expires.

Files

- file ctimer.h

  Header file for the callback timer.

- file ctimer.c

  Callback timer implementation.

6.53 Identified best-effort local area broadcast

6.53.1 Detailed Description

The ibc module sends packets to all local area neighbors with an a header that identifies the sender.

6.53.2 Channels

The ibc module uses 1 channel.

Files

- file ibc.h

  Header file for identified best-effort local area broadcast.

- file ibc.c

  Identified best-effort local area broadcast (ibc).

Data Structures

- struct ibc_callbacks

  Callback structure for abc.

- struct ibc_callbacks

  Callback structure for abc.

Functions

- void ibc_open (struct ibc_conn *c, u16_t channel, const struct ibc_callbacks *u)

  Set up an identified best-effort broadcast connection.

- void ibc_close (struct ibc_conn *c)

  Close an ibc connection.

- int ibc_send (struct ibc_conn *c)

  Send an anonymous best-effort broadcast packet.

6.53.3 Function Documentation

6.53.3.1 void ibc_close (struct ibc_conn * *c*)

Close an ibc connection.

Parameters:

c A pointer to a struct ibc_conn

This function closes an ibc connection that has previously been opened with ibc_open().

This function typically is called as an exit handler.

Definition at line 87 of file ibc.c.

References abc_close().

6.53.3.2 void ibc_open (struct ibc_conn * *c*, u16_t *channel*, const struct ibc_callbacks * *u*)

Set up an identified best-effort broadcast connection.

Parameters:

c A pointer to a struct ibc_conn

channel The channel on which the connection will operate

u A struct ibc_callbacks with function pointers to functions that will be called when a packet has been received

This function sets up an ibc connection on the specified channel. The caller must have allocated the memory for the struct ibc_conn, usually by declaring it as a static variable.

The struct ibc_callbacks pointer must point to a structure containing a pointer to a function that will be called when a packet arrives on the channel.

Definition at line 79 of file ibc.c.

References abc_open().

6.53.3.3 int ibc_send (struct ibc_conn * *c*)

Send an anonymous best-effort broadcast packet.

Parameters:

c The ibc connection on which the packet should be sent

Return values:

Non-zero if the packet could be sent, zero otherwise

This function sends an anonymous best-effort broadcast packet. The packet must be present in the rimebuf before this function is called.

The parameter c must point to an abc connection that must have previously been set up with ibc_open().

Definition at line 93 of file ibc.c.

References abc_send(), rimeaddr_copy(), rimeaddr_node_addr, rimebuf_hdralloc(), and rimebuf_hdrptr().

6.54 Mesh routing

6.54.1 Detailed Description

The mesh module sends packets using multi-hop routing to a specified receiver somewhere in the network.

6.54.2 Channels

The mesh module uses 3 channel; one for the multi-hop forwarding (mh) and two for the route disovery (route-discovery).

Files

- file mesh.h

  Header file for the Rime mesh routing protocol.

- file mesh.c

  A mesh routing protocol.

Data Structures

- struct mesh_callbacks

  Mesh callbacks.

- struct mesh_callbacks

  Mesh callbacks.

Functions

- void mesh_open (struct mesh_conn ∗c, u16_t channels, const struct mesh_callbacks ∗callbacks)

  Open a mesh connection.

- void mesh_close (struct mesh_conn ∗c)

  Close an mesh connection.

- int mesh_send (struct mesh_conn ∗c, rimeaddr_t ∗dest)

Send a mesh packet.

6.54.3 Function Documentation

6.54.3.1 void mesh_close (struct mesh_conn ∗ *c*)

Close an mesh connection.

Parameters:

c A pointer to a struct mesh_conn

This function closes an mesh connection that has previously been opened with mesh_open().

This function typically is called as an exit handler.

Examples:

test-meshroute.c.

Definition at line 129 of file mesh.c.

6.54.3.2 void mesh_open (struct mesh_conn ∗ *c*, u16_t *channels*, const struct mesh_callbacks ∗ *callbacks*)

Open a mesh connection.

Parameters:

c A pointer to a struct mesh_conn

channels The channels on which the connection will operate; mesh uses 3 channels

callbacks Pointer to callback structure

This function sets up a mesh connection on the specified channel. The caller must have allocated the memory for the struct mesh_conn, usually by declaring it as a static variable.

The struct mesh_callbacks pointer must point to a structure containing function pointers to functions that will be called when a packet arrives on the channel.

Definition at line 117 of file mesh.c.

References CLOCK_SECOND.

6.54.3.3 int mesh_send (struct mesh_conn ∗ *c*, rimeaddr_t ∗ *dest*)

Send a mesh packet.

Parameters:

c The mesh connection on which the packet should be sent

dest The address of the final destination of the packet

Return values:

Non-zero if the packet could be queued for sending, zero otherwise

This function sends a mesh packet. The packet must be present in the rimebuf before this function is called.

The parameter c must point to an abc connection that must have previously been set up with mesh_open().

Examples:
test-meshroute.c.

Definition at line 136 of file mesh.c.

References rimeaddr_copy().

6.55 Best-effort multihop forwarding

6.55.1 Detailed Description

The mh module implements a multihop forwarding mechanism. Routes must have already been setup with the route_add() function. Setting up routes is done with another Rime module such as the route-discovery module.

6.55.2 Channels

The mh module uses 1 channel.

Files

- file mh.h
  Multihop forwarding header file.
- file mh.c
  Multihop forwarding.

6.56 Rime neighbor management

6.56.1 Detailed Description

The neighbor module manages the neighbor table.

Files

- file neighbor.h
  Header file for the Contiki radio neighborhood management.
- file neighbor.c
  Radio neighborhood management.

6.57 Best-effort network flooding

6.57.1 Detailed Description

The nf module does best-effort flooding.

6.57.2 Channels

The nf module uses 1 channel.

Files

- file nf.h

  Header file for the best-effort network flooding (nf).

- file nf.c

  Best-effort network flooding (nf).

6.58 Rime queue buffer management

6.58.1 Detailed Description

The queuebuf module handles buffers that are queued.

Files

- file queuebuf.h

  Header file for the Rime queue buffer management.

- file queuebuf.c

  Implementation of the Rime queue buffers.

6.59 Rime addresses

6.59.1 Detailed Description

The rimeaddr module is an abstract repressentation of addresses in Rime.

Files

- file rimeaddr.h

  Header file for the Rime address repressentation.

- file rimeaddr.c

  Functions for manipulating Rime addresses.

Functions

- void rimeaddr_copy (rimeaddr_t ∗dest, const rimeaddr_t ∗from)

  Copy a Rime address.

- int rimeaddr_cmp (const rimeaddr_t ∗addr1, const rimeaddr_t ∗addr2)

  Compare two Rime addresses.

- void rimeaddr_set_node_addr (rimeaddr_t ∗addr)

  Set the address of the current node.

Variables

- rimeaddr_t rimeaddr_node_addr

  The Rime address of the node.

- const rimeaddr_t rimeaddr_null

  The null Rime address.

- rimeaddr_t rimeaddr_node_addr

  The Rime address of the node.

- const rimeaddr_t rimeaddr_null

  The null Rime address.

6.59.2 Function Documentation

6.59.2.1 int rimeaddr_cmp (const rimeaddr_t ∗ *addr1*, const rimeaddr_t ∗ *addr2*)

Compare two Rime addresses.

Parameters:

addr1 The first address

addr2 The second address

Returns:

Non-zero if the addresses are the same, zero if they are different

This function compares two Rime addresses and returns the result of the comparison. The function acts like the '==' operator and returns non-zero if the addresses are the same, and zero if the addresses are different.

Definition at line 59 of file rimeaddr.c.

6.59.2.2 void rimeaddr_copy (rimeaddr_t ∗ *dest*, const rimeaddr_t ∗ *from*)

Copy a Rime address.

Parameters:

dest The destination

from The source

This function copies a Rime address from one location to another.

Definition at line 53 of file rimeaddr.c.

Referenced by ibc_send(), mesh_send(), and rimeaddr_set_node_addr().

6.59.2.3 void rimeaddr_set_node_addr (rimeaddr_t * *addr*)

Set the address of the current node.

Parameters:

addr The address

This function sets the Rime address of the node.

Definition at line 65 of file rimeaddr.c.

References rimeaddr_copy(), and rimeaddr_node_addr.

6.59.3 Variable Documentation

6.59.3.1 rimeaddr_t rimeaddr_node_addr

The Rime address of the node.

This variable contains the Rime address of the node. This variable should not be changed directly; rather, the rimeaddr_set_node_addr() function should be used.

Definition at line 48 of file rimeaddr.c.

Referenced by abc_input_packet(), abc_send(), ibc_send(), and rimeaddr_set_node_addr().

6.59.3.2 rimeaddr_t rimeaddr_node_addr

The Rime address of the node.

This variable contains the Rime address of the node. This variable should not be changed directly; rather, the rimeaddr_set_node_addr() function should be used.

Definition at line 48 of file rimeaddr.c.

Referenced by abc_input_packet(), abc_send(), ibc_send(), and rimeaddr_set_node_addr().

6.59.3.3 const rimeaddr_t rimeaddr_null

The null Rime address.

This variable contains the null Rime address. The null address is used in route tables to indicate that the table entry is unused. Nodes with no configured address has the null address. Nodes with their node address set to the null address will have problems communicating with other nodes.

Definition at line 49 of file rimeaddr.c.

6.59.3.4 const rimeaddr_t rimeaddr_null

The null Rime address.

This variable contains the null Rime address. The null address is used in route tables to indicate that the table entry is unused. Nodes with no configured address has the null address. Nodes with their node address set to the null address will have problems communicating with other nodes.

Definition at line 49 of file rimeaddr.c.

6.60 Rime buffer management

6.60.1 Detailed Description

The rimebuf module does Rime's buffer management.

Files

- file rimebuf.h

  Header file for the Rime buffer (rimebuf) management.

- file rimebuf.c

  Rime buffer (rimebuf) management.

Defines

- #define RIMEBUF_SIZE 128

  The size of the rimebuf, in bytes.

- #define RIMEBUF_HDR_SIZE 32

  The size of the rimebuf header, in bytes.

Functions

- void rimebuf_clear (void)

  Clear and reset the rimebuf.

- void * rimebuf_dataptr (void)

  Get a pointer to the data in the rimebuf.

- void * rimebuf_hdrptr (void)

  Get a pointer to the header in the rimebuf, for outbound packets.

- u8_t rimebuf_hdrlen (void)

  Get the length of the header in the rimebuf, for outbound packets.

- u16_t rimebuf_datalen (void)

  Get the length of the data in the rimebuf.

- u16_t rimebuf_totlen (void)

  Get the total length of the header and data in the rimebuf.

- void rimebuf_set_datalen (u16_t len)

  Set the length of the data in the rimebuf.

- void rimebuf_reference (void *ptr, u16_t len)

  Point the rimebuf to external data.

- int rimebuf_is_reference (void)

  Check if the rimebuf references external data.

- void * rimebuf_reference_ptr (void)

  Get a pointer to external data referenced by the rimebuf.

- void rimebuf_compact (void)

  Compact the rimebuf.

- int rimebuf_copyfrom (u8_t *from, u16_t len)

  Copy from external data into the rimebuf.

- int rimebuf_copyto (u8_t *to)

  Copy the entire rimebuf to an external buffer.

- int rimebuf_copyto_hdr (u8_t *to)

  Copy the header portion of the rimebuf to an external buffer.

- int rimebuf_hdralloc (int size)

  Extend the header of the rimebuf, for outbound packets.

- int rimebuf_hdrreduce (int size)

  Reduce the header in the rimebuf, for incoming packets.

6.60.2 Function Documentation

6.60.2.1 void rimebuf_clear (void)

Clear and reset the rimebuf.

This function clears the rimebuf and resets all internal state pointers (header size, header pointer, external data pointer). It is used before preparing a packet in the rimebuf.

Examples:
test-treeroute.c.

Definition at line 69 of file rimebuf.c.

References RIMEBUF_HDR_SIZE.

Referenced by rimebuf_copyfrom(), and rimebuf_reference().

6.60.2.2 void rimebuf_compact (void)

Compact the rimebuf.

This function compacts the rimebuf by copying the data portion of the rimebuf so that becomes consecutive to the header. It also copies external data that has previously been referenced with rimebuf_reference() into the rimebuf.

This function is called by the Rime code before a packet is to be sent by a device driver. This assures that the entire packet is consecutive in memory.

Definition at line 90 of file rimebuf.c.

References rimebuf_datalen(), RIMEBUF_HDR_SIZE, rimebuf_is_reference(), and rimebuf_reference_-ptr().

6.60.2.3 int rimebuf_copyfrom (u8_t * *from*, u16_t *len*)

Copy from external data into the rimebuf.

Parameters:

from A pointer to the data from which to copy

len The size of the data to copy

Return values:

The number of bytes that was copied into the rimebuf

This function copies data from a pointer into the rimebuf. If the data that is to be copied is larger than the rimebuf, only the data that fits in the rimebuf is copied. The number of bytes that could be copied into the rimbuf is returned.

Examples:

test-abc.c, test-meshroute.c, and test-trickle.c.

Definition at line 78 of file rimebuf.c.

References rimebuf_clear(), and RIMEBUF_SIZE.

6.60.2.4 int rimebuf_copyto (u8_t * *to*)

Copy the entire rimebuf to an external buffer.

Parameters:

to A pointer to the buffer to which the data is to be copied

Return values:

The number of bytes that was copied to the external buffer

This function copies the rimebuf to an external buffer. Both the data portion and the header portion of the rimebuf is copied. If the rimebuf referenced external data (referenced with rimebuf_reference()) the external data is copied.

The external buffer to which the rimebuf is to be copied must be able to accomodate at least (RIMEBUF_-SIZE + RIMEBUF_HDR_SIZE) bytes. The number of bytes that was copied to the external buffer is returned.

Definition at line 120 of file rimebuf.c.

References RIMEBUF_HDR_SIZE.

6.60.2.5 int rimebuf_copyto_hdr (u8_t * *to*)

Copy the header portion of the rimebuf to an external buffer.

Parameters:

to A pointer to the buffer to which the data is to be copied

Return values:

The number of bytes that was copied to the external buffer

This function copies the header portion of the rimebuf to an external buffer.

The external buffer to which the rimebuf is to be copied must be able to accomodate at least RIMEBUF_-HDR_SIZE bytes. The number of bytes that was copied to the external buffer is returned.

Definition at line 103 of file rimebuf.c.

References RIMEBUF_HDR_SIZE.

6.60.2.6 u16_t rimebuf_datalen (void)

Get the length of the data in the rimebuf.

Returns:

Length of the data in the rimebuf

For outbound packets, the rimebuf consists of two parts: header and data. This function is used to get the length of the data in the rimebuf. The data is stored in the rimebuf and accessed via the rimebuf_dataptr() function.

For incoming packets, both the packet header and the packet data is stored in the data portion of the rimebuf. This function is then used to get the total length of the packet - both header and data.

Examples:

test-meshroute.c.

Definition at line 210 of file rimebuf.c.

Referenced by rimebuf_compact(), and rimebuf_totlen().

6.60.2.7 void * rimebuf_dataptr (void)

Get a pointer to the data in the rimebuf.

Returns:

Pointer to the rimebuf data

This function is used to get a pointer to the data in the rimebuf. The data is either stored in the rimebuf, or referenced to an external location.

For outbound packets, the rimebuf consists of two parts: header and data. The header is accessed with the rimebuf_hdrptr() function.

For incoming packets, both the packet header and the packet data is stored in the data portion of the rimebuf. Thus this function is used to get a pointer to the header for incoming packets.

Examples:

test-abc.c, test-meshroute.c, and test-treeroute.c.

Definition at line 178 of file rimebuf.c.

References RIMEBUF_HDR_SIZE.

Referenced by abc_input_packet().

6.60.2.8 int rimebuf_hdralloc (int *size*)

Extend the header of the rimebuf, for outbound packets.

Parameters:

size The number of bytes the header should be extended

Return values:

Non-zero if the header could be extended, zero otherwise

This function is used to allocate extra space in the header portion in the rimebuf, when preparing outbound packets for transmission. If the function is unable to allocate sufficient header space, the function returns zero and does not allocate anything.

Definition at line 148 of file rimebuf.c.

Referenced by abc_send(), and ibc_send().

6.60.2.9 u8_t rimebuf_hdrlen (void)

Get the length of the header in the rimebuf, for outbound packets.

Returns:

Length of the header in the rimebuf

For outbound packets, the rimebuf consists of two parts: header and data. This function is used to get the length of the header in the rimebuf. The header is stored in the rimebuf and accessed via the rimebuf_-hdrptr() function.

Definition at line 216 of file rimebuf.c.

References RIMEBUF_HDR_SIZE.

Referenced by rimebuf_totlen().

6.60.2.10 void ∗ rimebuf_hdrptr (void)

Get a pointer to the header in the rimebuf, for outbound packets.

Returns:

Pointer to the rimebuf header

For outbound packets, the rimebuf consists of two parts: header and data. This function is used to get a pointer to the header in the rimebuf. The header is stored in the rimebuf.

Definition at line 184 of file rimebuf.c.

Referenced by abc_send(), and ibc_send().

6.60.2.11 int rimebuf_hdrreduce (int *size*)

Reduce the header in the rimebuf, for incoming packets.

Parameters:

size The number of bytes the header should be reduced

Return values:

Non-zero if the header could be reduced, zero otherwise

This function is used to remove the first part of the header in the rimebuf, when processing incoming packets. If the function is unable to remove the requested amount of header space, the function returns zero and does not allocate anything.

Definition at line 159 of file rimebuf.c.

Referenced by abc_input_packet().

6.60.2.12 int rimebuf_is_reference (void)

Check if the rimebuf references external data.

Return values:

Non-zero if the rimebuf references external data, zero otherwise.

For outbound packets, the rimebuf consists of two parts: header and data. This function is used to check if the rimebuf points to external data that has previously been referenced with rimebuf_reference().

Definition at line 198 of file rimebuf.c.

References RIMEBUF_HDR_SIZE.

Referenced by rimebuf_compact().

6.60.2.13 void rimebuf_reference (void * *ptr*, u16_t *len*)

Point the rimebuf to external data.

Parameters:

ptr A pointer to the external data

len The length of the external data

For outbound packets, the rimebuf consists of two parts: header and data. This function is used to make the rimebuf point to external data. The function also specifies the length of the external data that the rimebuf references.

Definition at line 190 of file rimebuf.c.

References rimebuf_clear().

6.60.2.14 void * rimebuf_reference_ptr (void)

Get a pointer to external data referenced by the rimebuf.

Return values:

A pointer to the external data

For outbound packets, the rimebuf consists of two parts: header and data. The data may point to external data that has previously been referenced with rimebuf_reference(). This function is used to get a pointer to the external data.

Definition at line 204 of file rimebuf.c.

Referenced by rimebuf_compact().

6.60.2.15 void rimebuf_set_datalen (u16_t *len*)

Set the length of the data in the rimebuf.

Parameters:

len The length of the data

For outbound packets, the rimebuf consists of two parts: header and data. This function is used to set the length of the data in the rimebuf.

Examples:

test-treeroute.c.

Definition at line 171 of file rimebuf.c.

6.60.2.16 u16_t rimebuf_totlen (void)

Get the total length of the header and data in the rimebuf.

Returns:

Length of data and header in the rimebuf

Definition at line 222 of file rimebuf.c.

References rimebuf_datalen(), and rimebuf_hdrlen().

6.61 Rime route discovery protocol

6.61.1 Detailed Description

The route-discovery module does route discovery for Rime.

6.61.2 Channels

The ibc module uses 2 channels; one for the flooded route request packets and one for the unicast route replies.

Files

- file route-discovery.h

  Header file for the Rime mesh routing protocol.

- file route-discovery.c

  Route discovery protocol.

6.62 Rime route table

6.62.1 Detailed Description

The route module handles the route table in Rime.

Files

- file route.h

  Header file for the Rime route table.

- file route.c

  Rime route table.

6.63 Stubborn anonymous best-effort local area broadcast

6.63.1 Detailed Description

The sabc module provides stubborn anonymous best-effort local area broadcast. A message sent with the sabc module is repeated until either the mssage is canceled or a new message is sent. Messages sent with the sabc module are not identified with a sender ID.

6.63.2 Channels

The sabc module uses 1 channel.

Files

- file sabc.h

  Header file for the Rime module Stubborn Anonymous BroadCast (sabc).

- file sabc.c

  Implementation of the Rime module Stubborn Anonymous BroadCast (sabc).

Data Structures

- struct sabc_conn

  A sabc connection.

Functions

- void sabc_open (struct sabc_conn ∗c, u16_t channel, const struct sabc_callbacks ∗u)

  Set up a sabc connection.

- int sabc_send_stubborn (struct sabc_conn ∗c, clock_time_t t)

  Send a stubborn message.

- void sabc_cancel (struct sabc_conn ∗c)

  Cancel the current stubborn message.

- void sabc_set_timer (struct sabc_conn ∗c, clock_time_t t)

  Set the retransmission time of the current stubborn message.

6.63.3 Function Documentation

6.63.3.1 void sabc_cancel (struct sabc_conn * *c*)

Cancel the current stubborn message.

Parameters:

c A sabc connection that must have been previously set up with sabc_open()

This function cancels a stubborn message that has previously been sent with the sabc_send_stubborn() function.

Definition at line 116 of file sabc.c.

6.63.3.2 void sabc_open (struct sabc_conn * *c*, u16_t *channel*, const struct sabc_callbacks * *u*)

Set up a sabc connection.

Parameters:

c A pointer to a user-supplied struct sabc variable.

channel The Rime channel on which messages should be sent.

u Pointer to the upper layer functions that should be used for this connection.

This function sets up a sabc connection on the specified channel. No checks are made if the channel is currently used by another connection.

This function must be called before any other function that operates on the connection is called.

Definition at line 65 of file sabc.c.

References abc_open().

6.63.3.3 int sabc_send_stubborn (struct sabc_conn * *c*, clock_time_t *t*)

Send a stubborn message.

Parameters:

c A sabc connection that must have been previously set up with sabc_open()

t The time between message retransmissions.

This function sends a message from the Rime buffer. The message must have been previously constructed in the Rime buffer. When this function returns, the message has been copied into a queue buffer.

If another message has previously been sent, the old message is canceled.

Definition at line 100 of file sabc.c.

References sabc_conn::buf, and sabc_set_timer().

6.63.3.4 void sabc_set_timer (struct sabc_conn * *c*, clock_time_t *t*)

Set the retransmission time of the current stubborn message.

Parameters:

c A sabc connection that must have been previously set up with sabc_open()

t The new time between message retransmissions.

This function sets the retransmission timer for the current stubborn message to a new value.

Definition at line 94 of file sabc.c.

Referenced by sabc_send_stubborn().

6.64 Stubborn identified broadcast

6.64.1 Detailed Description

The sibc module provides stubborn identified best-effort local area broadcast. A message sent with the sibc module is repeated until either the mssage is canceled or a new message is sent. Messages sent with the sibc module are identified with a sender ID.

6.64.2 Channels

The sibc module uses 1 channel.

Files

- file sibc.h

  Header file for the Rime module Stubborn Identified BroadCast (sibc).

- file sibc.c

  Implementation of the Rime module Stubborn Identified BroadCast (sibc).

6.65 Stubborn unicast

6.65.1 Detailed Description

The suc module takes one packet and sends it repetedly.

6.65.2 Channels

The suc module uses 1 channel.

Files

- file suc.h

  Stubborn unicast header file.

- file suc.c

  Stubborn unicast.

6.66 Tree-based hop-by-hop reliable data collection

6.66.1 Detailed Description

The tree module implements a hop-by-hop reliable data collection mechanism.

6.66.2 Channels

The tree module uses 2 channels; one for neighbor discovery and one for data packets.

Files

- file tree.h

  Header file for hop-by-hop reliable data collection.

- file tree.c

  Tree-based hop-by-hop reliable data collection.

6.67 Reliable single-source multi-hop flooding

6.67.1 Detailed Description

The trickle module sends a single packet to all nodes on the network.

6.67.2 Channels

The trickle module uses 1 channel.

Files

- file trickle.h

  Header file for Trickle (reliable single source flooding) for Rime.

- file trickle.c

  Trickle (reliable single source flooding) for Rime.

6.68 Unique anonymous best effort local area broadcast

6.68.1 Detailed Description

The uabc module sends one anonymous packet that is unique within a time interval.

6.68.2 Channels

The uabc module uses 1 channel.

Files

- file uabc.h

  Header file for Unique Anonymous best effort local area BroadCast (uabc).

- file uabc.c

  Unique Anonymous best effort local area BroadCast (uabc).

6.69 Single-hop unicast

6.69.1 Detailed Description

The uc module sends a packet to a single receiver.

6.69.2 Channels

The uc module uses 1 channel.

Files

- file uc.h

  Header file for Rime's single-hop unicast.

- file uc.c

  Single-hop unicast.

6.70 Unique identified best effort local area broadcast

6.70.1 Detailed Description

The uibc module sends one packet that is unique within a time interval.

6.70.2 Channels

The uibc module uses 1 channel.

Files

- file uibc.h

  Header file for Unique Identified best effort local area BroadCast (uibc).

- file uibc.c

  Unique Identified best effort local area BroadCast (uibc).

6.71 Single-hop reliable bulk data transfer

6.71.1 Detailed Description

The rudolph0 module implements a single-hop reliable bulk data transfer mechanism.

6.71.2 Channels

The rudolph0 module uses 2 channels; one for data packets and one for NACK and repair packets.

Files

- file rudolph0.h

  Header file for the single-hop reliable bulk data transfer module.

- file rudolph0.c

  Rudolph0: a simple block data flooding protocol.

6.72 Multi-hop reliable bulk data transfer

6.72.1 Detailed Description

The rudolph1 module implements a multi-hop reliable bulk data transfer mechanism.

6.72.2 Channels

The rudolph1 module uses 2 channels; one for data transmissions and one for NACKs and repair packets.

Files

- file rudolph1.h

  Header file for the multi-hop reliable bulk data transfer mechanism.

- file rudolph1.c

  Rudolph1: a simple block data flooding protocol.

6.73 Memory block management functions

6.73.1 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.

Files

- file memb.h

  Memory block allocation routines.

- file memb.c

  Memory block allocation routines.

Defines

- #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().

6.73.2 Define Documentation

6.73.2.1 #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, struct connection, 16);
```

Parameters:

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

structure The name of the struct that the memory block holds

num The total number of memory chunks in the block.

Definition at line 96 of file memb.h.

6.73.3 Function Documentation

6.73.3.1 void * memb_alloc (struct memb_blocks * *m*)

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

Parameters:

m A memory block previosly declared with MEMB().

Definition at line 60 of file memb.c.

6.73.3.2 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 previosly declared with MEMB().

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

Returns:

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

Definition at line 80 of file memb.c.

6.73.3.3 void memb_init (struct memb_blocks * *m*)

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

Parameters:

m A memory block previosly declared with MEMB().

Definition at line 53 of file memb.c.

6.74 Managed memory allocator

6.74.1 Detailed Description

The managed memory allocator is a fragmentation-free memory manager.

It keeps the allocated memory free from fragmentation by compacting the memory when blocks are freed. A program that uses the managed memory module cannot be sure that allocated memory stays in place. Therefore, a level of indirection is used: access to allocated memory must always be done using a special macro.

Note:

This module has not been heavily tested.

Files

- file mmem.h

  Header file for the managed memory allocator.

- file mmem.c

  Implementation of the managed memory allocator.

Defines

- #define MMEM_PTR(m)

  Get a pointer to the managed memory.

Functions

- int mmem_alloc (struct mmem ∗m, unsigned int size)

  Allocate a managed memory block.

- void mmem_free (struct mmem ∗)

  Deallocate a managed memory block.

- void mmem_init (void)

  Initialize the managed memory module.

6.74.2 Define Documentation

6.74.2.1 #define MMEM_PTR(m)

Get a pointer to the managed memory.

Parameters:

m A pointer to the struct mmem

Returns:

A pointer to the memory block, or NULL if memory could not be allcated.

Author:

Adam Dunkels

This macro is used to get a pointer to a memory block allocated with mmem_alloc().

Definition at line 76 of file mmem.h.

6.74.3 Function Documentation

6.74.3.1 int mmem_alloc (struct mmem ∗ *m*, unsigned int *size*)

Allocate a managed memory block.

Parameters:

m A pointer to a struct mmem.

size The size of the requested memory block

Returns:

Non-zero if the memory could be allocated, zero if memory was not available.

Author:

Adam Dunkels

This function allocates a chunk of managed memory. The memory allocated with this function must be deallocated using the mmem_free() function.

Note:

This function does NOT return a pointer to the allocated memory, but a pointer to a structure that contains information about the managed memory. The macro MMEM_PTR() is used to get a pointer to the allocated memory.

Definition at line 84 of file mmem.c.

References list_add().

6.74.3.2 void mmem_free (struct mmem * *m*)

Deallocate a managed memory block.

Parameters:

m A pointer to the managed memory block

Author:

Adam Dunkels

This function deallocates a managed memory block that previously has been allocated with mmem_alloc().

Definition at line 120 of file mmem.c.

References list_remove().

6.74.3.3 void mmem_init (void)

Initialize the managed memory module.

Author:

Adam Dunkels

This function initializes the managed memory module and should be called before any other function from the module.

Definition at line 153 of file mmem.c.

References list_init().

6.75 Linked list library

6.75.1 Detailed Description

The linked list library provides a set of functions for manipulating linked lists.

A linked list is made up of elements where the first element **must** be a pointer. This pointer is used by the linked list library to form lists of the elements.

Lists are declared with the LIST() macro. The declaration specifies the name of the list that later is used with all list functions.

Lists can be manipulated by inserting or removing elements from either sides of the list (list_push(), list_add(), list_pop(), list_chop()). A specified element can also be removed from inside a list with list_remove(). The head and tail of a list can be extracted using list_head() and list_tail(), respecitively.

Files

- file list.h

  Linked list manipulation routines.

- file list.c

  Linked list library implementation.

Defines

- #define LIST(name)

  Declare a linked list.

Typedefs

- typedef void ** list_t

  The linked list type.

Functions

- void list_init (list_t list)

  Initialize a list.

- void * list_head (list_t list)

  Get a pointer to the first element of a list.

- void * list_tail (list_t list)

  Get the tail of a list.

- void * list_pop (list_t list)

  Remove the first object on a list.

- void list_push (list_t list, void ∗item)

  Add an item to the start of the list.

- void ∗ list_chop (list_t list)

  Remove the last object on the list.

- void list_add (list_t list, void ∗item)

  Add an item at the end of a list.

- void list_remove (list_t list, void ∗item)

  Remove a specific element from a list.

- int list_length (list_t list)

  Get the length of a list.

- void list_copy (list_t dest, list_t src)

  Duplicate a list.

- void list_insert (list_t list, void ∗previtem, void ∗newitem)

  Insert an item after a specified item on the list.

6.75.2 Define Documentation

6.75.2.1 #define LIST(name)

Value:

```
static void *LIST_CONCAT(name,_list) = NULL; \
         static list_t name = (list_t)&LIST_CONCAT(name,_list)
```

Declare a linked list.

This macro declares a linked list with the specified `type`. The type **must** be a structure (`struct`) with its first element being a pointer. This pointer is used by the linked list library to form the linked lists.

Parameters:

name The name of the list.

Examples:

example-list.c.

Definition at line 85 of file list.h.

6.75.3 Function Documentation

6.75.3.1 void list_add (list_t *list*, void ∗ *item*)

Add an item at the end of a list.

This function adds an item to the end of the list.

Parameters:
list The list.

item A pointer to the item to be added.

See also:
list_push()

Examples:
example-list.c.

Definition at line 143 of file list.c.

References list_tail().

Referenced by abc_open(), and mmem_alloc().

6.75.3.2 void ∗ list_chop (list_t *list*)

Remove the last object on the list.

This function removes the last object on the list and returns it.

Parameters:
list The list

Returns:
The removed object

Definition at line 180 of file list.c.

6.75.3.3 void list_copy (list_t *dest*, list_t *src*)

Duplicate a list.

This function duplicates a list by copying the list reference, but not the elements.

Note:
This function does **not** copy the elements of the list, but merely duplicates the pointer to the first element of the list.

Parameters:
dest The destination list.

src The source list.

Definition at line 101 of file list.c.

6.75.3.4 void ∗ list_head (list_t *list*)

Get a pointer to the first element of a list.

This function returns a pointer to the first element of the list. The element will **not** be removed from the list.

Parameters:
list The list.

Returns:
A pointer to the first element on the list.

See also:
list_tail()

Examples:
example-list.c.

Definition at line 83 of file list.c.

Referenced by abc_input_packet().

6.75.3.5 void list_init (list_t *list*)

Initialize a list.

This function initalizes a list. The list will be empty after this function has been called.

Parameters:
list The list to be initialized.

Examples:
example-list.c.

Definition at line 66 of file list.c.

Referenced by mmem_init().

6.75.3.6 void list_insert (list_t *list*, void ∗ *previtem*, void ∗ *newitem*)

Insert an item after a specified item on the list.

Parameters:
list The list
previtem The item after which the new item should be inserted
newitem The new item that is to be inserted

Author:
Adam Dunkels

This function inserts an item right after a specified item on the list. This function is useful when using the list module to ordered lists.

If previtem is NULL, the new item is placed at the start of the list.

Definition at line 295 of file list.c.

References list_push().

6.75.3.7 int list_length (list_t *list*)

Get the length of a list.

This function counts the number of elements on a specified list.

Parameters:

list The list.

Returns:

The length of the list.

Definition at line 267 of file list.c.

6.75.3.8 void * list_pop (list_t *list*)

Remove the first object on a list.

This function removes the first object on the list and returns a pointer to the list.

Parameters:

list The list.

Returns:

The new head of the list.

Definition at line 212 of file list.c.

6.75.3.9 void list_remove (list_t *list*, void * *item*)

Remove a specific element from a list.

This function removes a specified element from the list.

Parameters:

list The list.

item The item that is to be removed from the list.

Definition at line 232 of file list.c.

Referenced by abc_close(), and mmem_free().

6.75.3.10 void * list_tail (list_t *list*)

Get the tail of a list.

This function returns a pointer to the elements following the first element of a list. No elements are removed by this function.

Parameters:

list The list

Returns:

A pointer to the element after the first element on the list.

See also:

list_head()

Definition at line 118 of file list.c.

Referenced by list_add().

6.76 Table-driven Manchester encoding and decoding

6.76.1 Detailed Description

Manchester encoding is a bit encoding scheme which translates each bit into two bits: the original bit and the inverted bit.

Manchester encoding is used for transmitting ones and zeroes between two computers. The Manchester encoding reduces the receive oscillator drift by making sure that no consecutive ones or zeroes are ever transmitted.

The table driven method of Manchester encoding and decoding uses two tables with 256 entries. One table is a direct mapping of an 8-bit byte into a 16-bit Manchester encoding of the byte. The second table is a mapping of a Manchester encoded 8-bit byte to 4 decoded bits.

Files

- file me.h

  Header file for the table-driven Manchester encoding and decoding.

- file me.c

  Implementation of the table-driven Manchester encoding and decoding.

Functions

- unsigned char me_valid (unsigned char m)

  Check if an encoded byte is valid.

- unsigned short me_encode (unsigned char c)

  Manchester encode an 8-bit byte.

- unsigned char me_decode16 (unsigned short m)

  Decode a Manchester encoded 16-bit word.

- unsigned char me_decode8 (unsigned char m)

  Decode a Manchester encoded 8-bit byte.

6.76.2 Function Documentation

6.76.2.1 unsigned char me_decode16 (unsigned short *m*)

Decode a Manchester encoded 16-bit word.

This function decodes a Manchester encoded 16-bit word into a 8-bit byte. The function does not check for parity errors in the encoded byte.

Parameters:

m The 16-bit Manchester encoded word

Returns:

The decoded 8-bit byte

Definition at line 76 of file me.c.

6.76.2.2 unsigned char me_decode8 (unsigned char *m*)

Decode a Manchester encoded 8-bit byte.

This function decodes a Manchester encoded 8-bit byte into 4 decoded bits.. The function does not check for parity errors in the encoded byte.

Parameters:

m The 8-bit Manchester encoded byte

Returns:

The decoded 4 bits

Definition at line 100 of file me.c.

6.76.2.3 unsigned short me_encode (unsigned char *c*)

Manchester encode an 8-bit byte.

This function Manchester encodes an 8-bit byte into a 16-bit word. The function me_decode() does the inverse operation.

Parameters:

c The byte to be encoded

Return values:

The encoded word.

Definition at line 59 of file me.c.

6.77 Cyclic Redundancy Check 16 (CRC16) calculcation

6.77.1 Detailed Description

The Cyclic Redundancy Check 16 is a hash function that produces a checksum that is used to detect errors in transmissions.

The CRC16 calculation module is an iterative CRC calculator that can be used to cummulatively update a CRC checksum for every incoming byte.

Files

- file crc16.h

  Header file for the CRC16 calculcation.

- file crc16.c

  Implementation of the CRC16 calculcation.

Functions

- unsigned short crc16_add (unsigned char b, unsigned short crc)
  Update an accumulated CRC16 checksum with one byte.

6.77.2 Function Documentation

6.77.2.1 unsigned short crc16_add (unsigned char *b*, unsigned short *crc*)

Update an accumulated CRC16 checksum with one byte.

Parameters:
- ***b*** The byte to be added to the checksum
- ***crc*** The accumulated CRC that is to be updated.

Returns:
The updated CRC checksum.

This function updates an accumulated CRC16 checksum with one byte. It can be used as a running checksum, or to checksum an entire data block.

Note:
The algorithm used in this implementation is tailored for a running checksum and does not perform as well as a table-driven algorithm when checksumming an entire data block.

Definition at line 48 of file crc16.c.

6.78 The Tmote Sky Board

It is an MSP430-based board with an 802.15.4-compatible CC2420 radio chip, a 1 megabyte external serial flash memory, and two light sensors. Contiki was ported to the Tmote Sky by Björn Grönvall as part of the RUNES project. The Tmote Sky port was integrated into the Contiki build system in March 2007.

The platform-specif source code for the Tmote Sky port can be found in the directories platform/sky and cpu/msp430 in the Contiki source tree. Code for writing to the on-chip flash ROM is in the cpu/msp430/flash.c and code for reading and writing to the external flash is the file platform/sky/dev/xmem.c. Code for reading the light sensors is in platform/sky/dev/light.c.

The serial/USB port is read from and written to with either the code in cpu/msp430/dev/uart1.c or platform/sky/slip_uart1.c, depending on weather or not the Tmote Sky is running TCP/IP or not.

There are currently two CC2420 drivers in the Contiki source code, core/dev/simple-cc2420.c (a really simple CC2420 driver) and core/dev/cc2420.c (a more feature-rich CC2420 driver).

More information about the Tmote Sky, including data sheets, can be found at Moteiv's web site: `http://www.moteiv.com`

6.79 The ESB Embedded Sensor Board

6.79.1 Detailed Description

The ESB (Embedded Sensor Board) is a prototype wireless sensor network device developed at Freie Universität Berlin.

The ESB consists of a Texas Instruments MSP430 low-power microcontroller with 2k RAM and 60k flash ROM, a TR1001 radio transceiver, a 32k serial EEPROM, an RS232 port, a JTAG port, a beeper, and a number of sensors (passive IR, active IR sender/receiver, vibration/tilt, microphone, temperature).

The Contiki/ESB port contains drivers for most of the sensors. The drivers were mostly adapted from sources from FU Berlin.

6.79.2 Getting started with Contiki for the ESB platform

The ESB is equipped with an MSP430 microcontroller. The first step to getting started with Contiki for the ESB is to install the development tools for compiling Contiki for the MSP430.

Windows users, see Setting up the Windows environment. FreeBSD users, see Setting up the FreeBSD environment

6.79.3 Setting up the Windows environment

The Contiki development environment under Windows uses the Cygwin environment. Cygwin is a Linux-like environment for Windows. Cygwin can be found at `http://www.cygwin.com.` Click on the icon "Install Cygwin Now" to the right to get the installation started.

Choose "Install from Internet" and then specify where you want to install cygwin (recommended installation path: `C:\cygwin`). Continue with the installation until you are asked to select packages. Most packages can be left as "Default" but there is one package that are not installed by default. Install the following package by clicking at "Default" until it changes to "Install":

- Devel - contains things for developers (make, etc).

When cygwin is installed there should be a cygwin icon that starts up a cygwin bash when clicked on. Whenever it is time to compile and send programs to the ESB nodes it will be done from a cygwin shell.

6.79.3.1 C programming editor If you do not already have a nice programming editor it is a good idea to download and install one. The Crimson editor is a nice windows based editor that is both easy to get started with and fairly powerful.

Crimson Editor can be found at: `http://www.crimsoneditor.com/`

The editor is useful both when editing C programs and when modifying scripts and configuration files.

6.79.3.2 MSP430 Compiler and tools A compiler is needed to compile the programs to the MSP430 microprocessor that is used on the ESB sensor nodes. Download and install the GCC toolchain for MSP430 (recommended installation path: C:\MSP430\).

The GCC toolchain for MSP430 can be found at: `http://sourceforge.net/projects/mspgcc/`

When the above software is installed you also need to set-up the PATH so that all of the necessary tools can be reached. In cygwin this is done by the following line (given that you have installed at recommended locations):

```
export PATH=$PATH:/cygdrive/c/MSP430/mspgcc/bin
```

This line can also be added to the .profile startup file in your cygwin home directory (`C:\cygwin\home\<YOUR USERNAME>\.profile`).

If your home directory is located elsewhere you can find it by starting cygwin and running `cd` followed by `pwd`.

6.79.3.3 The Contiki operating system, including examples

When programming the ESB sensor nodes it is very useful to have an operating system that takes care of some of the low-level tasks and also gives you as a programmer APIs for things like events, hardware and networking. We will use the Contiki operating system developed by Adam Dunkels, SICS, which is very well suited when programming small embedded systems.

The Contiki OS can be found at: `http://www.sics.se/~adam/contiki/`

Unzip the Contiki OS at (for example) C:\ and you will get the following directories among others:

- contiki-2.x/core - the contiki operating system
- contiki-2.x/platform/esb - the contiki operating system drivers, etc for the ESB
- contiki-2.x/platform/esb/apps/ - example applications for the ESB

6.79.3.4 Testing the tools

Now everything necessary to start developing Contiki-based sensor net applications should be installed. Start cygwin and change to the directory `contiki-2.x/platform/esb/`. Then call `make beeper.co`.

If you get an error about multiple cygwin dlls when compiling, you need to delete `cygwin1.dll` from the MSP430 GCC toolchain (`C:\MSP430\bin\cygwin1.dll`).

Connect a node and turn it on. Upload the test application by calling `make beeper.u`.

6.79.3.5 Development tools

- `make <SPEC>` will compile and make a executable file ready for sending to the ESB nodes. Depending on the `SPEC` it might even startup the application that sends the executable to the node. Typically you would write things like `"make beeper.u"` to get the file `beeper.c` compiled, linked and sent out to the ESB node

6.79.3.6 Some basic shell commands

- `cd <DIR>` change to a specified directory (same as in DOS)
- `pwd <DIR>` shows your current directory
- `ls` list the directory
- `mkdir <DIR>` creates a new directory
- `cp <SRC> <DEST>` copies a file

6.79.4 Setting up the FreeBSD environment

Download the msp430-gcc, msp430-binutils, and msp430-libc packages from `http://www.sics.se/~adam/contiki/freebsd-packages/`. Install the packages (as root) with `pkg_add`.

6.79.5 Compiling your first Contiki system

6.79.6 Burning node IDs to EEPROM

The Contiki ESB port comes with a small program, `burn-nodeid` that semi-permanently stores a (unique) node ID number in the ESB EEPROM. When the Contiki ESB port boots up, this node ID is restored from the EEPROM. To compile and run this program, go into your project directory and run

```
make burn-nodeid.u nodeid=X
```

Where `X` is the node ID that will be burned into EEPROM. The `burn-nodeid` program stores the node ID in EEPROM, reads it back, and writes the output

Modules

- Introduction to Over The Air Reprogramming under Windows
- Beeper interface
- ESB RS232
- TR1001 radio tranciever device driver

6.80 Introduction to Over The Air Reprogramming under Windows

Author:

Joakim Eriksson, Niclas Finne

6.80.1 Introduction

This is a brief introduction how to program ESB sensor nodes over radio under Windows. It is assumed that you already have the environment setup for programming ESB sensor nodes using JTAG cable.

6.80.2 Configuring SLIP under Windows XP

This section describes how to setup a SLIP connection under Windows. A SLIP connection forwards TCP/IP traffic to/from the sensor nodes and lets you communicate with them using standard network tools such as `ping`.

1. Click start button and choose 'My Computer'. Right-click 'My Network Places' and choose 'Properties'.
2. Click 'Create a new connection'.
3. Select 'Set up an advanced connection'.
4. Select 'Connect directly to another computer'.
5. Select 'Guest'.
6. Select a name for the slip connection (for example 'ESB').
7. Select the serial port to use when communicating with the sensor node.
8. Add the connection by clicking 'Finish'.
9. A connection window will open. Choose 'Properties'.

10. Click on 'Configure...' and deselect all selected buttons. Choose the speed 57600 bps.
11. Close the modem configuration window, and go to the 'Options' tab in the ESB properties. Deselect all except 'Display progress...'.
12. Go to the 'Networking' tab. Change to 'SLIP: Unix Connection' and deselect all except the first two items in the connection item list.
13. Select 'Internet Protocol (TCP/IP)' and click 'Properties'. Enter the IP address '172.16.0.1'.
14. Click 'Advanced' and deselect all checkboxes in the 'Advanced TCP/IP Settings'. Go to the 'WINS' tab and deselect 'Enable LMHOSTS lookup' if it is selected. Also select 'Disable NetBIOS over TCP/IP'.

6.80.3 Setup ESB for over the air programming

1. Make sure you have the latest version of contiki (older versions of contiki might not work with SLIP under Windows)
2. Install the contiki kernel by running

   ```
   make core.u
   ```

3. Attach the ESB node to the serial port and make sure it is turned on. Select your ESB SLIP connection in your 'Network Connections' and choose 'Connect' (or double click on it). If everything works Windows should say that you have a new connection.
4. Set the IP address for the node by pinging it (it will claim the IP address of the first ping it hears). Note that the slip interface has IP address 172.16.0.1 but the node will have the IP address 172.16.1.1.

   ```
   ping 172.16.1.1
   ```

If everything works the node should click and reply to the pings.

6.80.4 Send programs over the air

Contiki applications to be installed via radio are compiled somewhat different compared to normal applications.

Each node needs an IP address for OTA to work. A node id can be specified when you upload the contiki kernel to a node and this is used to construct an IP address for the node. If you specify 2 as node id, the node will have the IP address 172.16.1.2. Each node should have its own unique node id.

You need to compile a core and upload it onto the nodes. All nodes must run the same core. Move to the directory '`contiki-2.x/platform/esb`' and run

```
make
make core.u nodeid=X
```

to upload the core to your nodes. Use the number 1, 2, 3, etc, as the node id (`X`) for the nodes. This will give the nodes the IP addresses 172.16.1.1, 172.16.1.2, etc.

Then you need a program to send the application to connected nodes. Compile it by running

```
make send
```

Make sure you have a node with IP address 172.16.1.1 connected to your serial port and have SLIP activated. Then compile and send a testprogram by running

```
make beeper.ce
./send 172.16.1.1 beeper.ce
```

6.81 Beeper interface

Files

- file beep.h

  Interface to the beeper.

Functions

- void beep_beep (int len)

  Beep for a specified time.

- void beep_alarm (int alarmmode, int len)

  Beep an alarm for a specified time.

- void beep (void)

  Produces a quick click-like beep.

- void beep_down (int len)

  A beep with a pitch-bend down.

- void beep_on (void)

  Turn the beeper on.

- void beep_off (void)

  Turn the beeper off.

- void beep_spinup (void)

  Produce a sound similar to a hard-drive spinup.

- void beep_long (clock_time_t len)

  Beep for a long time (seconds).

6.81.1 Function Documentation

6.81.1.1 void beep (void)

Produces a quick click-like beep.

This function produces a short beep that sounds like a click.

6.81.1.2 void beep_alarm (int *alarmmode*, int *len*)

Beep an alarm for a specified time.

This function causes the beeper to beep for the specified time. The time is measured in the same units as for the clock_delay() function.

Note:

This function will hang the CPU during the beep.
This function will stop any beep that was on previously when this function ends.
If the beeper is turned off with beep_off() this call will still take the same time, though it will be silent.

Parameters:

alarmmode The alarm mode (BEEP_ALARM1,BEEP_ALARM2)

len The length of the beep.

6.81.1.3 void beep_beep (int *len*)

Beep for a specified time.

This function causes the beeper to beep for the specified time. The time is measured in the same units as for the clock_delay() function.

Note:

This function will hang the CPU during the beep.
This function will stop any beep that was on previously when this function ends.
If the beeper is turned off with beep_off() this call will still take the same time, though it will be silent.

Parameters:

len The length of the beep.

6.81.1.4 void beep_down (int *len*)

A beep with a pitch-bend down.

This function produces a pitch-bend sound with deecreasing frequency.

Parameters:

len The length of the pitch-bend.

6.81.1.5 void beep_long (clock_time_t *len*)

Beep for a long time (seconds).

This function produces a beep with the specified length and will not return until the beep is complete. The length of the beep is specified using CLOCK_SECOND: a two second beep is CLOCK_SECOND $*$ 2, and a quarter second beep is CLOCK_SECOND / 4.

Note:

If the beeper is turned off with beep_off() this call will still take the same time, though it will be silent.

Parameters:

len The length of the beep, measured in units of CLOCK_SECOND

6.81.1.6 void beep_off (void)

Turn the beeper off.

This function turns the beeper off after it has been turned on with beep_on().

6.81.1.7 void beep_on (void)

Turn the beeper on.

This function turns on the beeper. The beeper is turned off with the beep_off() function.

6.81.1.8 void beep_spinup (void)

Produce a sound similar to a hard-drive spinup.

This function produces a sound that is intended to be similar to the sound a hard-drive makes when it starts.

6.82 ESB RS232

Files

- file rs232.h

  Header file for MSP430 RS232 driver.

- file rs232.c

  RS232 communication device driver for the MSP430.

Functions

- void rs232_init (void)

  Initialize the RS232 module.

- void rs232_set_input (int(∗f)(unsigned char))

  Set an input handler for incoming RS232 data.

- void rs232_set_speed (unsigned char speed)

  Configure the speed of the RS232 hardware.

- void rs232_print (char ∗str)

  Print a text string on RS232.

- void rs232_send (char c)

  Print a character on RS232.

6.82.1 Function Documentation

6.82.1.1 void rs232_init (void)

Initialize the RS232 module.

This function is called from the boot up code to initalize the RS232 module.

Definition at line 78 of file rs232.c.

References rs232_set_speed().

6.82.1.2 void rs232_print (char ∗ *str*)

Print a text string on RS232.

Parameters:

str A pointer to the string that is to be printed

This function prints a string to RS232. The string must be terminated by a null byte. The RS232 module must be correctly initalized and configured for this function to work.

Definition at line 135 of file rs232.c.

References rs232_send().

6.82.1.3 void rs232_send (char *c*)

Print a character on RS232.

Parameters:

c The character to be printed

This function prints a character to RS232. The RS232 module must be correctly initalized and configured for this function to work.

Definition at line 94 of file rs232.c.

Referenced by rs232_print().

6.82.1.4 void rs232_set_input (int(∗)(unsigned char) *f*)

Set an input handler for incoming RS232 data.

Parameters:

f A pointer to a byte input handler

This function sets the input handler for incoming RS232 data. The input handler function is called for every incoming data byte. The function is called from the RS232 interrupt handler, so care must be taken when implementing the input handler to avoid race conditions.

The return value of the input handler affects the sleep mode of the CPU: if the input handler returns non-zero (true), the CPU is awakened to let other processing take place. If the input handler returns zero, the CPU is kept sleeping.

Definition at line 144 of file rs232.c.

6.82.1.5 void rs232_set_speed (unsigned char *speed*)

Configure the speed of the RS232 hardware.

Parameters:

speed The speed

This function configures the speed of the RS232 hardware. The allowed parameters are RS232_19200, RS232_38400, RS232_57600, and RS232_115200.

Definition at line 108 of file rs232.c.

Referenced by rs232_init().

6.83 TR1001 radio tranciever device driver

Files

- file tr1001.c

  Device driver and packet framing for the RFM-TR1001 radio module.

6.84 Microsoft Windows

Author:

Oliver Schmidt <ol.sc@web.de>

6.84.1 Getting started

The Microsoft Windows port of Contiki doesn't use the the Contiki build system. Instead it is built with the Microsoft Visual C++. If you already have Microsoft Visual Studio 2005 Standard Edition (or better) installed you're ready to go.

Otherwise you can download and install the free Visual C++ 2005 Express Edition. I'd recommend to download the full package and then do a local install. That so called Manual Installation is described on this page: `http://msdn.microsoft.com/vstudio/express/support/install/`

Then follow the instructions on this page carefully (Note - You only need to install the Microsoft Windows Core SDK from the Microsoft Platform SDK): `http://msdn.microsoft.com/vstudio/express/visualc/usingpsdk/default.aspx`

Finally you might want to integrate the Platform SDK help content into the IDE help system (although this isn't necessary for Contiki development) by choosing this topic in the IDE help system and following the instructions given: ms-help://MS.VSExpressCC.v80/dv_vsexpcc/local/CollectionManagerExpress.htm

For network I/O you need the WinPcap library which is available on this page: `http://www.winpcap.org/install/default.htm`

I'd recommend to install the Wireshark network protocol analyzer which uses (and therefore comes with) the WinPcap libary from this page instead of installing just the WinPcap library from the page above: `http://www.wireshark.org/download.html`

Doubleclicking contiki-2.x\platform\win32\contiki.sln should bring up the IDE and load the Contiki Solution. This takes some time if done for the first time as the source is scanned and quite some metadata gets generated.

Now press F7 for Build Solution and then F5 for Start Debugging. This should bring up Contiki inside a new command prompt window. Starting Contiki in the debugger has the benefit of having the debug output routed to the IDE output pane.

Depending on your settings for command prompt windows the mouse should work right away with Contiki.

Contiki resizes its desktop on resizing the command prompt window Contiki is running in.

Beside the Contiki Quit menu entry you can safely use Ctrl-C for a clean exit. You'll notice that when running Contiki from an already open command prompt window (that therefore doesn't close on Contiki exit): The caption, colors, cursor, ... are restored.

6.85 Uiparch

Variables

- u8_t uip_acc32 [4]

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

7 Contiki 2.x Directory Documentation

7.1 apps/ Directory Reference

Directories

- directory program-handler

7.2 core/cfs/ Directory Reference

Files

- file cfs.h

  CFS header file.

7.3 core/ Directory Reference

Directories

- directory cfs
- directory ctk
- directory dev
- directory lib
- directory loader
- directory net
- directory sys

7.4 core/ctk/ Directory Reference

Files

- file ctk-draw.h

  CTK screen drawing module interface, ctk-draw.

- file ctk.c

  The Contiki Toolkit CTK, the Contiki GUI.

- file ctk.h

  CTK header file.

7.5 platform/esb/dev/ Directory Reference

Files

- file beep.h

  Interface to the beeper.

- file eeprom.c

  EEPROM functions.

- file rs232.c

  RS232 communication device driver for the MSP430.

- file rs232.h

  Header file for MSP430 RS232 driver.

- file tr1001.c

  Device driver and packet framing for the RFM-TR1001 radio module.

7.6 core/dev/ Directory Reference

Files

- file eeprom.h

  EEPROM functions.

- file radio.h

  Header file for the radio API.

7.7 platform/esb/ Directory Reference

Directories

- directory dev

7.8 core/lib/ Directory Reference

Files

- file crc16.c

  Implementation of the CRC16 calculcation.

- file crc16.h

  Header file for the CRC16 calculcation.

- file ctk-textedit.c

  An experimental CTK text edit widget.

- file ctk-textedit.h

  Header file for the experimental application level CTK textedit widget.

- file list.c

  Linked list library implementation.

- file list.h

  Linked list manipulation routines.

- file me.c

  Implementation of the table-driven Manchester encoding and decoding.

- file me.h

  Header file for the table-driven Manchester encoding and decoding.

- file memb.c

  Memory block allocation routines.

- file memb.h

  Memory block allocation routines.

- file mmem.c

  Implementation of the managed memory allocator.

- file mmem.h

  Header file for the managed memory allocator.

- file petsciiconv.h

  PETSCII/ASCII conversion functions.

7.9 core/loader/ Directory Reference

Files

- file elfloader-arch.h

  Header file for the architecture specific parts of the Contiki ELF loader.

- file elfloader.h

  Header file for the Contiki ELF loader.

7.10 core/net/ Directory Reference

Directories

- directory rime

Files

- file **psock.c**
- file psock.h

  Protosocket library header file.

- file resolv.c

  DNS host name to IP address resolver.

- file resolv.h

  uIP DNS resolver code header file.

- file rime.h

  Header file for the Rime stack.

- file **tcpip.c**
- file tcpip.h

  Header for the Contiki/uIP interface.

- file uip-fw.c

  uIP packet forwarding.

- file uip-fw.h

  uIP packet forwarding header file.

- file **uip-split.c**
- file uip-split.h

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

- file uip.c

  The uIP TCP/IP stack code.

- file uip.h

  Header file for the uIP TCP/IP stack.

- file uip_arp.c

  Implementation of the ARP Address Resolution Protocol.

- file uip_arp.h

  Macros and definitions for the ARP module.

- file **uiplib.c**
- file uiplib.h

  Various uIP library functions.

- file uipopt.h

  Configuration options for uIP.

7.11 platform/ Directory Reference

Directories

- directory esb

7.12 apps/program-handler/ Directory Reference

Files

- file program-handler.c

  The program handler, used for loading programs and starting the screensaver.

7.13 core/net/rime/ Directory Reference

Files

- file abc.c

  Anonymous best-effort local area Broad Cast (abc).

- file abc.h

  Header file for the Rime module Anonymous BroadCast (abc).

- file ctimer.c

  Callback timer implementation.

- file ctimer.h

  Header file for the callback timer.

- file ibc.c

  Identified best-effort local area broadcast (ibc).

- file ibc.h

  Header file for identified best-effort local area broadcast.

- file mesh.c

  A mesh routing protocol.

- file mesh.h

  Header file for the Rime mesh routing protocol.

- file mh.c

  Multihop forwarding.

- file mh.h

  Multihop forwarding header file.

- file neighbor.c

  Radio neighborhood management.

- file neighbor.h

  Header file for the Contiki radio neighborhood management.

- file nf.c

  Best-effort network flooding (nf).

- file nf.h

  Header file for the best-effort network flooding (nf).

- file queuebuf.c

  Implementation of the Rime queue buffers.

- file queuebuf.h

  Header file for the Rime queue buffer management.

- file rimeaddr.c

  Functions for manipulating Rime addresses.

- file rimeaddr.h

  Header file for the Rime address repressentation.

- file rimebuf.c

  Rime buffer (rimebuf) management.

- file rimebuf.h

  Header file for the Rime buffer (rimebuf) management.

- file route-discovery.c

  Route discovery protocol.

- file route-discovery.h

  Header file for the Rime mesh routing protocol.

- file route.c

  Rime route table.

- file route.h

Header file for the Rime route table.

- file ruc.c

  Reliable unicast.

- file ruc.h

  Reliable unicast header file.

- file rudolph0.c

  Rudolph0: a simple block data flooding protocol.

- file rudolph0.h

  Header file for the single-hop reliable bulk data transfer module.

- file rudolph1.c

  Rudolph1: a simple block data flooding protocol.

- file rudolph1.h

  Header file for the multi-hop reliable bulk data transfer mechanism.

- file sabc.c

  Implementation of the Rime module Stubborn Anonymous BroadCast (sabc).

- file sabc.h

  Header file for the Rime module Stubborn Anonymous BroadCast (sabc).

- file sibc.c

  Implementation of the Rime module Stubborn Identified BroadCast (sibc).

- file sibc.h

  Header file for the Rime module Stubborn Identified BroadCast (sibc).

- file suc.c

  Stubborn unicast.

- file suc.h

  Stubborn unicast header file.

- file tree.c

  Tree-based hop-by-hop reliable data collection.

- file tree.h

  Header file for hop-by-hop reliable data collection.

- file trickle.c

  Trickle (reliable single source flooding) for Rime.

- file trickle.h

  Header file for Trickle (reliable single source flooding) for Rime.

- file uabc.c

  Unique Anonymous best effort local area BroadCast (uabc).

- file uabc.h

  Header file for Unique Anonymous best effort local area BroadCast (uabc).

- file uc.c

  Single-hop unicast.

- file uc.h

  Header file for Rime's single-hop unicast.

- file uibc.c

  Unique Identified best effort local area BroadCast (uibc).

- file uibc.h

  Header file for Unique Identified best effort local area BroadCast (uibc).

7.14 core/sys/ Directory Reference

Files

- file arg.c

  Argument buffer for passing arguments when starting processes.

- file cc.h

  Default definitions of C compiler quirk work-arounds.

- file **clock.h**
- file dsc.h

  Declaration of the DSC program description structure.

- file etimer.c

  Event timer library implementation.

- file etimer.h

  Event timer header file.

- file lc-addrlabels.h

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

- file lc-switch.h

  Implementation of local continuations based on switch() statment.

- file lc.h

  Local continuations.

- file loader.h

Default definitions and error values for the Contiki program loader.

- file mt.c

  Implementation of the archtecture agnostic parts of the preemptive multithreading library for Contiki.

- file mt.h

  Header file for the preemptive multitasking library for Contiki.

- file process.c

  Implementation of the Contiki process kernel.

- file process.h

  Header file for the Contiki process interface.

- file **procinit.c**
- file **procinit.h**
- file pt-sem.h

  Counting semaphores implemented on protothreads.

- file pt.h

  Protothreads implementation.

- file timer.c

  Timer library implementation.

- file timer.h

  Timer library header file.

8 Contiki 2.x Data Structure Documentation

8.1 abc_callbacks Struct Reference

`#include <abc.h>`

8.1.1 Detailed Description

Callback structure for abc.

Examples:

test-abc.c.

Definition at line 69 of file abc.h.

Data Fields

- void(∗ recv)(struct abc_conn ∗ptr)

  Called when a packet has been received by the abc module.

8.2 ctk_menu Struct Reference

`#include <ctk.h>`

8.2.1 Detailed Description

Representation of an individual menu.

Definition at line 567 of file ctk.h.

Data Fields

- ctk_menu ∗ next

  Apointer to the next menu, or is NULL if this is the last menu, and should be used by the ctk-draw module when stepping through the menus when drawing them on screen.

- char ∗ title

  The menu title.

- unsigned char titlelen

  The length of the title in characters.

- unsigned char nitems

  The total number of menu items in the menu.

- unsigned char active

  The currently active menu item.

- ctk_menuitem items [CTK_MAXMENUITEMS]

  The array which contains all the menu items.

8.2.2 Field Documentation

8.2.2.1 unsigned char ctk_menu::titlelen

The length of the title in characters.

Cached for speed reasons.

Definition at line 574 of file ctk.h.

Referenced by ctk_menu_new().

8.3 ctk_menuitem Struct Reference

`#include <ctk.h>`

8.3.1 Detailed Description

Representation of an individual menu item.

Definition at line 552 of file ctk.h.

Data Fields

- char ∗ title

  The menu items text.

- unsigned char titlelen

  The length of the item text, cached for speed.

8.4 ctk_menus Struct Reference

`#include <ctk.h>`

8.4.1 Detailed Description

Representation of the menu bar.

Definition at line 592 of file ctk.h.

Data Fields

- ctk_menu ∗ menus

  A pointer to a linked list of all menus, including the open menu and the desktop menu.

- ctk_menu ∗ open

  The currently open menu, if any.

- ctk_menu ∗ desktopmenu

  A pointer to the "Desktop" menu that can be used for drawing the desktop menu in a special way (such as drawing it at the rightmost position).

8.4.2 Field Documentation

8.4.2.1 struct ctk_menu∗ ctk_menus::open

The currently open menu, if any.

If all menus are closed, this item is NULL:

Definition at line 596 of file ctk.h.

Referenced by ctk_window_redraw().

8.5 ctk_widget Struct Reference

`#include <ctk.h>`

8.5.1 Detailed Description

The generic CTK widget structure that contains all other widget structures.

Since the widgets of a window are arranged on a linked list, the widget structure contains a next pointer which is used for this purpose. The widget structure also contains the placement and the size of the widget.

Finally, the actual per-widget structure is contained in this top-level widget structure.

Definition at line 427 of file ctk.h.

Data Fields

- ctk_widget ∗ next

  The next widget in the linked list of widgets that is contained in the ctk_window structure.

- ctk_window ∗ window

  The window in which the widget is contained.

- unsigned char x

  The x position of the widget within the containing window, in character coordinates.

- unsigned char y

  The y position of the widget within the containing window, in character coordinates.

- unsigned char type

  The type of the widget: CTK_WIDGET_SEPARATOR, CTK_WIDGET_LABEL, CTK_WIDGET_-BUTTON, CTK_WIDGET_HYPERLINK, CTK_WIDGET_TEXTENTRY, CTK_WIDGET_BITMAP or CTK_WIDGET_ICON.

- unsigned char w

  The width of the widget in character coordinates.

- unsigned char h

  The height of the widget in character coordinates.

- union {
  } widget

  The union which contains the actual widget structure, as determined by the type field.

8.6 ctk_window Struct Reference

```
#include <ctk.h>
```

8.6.1 Detailed Description

Representation of a CTK window.

For the CTK, each window is repessented by a ctk_window structure. All open windows are kept on a doubly linked list, linked by the next and prev fields in the ctk_window struct. The window structure holds all widgets that is contained in the window as well as a pointer to the currently selected widget.

Definition at line 489 of file ctk.h.

Data Fields

- ctk_window * next

  The next window in the doubly linked list of open windows.

- ctk_window * prev

  The previous window in the doubly linked list of open windows.

- ctk_desktop * desktop

  The desktop on which this window is open.

- process * owner

  The process that owns the window.

- char * title

  The title of the window.

- unsigned char titlelen

  The length of the title, cached for speed reasons.

- unsigned char x

  The x coordinate of the window, in characters.

- unsigned char y

  The y coordinate of the window, in characters.

- unsigned char w

  The width of the window, excluding window borders.

- unsigned char h

  The height of the window, excluding window borders.

- ctk_widget * inactive

  The list if widgets that cannot be selected by the user.

- ctk_widget * active

  The list of widgets that can be selected by the user.

- ctk_widget * focused

  A pointer to the widget on the active list that is currently selected, or NULL if no widget is selected.

8.6.2 Field Documentation

8.6.2.1 struct ctk_widget* ctk_window::active

The list of widgets that can be selected by the user.

Buttons, hyperlinks, text entry fields, etc., are placed on this list.

Definition at line 539 of file ctk.h.

Referenced by ctk_widget_add(), and ctk_window_clear().

8.6.2.2 struct ctk_widget* ctk_window::inactive

The list if widgets that cannot be selected by the user.

Labels and separator widgets are placed on this list.

Definition at line 535 of file ctk.h.

Referenced by ctk_widget_add().

8.6.2.3 struct process* ctk_window::owner

The process that owns the window.

This process will be the receiver of all CTK signals that pertain to this window.

Definition at line 498 of file ctk.h.

8.6.2.4 char* ctk_window::title

The title of the window.

Used for constructing the "Dekstop" menu.

Definition at line 503 of file ctk.h.

8.7 dsc Struct Reference

```
#include <dsc.h>
```

8.7.1 Detailed Description

The DSC program description structure.

The DSC structure is used for describing a Contiki program. It includes a short textual description of the program, either the name of the program on disk, or a pointer to the init() function, and an icon for the program.

Definition at line 75 of file dsc.h.

Data Fields

- char * description

  A text string containing a one-line description of the program.

- char * prgname

  The name of the program on disk.

- ctk_icon * icon

  A pointer to the ctk_icon structure for the DSC.

- void * loadaddr

The loading address of the DSC.

8.7.2 Field Documentation

8.7.2.1 void∗ dsc::loadaddr

The loading address of the DSC.

Used by the LOADER_UNLOAD() function when deallocating the memory allocated for the DSC when loading it.

Definition at line 89 of file dsc.h.

8.8 etimer Struct Reference

`#include <etimer.h>`

8.8.1 Detailed Description

A timer.

This structure is used for declaring a timer. The timer must be set with etimer_set() before it can be used.

Examples:

example-program.c, test-abc.c, test-meshroute.c, and test-treeroute.c.

Definition at line 77 of file etimer.h.

8.9 ibc_callbacks Struct Reference

`#include <ibc.h>`

8.9.1 Detailed Description

Callback structure for abc.

Definition at line 71 of file ibc.h.

Data Fields

- void(∗ recv)(struct ibc_conn ∗ptr, rimeaddr_t ∗sender)

  Called when a packet has been received by the ibc module.

8.10 mesh_callbacks Struct Reference

`#include <mesh.h>`

8.10.1 Detailed Description

Mesh callbacks.

Examples:
test-meshroute.c.

Definition at line 74 of file mesh.h.

Data Fields

- void(∗ recv)(struct mesh_conn ∗c, rimeaddr_t ∗from)
  Called when a packet is received.
- void(∗ sent)(struct mesh_conn ∗c)
  Called when a packet, sent with mesh_send(), is actually transmitted.
- void(∗ timedout)(struct mesh_conn ∗c)
  Called when a packet, sent with mesh_send(), times out and is dropped.

8.11 psock Struct Reference

`#include <psock.h>`

8.11.1 Detailed Description

The representation of a protosocket.

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

Examples:
example-psock-server.c.

Definition at line 113 of file psock.h.

8.12 radio_driver Struct Reference

`#include <radio.h>`

8.12.1 Detailed Description

The structure of a device driver for a radio in Contiki.

Definition at line 61 of file radio.h.

Data Fields

- int(∗ send)(const void ∗payload, unsigned short payload_len)
  Send a packet.

- int(∗ read)(void ∗buf, unsigned short buf_len)

  Read a received packet into a buffer.

- void(∗ set_receive_function)(void(∗f)(const struct radio_driver ∗d))

  Set a function to be called when a packet has been received.

- int(∗ on)(void)

  Turn the radio on.

- int(∗ off)(void)

  Turn the radio off.

8.13 sabc_conn Struct Reference

`#include <sabc.h>`

8.13.1 Detailed Description

A sabc connection.

This is an opaque structure with no user-visible fields. The sabc_open() function is used for setting up a sabc connection.

Definition at line 80 of file sabc.h.

8.14 timer Struct Reference

`#include <timer.h>`

8.14.1 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.

Definition at line 87 of file timer.h.

8.15 uip_conn Struct Reference

`#include <uip.h>`

8.15.1 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 1201 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.

8.16 uip_eth_addr Struct Reference

`#include <uip.h>`

8.16.1 Detailed Description

Representation of a 48-bit Ethernet address.

Definition at line 1590 of file uip.h.

8.17 uip_eth_hdr Struct Reference

`#include <uip_arp.h>`

8.17.1 Detailed Description

The Ethernet header.

Definition at line 63 of file uip_arp.h.

8.18 uip_fw_netif Struct Reference

`#include <uip-fw.h>`

8.18.1 Detailed Description

Representation of a uIP network interface.

Definition at line 54 of file uip-fw.h.

Data Fields

- uip_fw_netif ∗ next

  Pointer to the next interface when linked in a list.

- uip_ipaddr_t ipaddr

  The IP address of this interface.

- uip_ipaddr_t netmask

  The netmask of the interface.

- u8_t(∗ output)(void)

  A pointer to the function that sends a packet.

8.19 uip_ip4addr_t Union Reference

`#include <uip.h>`

8.19.1 Detailed Description

Representation of an IP address.

Definition at line 62 of file uip.h.

8.20 uip_stats Struct Reference

`#include <uip.h>`

8.20.1 Detailed Description

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

Definition at line 1280 of file uip.h.

Data Fields

- struct {
  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 drop
  Number of dropped 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 recv
  Number of received ICMP packets.
  uip_stats_t sent
  Number of sent ICMP packets.
  uip_stats_t drop
  Number of dropped ICMP packets.
  uip_stats_t typeerr
  Number of ICMP packets with a wrong type.
  } icmp

ICMP statistics.

- struct {
  uip_stats_t recv
  Number of recived TCP segments.
  uip_stats_t sent
  Number of sent TCP segments.
  uip_stats_t drop
  Number of dropped 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 recevied 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 avaliable.
  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 recived 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.

8.21 uip_udp_conn Struct Reference

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

8.21.1 Detailed Description

Representation of a uIP UDP connection.

Examples:
example-program.c.

Definition at line 1258 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.

9 Contiki 2.x File Documentation

9.1 apps/program-handler/program-handler.c File Reference

9.1.1 Detailed Description

The program handler, used for loading programs and starting the screensaver.

Author:

Adam Dunkels <adam@dunkels.com>

The Contiki program handler is responsible for the Contiki menu and the desktop icons, as well as for loading programs and displaying a dialog with a message telling which program that is loading.

The program handler also is responsible for starting the screensaver when the CTK detects that it should be started.

Definition in file program-handler.c.

```
#include <string.h>
#include <stdlib.h>
#include "contiki.h"
#include "ctk/ctk.h"
#include "ctk/ctk-draw.h"
#include "program-handler.h"
```

Defines

- #define NUM_PNARGS 6

    Initializes the program handler.

Functions

- void program_handler_add (struct dsc *dsc, char *menuname, unsigned char desktop)
  Add a program to the program handler.
- void program_handler_load (char *name, char *arg)
  Loads a program and displays a dialog telling the user about it.

9.1.2 Define Documentation

9.1.2.1 #define NUM_PNARGS 6

Initializes the program handler.

Is called by the initialization before any programs have been added with program_handler_add().

Definition at line 180 of file program-handler.c.

9.1.3 Function Documentation

9.1.3.1 void program_handler_add (struct dsc * *dsc*, char * *menuname*, unsigned char *desktop*)

Add a program to the program handler.

Parameters:

dsc The DSC description structure for the program to be added.

menuname The name that the program should have in the Contiki menu.

desktop Flag which specifies if the program should show up as an icon on the desktop or not.

Definition at line 161 of file program-handler.c.

References CTK_ICON_ADD, and ctk_menuitem_add().

9.1.3.2 void program_handler_load (char * *name*, char * *arg*)

Loads a program and displays a dialog telling the user about it.

Parameters:

name The name of the program to be loaded.

arg An argument which is passed to the new process when it is loaded.

Definition at line 223 of file program-handler.c.

References ctk_dialog_open(), ctk_label_set_text, and process_post().

9.2 core/cfs/cfs.h File Reference

9.2.1 Detailed Description

CFS header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file cfs.h.

```
#include "contiki.h"
```

Defines

- #define CFS_READ 1

  Specify that cfs_open() should open a file for reading.

- #define CFS_WRITE 2

  Specify that cfs_open() should open a file for writing.

- #define CFS_APPEND 4

  Specify that cfs_open() should append written data to the file rather than overwriting it.

Functions

- CCIF int cfs_open (const char ∗name, int flags)

  Open a file.

- CCIF void cfs_close (int fd)

  Close an open file.

- CCIF int cfs_read (int fd, char ∗buf, unsigned int len)

  Read data from an open file.

- CCIF int cfs_write (int fd, char ∗buf, unsigned int len)

  Write data to an open file.

- CCIF int cfs_seek (int fd, unsigned int offset)

  Seek to a specified position in an open file.

- CCIF int cfs_opendir (struct cfs_dir ∗dirp, const char ∗name)

  Open a directory for reading directory entries.

- CCIF int cfs_readdir (struct cfs_dir ∗dirp, struct cfs_dirent ∗dirent)

  Read a directory entry.

- CCIF int cfs_closedir (struct cfs_dir ∗dirp)

  Close a directory opened with cfs_opendir().

9.3 core/ctk/ctk-draw.h File Reference

9.3.1 Detailed Description

CTK screen drawing module interface, ctk-draw.

Author:

Adam Dunkels <adam@dunkels.com>

This file contains the interface for the ctk-draw module.The ctk-draw module takes care of the actual screen drawing for CTK by implementing a handful of functions that are called by CTK.

Definition in file ctk-draw.h.

```
#include "ctk/ctk.h"
#include "contiki-conf.h"
```

Functions

- void ctk_draw_init (void)

  The initialization function.

- void ctk_draw_clear (unsigned char clipy1, unsigned char clipy2)

  Clear the screen between the clip bounds.

- void ctk_draw_clear_window (struct ctk_window *window, unsigned char focus, unsigned char clipy1, unsigned char clipy2)

  Draw the window background.

- void ctk_draw_window (struct ctk_window *window, unsigned char focus, unsigned char clipy1, unsigned char clipy2, unsigned char draw_borders)

  Draw a window onto the screen.

- void ctk_draw_dialog (struct ctk_window *dialog)

  Draw a dialog onto the screen.

- void ctk_draw_widget (struct ctk_widget *w, unsigned char focus, unsigned char clipy1, unsigned char clipy2)

  Draw a widget on a window.

9.4 core/ctk/ctk.c File Reference

9.4.1 Detailed Description

The Contiki Toolkit CTK, the Contiki GUI.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file ctk.c.

```
#include <string.h>
#include "contiki.h"
#include "ctk/ctk.h"
#include "ctk/ctk-draw.h"
#include "ctk/ctk-mouse.h"
```

Functions

- void ctk_mode_set (unsigned char m)

  Sets the current CTK mode.

- unsigned char ctk_mode_get (void)

  Retrieves the current CTK mode.

- void ctk_icon_add (CC_REGISTER_ARG struct ctk_widget ∗icon, struct process ∗p)

  Add an icon to the desktop.

- void ctk_dialog_open (struct ctk_window ∗d)

  Open a dialog box.

- void ctk_dialog_close (void)

  Close the dialog box, if one is open.

- void ctk_window_open (CC_REGISTER_ARG struct ctk_window ∗w)

  Open a window, or bring window to front if already open.

- void ctk_window_close (struct ctk_window ∗w)

  Close a window if it is open.

- void ctk_window_clear (struct ctk_window ∗w)

  Remove all widgets from a window.

- void ctk_menu_add (struct ctk_menu ∗menu)

  Add a menu to the menu bar.

- void ctk_menu_remove (struct ctk_menu ∗menu)

  Remove a menu from the menu bar.

- void ctk_window_redraw (struct ctk_window ∗w)

  Redraw a window.

- void ctk_window_new (struct ctk_window ∗window, unsigned char w, unsigned char h, char ∗title)

  Create a new window.

- void ctk_dialog_new (CC_REGISTER_ARG struct ctk_window ∗dialog, unsigned char w, unsigned char h)

  Creates a new dialog.

- void ctk_menu_new (CC_REGISTER_ARG struct ctk_menu *menu, char *title)

  Creates a new menu.

- unsigned char ctk_menuitem_add (CC_REGISTER_ARG struct ctk_menu *menu, char *name)

  Adds a menu item to a menu.

- void CC_FASTCALL ctk_widget_add (CC_REGISTER_ARG struct ctk_window *window, CC_-REGISTER_ARG struct ctk_widget *widget)

  Adds a widget to a window.

Variables

- process_event_t ctk_signal_keypress

  Emitted for every key being pressed.

- process_event_t ctk_signal_widget_activate

  Emitted when a widget is activated (pressed).

- process_event_t ctk_signal_button_activate

  Same as ctk_signal_widget_activate.

- process_event_t ctk_signal_widget_select

  Emitted when a widget is selected.

- process_event_t ctk_signal_button_hover

  Same as ctk_signal_widget_select.

- process_event_t ctk_signal_hyperlink_activate

  Emitted when a hyperlink is activated.

- process_event_t ctk_signal_hyperlink_hover

  Same as ctk_signal_widget_select.

- process_event_t ctk_signal_menu_activate

  Emitted when a menu item is activated.

- process_event_t ctk_signal_window_close

  Emitted when a window is closed.

- process_event_t ctk_signal_pointer_move

  Emitted when the mouse pointer is moved.

- process_event_t ctk_signal_pointer_button

  Emitted when a mouse button is pressed.

9.5 core/ctk/ctk.h File Reference

9.5.1 Detailed Description

CTK header file.

Author:

Adam Dunkels <adam@dunkels.com>

The CTK header file contains functioin declarations and definitions of CTK structures and macros.

Definition in file ctk.h.

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

Defines

- #define CTK_WIDGET_SEPARATOR 1

  Widget number: The CTK separator widget.

- #define CTK_WIDGET_LABEL 2

  Widget number: The CTK label widget.

- #define CTK_WIDGET_BUTTON 3

  Widget number: The CTK button widget.

- #define CTK_WIDGET_HYPERLINK 4

  Widget number: The CTK hyperlink widget.

- #define CTK_WIDGET_TEXTENTRY 5

  Widget number: The CTK textentry widget.

- #define CTK_WIDGET_BITMAP 6

  Widget number: The CTK bitmap widget.

- #define CTK_WIDGET_ICON 7

  Widget number: The CTK icon widget.

- #define CTK_SEPARATOR(x, y, w) NULL, NULL, x, y, CTK_WIDGET_SEPARATOR, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0)

  Instantiating macro for the ctk_separator widget.

- #define CTK_BUTTON(x, y, w, text) NULL, NULL, x, y, CTK_WIDGET_BUTTON, w, 1, CTK_-WIDGET_FLAG_INITIALIZER(0) text

  Instantiating macro for the ctk_button widget.

- #define CTK_LABEL(x, y, w, h, text) NULL, NULL, x, y, CTK_WIDGET_LABEL, w, h, CTK_-WIDGET_FLAG_INITIALIZER(0) text,

  Instantiating macro for the ctk_label widget.

- #define CTK_HYPERLINK(x, y, w, text, url) NULL, NULL, x, y, CTK_WIDGET_HYPERLINK, w, 1, CTK_WIDGET_FLAG_INITIALIZER(0) text, url

    Instantiating macro for the ctk_hyperlink widget.

- #define CTK_TEXTENTRY_CLEAR(e)

    Clears a text entry widget and sets the cursor to the start of the text line.

- #define CTK_TEXTENTRY(x, y, w, h, text, len)

    Instantiating macro for the ctk_textentry widget.

- #define CTK_ICON(title, bitmap, textmap)

    Instantiating macro for the ctk_icon widget.

- #define CTK_ICON_ADD(icon, p) ctk_icon_add((struct ctk_widget ∗)icon, p)

    Add an icon to the desktop.

- #define CTK_WIDGET_ADD(win, widg) ctk_widget_add(win, (struct ctk_widget ∗)widg)

    Add a widget to a window.

- #define CTK_WIDGET_FOCUS(win, widg) (win) → focused = (struct ctk_widget ∗)(widg)

    Set focus to a widget.

- #define CTK_WIDGET_REDRAW(widg) ctk_widget_redraw((struct ctk_widget ∗)widg)

    Add a widget to the redraw queue.

- #define CTK_WIDGET_TYPE(w) ((w) → type)

    Obtain the type of a widget.

- #define CTK_WIDGET_SET_WIDTH(widget, width)

    Sets the width of a widget.

- #define CTK_WIDGET_XPOS(w) (((struct ctk_widget ∗)(w)) → x)

    Retrieves the x position of a widget, relative to the window in which the widget is contained.

- #define CTK_WIDGET_SET_XPOS(w, xpos) ((struct ctk_widget ∗)(w)) → x = (xpos)

    Sets the x position of a widget, relative to the window in which the widget is contained.

- #define CTK_WIDGET_YPOS(w) (((struct ctk_widget ∗)(w)) → y)

    Retrieves the y position of a widget, relative to the window in which the widget is contained.

- #define CTK_WIDGET_SET_YPOS(w, ypos) ((struct ctk_widget ∗)(w)) → y = (ypos)

    Sets the y position of a widget, relative to the window in which the widget is contained.

- #define ctk_label_set_height(w, height) (w) → widget.label.h = (height)

    Set the height of a label.

- #define ctk_label_set_text(l, t) (l) → text = (t)

    Set the text of a label.

- #define ctk_button_set_text(b, t) (b) → text = (t)

Set the text of a button.

- #define CTK_FOCUS_NONE 0

  Widget focus flag: no focus.

- #define CTK_FOCUS_WIDGET 1

  Widget focus flag: widget has focus.

- #define CTK_FOCUS_WINDOW 2

  Widget focus flag: widget's window is the foremost one.

- #define CTK_FOCUS_DIALOG 4

  Widget focus flag: widget is in a dialog.

Typedefs

- typedef char ctk_arch_key_t

  The keyboard character type of the system.

Functions

- void ctk_mode_set (unsigned char mode)

  Sets the current CTK mode.

- unsigned char ctk_mode_get (void)

  Retrieves the current CTK mode.

- CCIF void ctk_window_new (struct ctk_window ∗window, unsigned char w, unsigned char h, char ∗title)

  Create a new window.

- CCIF void ctk_window_clear (struct ctk_window ∗w)

  Remove all widgets from a window.

- CCIF void ctk_window_close (struct ctk_window ∗w)

  Close a window if it is open.

- CCIF void ctk_window_redraw (struct ctk_window ∗w)

  Redraw a window.

- CCIF void ctk_dialog_open (struct ctk_window ∗d)

  Open a dialog box.

- CCIF void ctk_dialog_close (void)

  Close the dialog box, if one is open.

- CCIF void ctk_menu_add (struct ctk_menu ∗menu)

Add a menu to the menu bar.

- CCIF void ctk_menu_remove (struct ctk_menu *menu)

  Remove a menu from the menu bar.

- CCIF void ctk_widget_redraw (struct ctk_widget *w)

  Redraws a widget.

- void ctk_desktop_redraw (struct ctk_desktop *d)

  Redraw the entire desktop.

- CCIF unsigned char ctk_desktop_width (struct ctk_desktop *d)

  Gets the width of the desktop.

- unsigned char ctk_desktop_height (struct ctk_desktop *d)

  Gets the height of the desktop.

Variables

- CCIF process_event_t ctk_signal_keypress

  Emitted for every key being pressed.

- CCIF process_event_t ctk_signal_widget_activate

  Emitted when a widget is activated (pressed).

- CCIF process_event_t ctk_signal_widget_select

  Emitted when a widget is selected.

- CCIF process_event_t ctk_signal_menu_activate

  Emitted when a menu item is activated.

- CCIF process_event_t ctk_signal_window_close

  Emitted when a window is closed.

- CCIF process_event_t ctk_signal_pointer_move

  Emitted when the mouse pointer is moved.

- CCIF process_event_t ctk_signal_pointer_button

  Emitted when a mouse button is pressed.

- CCIF process_event_t ctk_signal_button_activate

  Same as ctk_signal_widget_activate.

- CCIF process_event_t ctk_signal_button_hover

  Same as ctk_signal_widget_select.

- CCIF process_event_t ctk_signal_hyperlink_activate

  Emitted when a hyperlink is activated.

- CCIF process_event_t ctk_signal_hyperlink_hover
  Same as ctk_signal_widget_select.

9.6 core/dev/eeprom.h File Reference

9.6.1 Detailed Description

EEPROM functions.

Author:
Adam Dunkels <adam@sics.se>

Definition in file eeprom.h.

Functions

- void eeprom_write (eeprom_addr_t addr, unsigned char *buf, int size)
  Write a buffer into EEPROM.
- void eeprom_read (eeprom_addr_t addr, unsigned char *buf, int size)
  Read data from the EEPROM.
- void eeprom_init (void)
  Initialize the EEPROM module.

9.7 core/dev/radio.h File Reference

9.7.1 Detailed Description

Header file for the radio API.

Author:
Adam Dunkels <adam@sics.se>

Definition in file radio.h.

9.8 core/lib/crc16.c File Reference

9.8.1 Detailed Description

Implementation of the CRC16 calculcation.

Author:
Adam Dunkels <adam@sics.se>

Definition in file crc16.c.

9.9 core/lib/crc16.h File Reference

9.9.1 Detailed Description

Header file for the CRC16 calculcation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file crc16.h.

Functions

- unsigned short crc16_add (unsigned char b, unsigned short crc)

  Update an accumulated CRC16 checksum with one byte.

9.10 core/lib/ctk-textedit.c File Reference

9.10.1 Detailed Description

An experimental CTK text edit widget.

Author:

Adam Dunkels <adam@dunkels.com>

This module contains an experimental CTK widget which is implemented in the application process rather than in the CTK process. The widget is instantiated in a similar fashion as other CTK widgets, but is different from other widgets in that it requires a signal handler function to be called by the process signal handler function.

Definition in file ctk-textedit.c.

```
#include "ctk-textedit.h"
#include <string.h>
```

Functions

- void ctk_textedit_add (struct ctk_window *w, struct ctk_textedit *t)

  Add a CTK textedit widget to a window.

- void ctk_textedit_eventhandler (struct ctk_textedit *t, process_event_t s, process_data_t data)

  The CTK textedit signal handler.

9.10.2 Function Documentation

9.10.2.1 void ctk_textedit_add (struct ctk_window * *w*, struct ctk_textedit * *t*)

Add a CTK textedit widget to a window.

Parameters:

w A pointer to the window to which the entry is to be added.

t A pointer to the CTK textentry structure.

Definition at line 70 of file ctk-textedit.c.

References CTK_WIDGET_ADD.

9.10.2.2 void ctk_textedit_eventhandler (struct ctk_textedit ∗ *t*, process_event_t *s*, process_data_t *data*)

The CTK textedit signal handler.

This function must be called as part of the normal signal handler of the process that contains the CTK textentry structure.

Parameters:

t A pointer to the CTK textentry structure.

s The signal number.

data The signal data.

Definition at line 89 of file ctk-textedit.c.

References ctk_signal_keypress, ctk_signal_widget_activate, CTK_WIDGET_FOCUS, and CTK_-WIDGET_REDRAW.

9.11 core/lib/ctk-textedit.h File Reference

9.11.1 Detailed Description

Header file for the experimental application level CTK textedit widget.

Author:

Adam Dunkels <`adam@dunkels.com`>

Definition in file ctk-textedit.h.

```
#include "ctk/ctk.h"
```

Defines

- #define CTK_TEXTEDIT(tx, ty, tw, th, ttext) {CTK_LABEL(tx, ty, tw, th, ttext)}, 0, 0

  Instantiating macro for the CTK textedit widget.

Functions

- void ctk_textedit_add (struct ctk_window ∗w, struct ctk_textedit ∗t)

  Add a CTK textedit widget to a window.

- void ctk_textedit_eventhandler (struct ctk_textedit ∗t, process_event_t s, process_data_t data)

  The CTK textedit signal handler.

9.11.2 Define Documentation

9.11.2.1 #define CTK_TEXTEDIT(tx, ty, tw, th, ttext) {CTK_LABEL(tx, ty, tw, th, ttext)}, 0, 0

Instantiating macro for the CTK textedit widget.

Parameters:

- ***tx*** The x position of the widget.
- ***ty*** The y position of the widget.
- ***tw*** The width of the widget.
- ***th*** The height of the widget.
- ***ttext*** The text buffer to be edited.

Definition at line 57 of file ctk-textedit.h.

9.11.3 Function Documentation

9.11.3.1 void ctk_textedit_add (struct ctk_window ∗ *w*, struct ctk_textedit ∗ *t*)

Add a CTK textedit widget to a window.

Parameters:

- ***w*** A pointer to the window to which the entry is to be added.
- ***t*** A pointer to the CTK textentry structure.

Definition at line 70 of file ctk-textedit.c.

References CTK_WIDGET_ADD.

9.11.3.2 void ctk_textedit_eventhandler (struct ctk_textedit ∗ *t*, process_event_t *s*, process_data_t *data*)

The CTK textedit signal handler.

This function must be called as part of the normal signal handler of the process that contains the CTK textentry structure.

Parameters:

- ***t*** A pointer to the CTK textentry structure.
- ***s*** The signal number.
- ***data*** The signal data.

Definition at line 89 of file ctk-textedit.c.

References ctk_signal_keypress, ctk_signal_widget_activate, CTK_WIDGET_FOCUS, and CTK_-WIDGET_REDRAW.

9.12 core/lib/list.c File Reference

9.12.1 Detailed Description

Linked list library implementation.

Author:
Adam Dunkels <adam@sics.se>

Definition in file list.c.

```
#include "lib/list.h"
```

9.13 core/lib/list.h File Reference

9.13.1 Detailed Description

Linked list manipulation routines.

Author:
Adam Dunkels <adam@sics.se>

Definition in file list.h.

Defines

- #define LIST(name)
  Declare a linked list.

Typedefs

- typedef void ** list_t
  The linked list type.

Functions

- void list_init (list_t list)
  Initialize a list.
- void * list_head (list_t list)
  Get a pointer to the first element of a list.
- void * list_tail (list_t list)
  Get the tail of a list.
- void * list_pop (list_t list)
  Remove the first object on a list.
- void list_push (list_t list, void *item)
  Add an item to the start of the list.
- void * list_chop (list_t list)
  Remove the last object on the list.

- void list_add (list_t list, void ∗item)

  Add an item at the end of a list.

- void list_remove (list_t list, void ∗item)

  Remove a specific element from a list.

- int list_length (list_t list)

  Get the length of a list.

- void list_copy (list_t dest, list_t src)

  Duplicate a list.

- void list_insert (list_t list, void ∗previtem, void ∗newitem)

  Insert an item after a specified item on the list.

9.14 core/lib/me.c File Reference

9.14.1 Detailed Description

Implementation of the table-driven Manchester encoding and decoding.

Author:

Adam Dunkels <adam@sics.se>

Definition in file me.c.

```
#include "me_tabs.h"
```

9.15 core/lib/me.h File Reference

9.15.1 Detailed Description

Header file for the table-driven Manchester encoding and decoding.

Author:

Adam Dunkels <adam@sics.se>

Definition in file me.h.

Functions

- unsigned char me_valid (unsigned char m)

  Check if an encoded byte is valid.

- unsigned short me_encode (unsigned char c)

  Manchester encode an 8-bit byte.

- unsigned char me_decode16 (unsigned short m)

Decode a Manchester encoded 16-bit word.

- unsigned char me_decode8 (unsigned char m)

  Decode a Manchester encoded 8-bit byte.

9.16 core/lib/memb.c File Reference

9.16.1 Detailed Description

Memory block allocation routines.

Author:

Adam Dunkels <adam@sics.se>

Definition in file memb.c.

```
#include <string.h>
#include "contiki.h"
#include "lib/memb.h"
```

9.17 core/lib/memb.h File Reference

9.17.1 Detailed Description

Memory block allocation routines.

Author:

Adam Dunkels <adam@sics.se>

Definition in file memb.h.

Defines

- #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().

9.18 core/lib/mmem.c File Reference

9.18.1 Detailed Description

Implementation of the managed memory allocator.

Author:

Adam Dunkels <adam@sics.se>

Definition in file mmem.c.

```
#include "mmem.h"
#include "list.h"
#include "contiki-conf.h"
#include <string.h>
```

9.19 core/lib/mmem.h File Reference

9.19.1 Detailed Description

Header file for the managed memory allocator.

Author:

Adam Dunkels <adam@sics.se>

Definition in file mmem.h.

Defines

- #define MMEM_PTR(m)

  Get a pointer to the managed memory.

Functions

- int mmem_alloc (struct mmem *m, unsigned int size)

  Allocate a managed memory block.

- void mmem_free (struct mmem *)

  Deallocate a managed memory block.

- void mmem_init (void)

  Initialize the managed memory module.

9.20 core/lib/petsciiconv.h File Reference

9.20.1 Detailed Description

PETSCII/ASCII conversion functions.

Author:

Adam Dunkels <adam@dunkels.com>

The Commodore based Contiki targets all have a special character encoding called PETSCII which differs from the ASCII encoding that normally is used for representing characters.

Note:

For targets that do not use PETSCII encoding the C compiler define WITH_ASCII should be used to avoid the PETSCII converting functions.

Definition in file petsciiconv.h.

9.21 core/loader/elfloader-arch.h File Reference

9.21.1 Detailed Description

Header file for the architecture specific parts of the Contiki ELF loader.

Author:

Adam Dunkels <adam@sics.se>

Definition in file elfloader-arch.h.

```
#include "loader/elfloader.h"
```

Functions

- void ∗ elfloader_arch_allocate_ram (int size)
  Allocate RAM for a new module.
- void ∗ elfloader_arch_allocate_rom (int size)
  Allocate program memory for a new module.
- void elfloader_arch_relocate (int fd, unsigned int sectionoffset, char ∗sectionaddr, struct elf32_rela ∗rela, char ∗addr)
  Perform a relocation.
- void elfloader_arch_write_rom (int fd, unsigned short textoff, unsigned int size, char ∗mem)
  Write to read-only memory (for example the text segment).

9.22 core/loader/elfloader.h File Reference

9.22.1 Detailed Description

Header file for the Contiki ELF loader.

Author:

Adam Dunkels <adam@sics.se>

Definition in file elfloader.h.

```
#include "cfs/cfs.h"
```

Defines

- #define ELFLOADER_OK 0

  Return value from elfloader_load() indicating that loading worked.

- #define ELFLOADER_BAD_ELF_HEADER 1

  Return value from elfloader_load() indicating that the ELF file had a bad header.

- #define ELFLOADER_NO_SYMTAB 2

  Return value from elfloader_load() indicating that no symbol table could be find in the ELF file.

- #define ELFLOADER_NO_STRTAB 3

  Return value from elfloader_load() indicating that no string table could be find in the ELF file.

- #define ELFLOADER_NO_TEXT 4

  Return value from elfloader_load() indicating that the size of the .text segment was zero.

- #define ELFLOADER_SYMBOL_NOT_FOUND 5

  Return value from elfloader_load() indicating that a symbol specific symbol could not be found.

- #define ELFLOADER_SEGMENT_NOT_FOUND 6

  Return value from elfloader_load() indicating that one of the required segments (.data, .bss, or .text) could not be found.

- #define ELFLOADER_NO_STARTPOINT 7

  Return value from elfloader_load() indicating that no starting point could be found in the loaded module.

Functions

- void elfloader_init (void)

  elfloader initialization function.

- int elfloader_load (int fd)

  Load and relocate an ELF file.

Variables

- process ** elfloader_autostart_processes

  A pointer to the processes loaded with elfloader_load().

- char elfloader_unknown [30]

  If elfloader_load() could not find a specific symbol, it is copied into this array.

9.23 core/net/psock.h File Reference

9.23.1 Detailed Description

Protosocket library header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file psock.h.

```
#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
```

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.

9.24 core/net/resolv.c File Reference

9.24.1 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 "net/tcpip.h"
#include "net/resolv.h"
#include <string.h>
```

Functions

- void resolv_query (char ∗name)
  Queues a name so that a question for the name will be sent out.
- u16_t ∗ resolv_lookup (char ∗name)
  Look up a hostname in the array of known hostnames.
- uip_ipaddr_t ∗ resolv_getserver (void)
  Obtain the currently configured DNS server.
- void resolv_conf (const uip_ipaddr_t ∗dnsserver)
  Configure a DNS server.

Variables

- process_event_t resolv_event_found
  Event that is broadcasted when a DNS name has been resolved.

9.25 core/net/resolv.h File Reference

9.25.1 Detailed Description

uIP DNS resolver code header file.

Author:
Adam Dunkels <adam@dunkels.com>

Definition in file resolv.h.

```
#include "contiki.h"
```

Functions

- CCIF void resolv_conf (const uip_ipaddr_t *dnsserver)
  Configure a DNS server.
- CCIF uip_ipaddr_t * resolv_getserver (void)
  Obtain the currently configured DNS server.
- CCIF u16_t * resolv_lookup (char *name)
  Look up a hostname in the array of known hostnames.
- CCIF void resolv_query (char *name)
  Queues a name so that a question for the name will be sent out.

Variables

- CCIF process_event_t resolv_event_found
  Event that is broadcasted when a DNS name has been resolved.

9.26 core/net/rime.h File Reference

9.26.1 Detailed Description

Header file for the Rime stack.

Author:
Adam Dunkels <adam@sics.se>

Definition in file rime.h.

```
#include "net/rime/rimestats.h"
#include "net/rime/rimeaddr.h"
#include "net/rime/ctimer.h"
#include "net/rime/rimebuf.h"
#include "net/rime/queuebuf.h"
#include "net/rime/ruc.h"
#include "net/rime/sibc.h"
#include "net/mac/mac.h"
```

Functions

- void rime_init (const struct mac_driver ∗)
  Initialize Rime.
- void rime_input (void)
  Send an incoming packet to Rime.
- void rime_driver_send (void)
  Rime calls this function to send out a packet.

9.27 core/net/rime/abc.c File Reference

9.27.1 Detailed Description

Anonymous best-effort local area Broad Cast (abc).

Author:
Adam Dunkels <adam@sics.se>

Definition in file abc.c.

```
#include "contiki-net.h"
#include "net/rime.h"
```

9.28 core/net/rime/abc.h File Reference

9.28.1 Detailed Description

Header file for the Rime module Anonymous BroadCast (abc).

Author:
Adam Dunkels <adam@sics.se>

Definition in file abc.h.

```
#include "net/rime/rimebuf.h"
```

Functions

- void abc_open (struct abc_conn ∗c, u16_t channel, const struct abc_callbacks ∗u)
  Set up an anonymous best-effort broadcast connection.
- void abc_close (struct abc_conn ∗c)
  Close an abc connection.
- int abc_send (struct abc_conn ∗c)
  Send an anonymous best-effort broadcast packet.
- void abc_input_packet (void)
  Internal Rime function: Pass a packet to the abc layer.

9.29 core/net/rime/ctimer.c File Reference

9.29.1 Detailed Description

Callback timer implementation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file ctimer.c.

```
#include "net/rime/ctimer.h"
#include "contiki.h"
#include "lib/list.h"
#include "net/rime.h"
```

9.30 core/net/rime/ctimer.h File Reference

9.30.1 Detailed Description

Header file for the callback timer.

Author:

Adam Dunkels <adam@sics.se>

Definition in file ctimer.h.

```
#include "sys/etimer.h"
```

9.31 core/net/rime/ibc.c File Reference

9.31.1 Detailed Description

Identified best-effort local area broadcast (ibc).

Author:
Adam Dunkels <adam@sics.se>

Definition in file ibc.c.

`#include "contiki-net.h"`

`#include <string.h>`

9.32 core/net/rime/ibc.h File Reference

9.32.1 Detailed Description

Header file for identified best-effort local area broadcast.

Author:
Adam Dunkels <adam@sics.se>

Definition in file ibc.h.

`#include "net/rime/abc.h"`

`#include "net/rime/rimeaddr.h"`

Functions

- void ibc_open (struct ibc_conn ∗c, u16_t channel, const struct ibc_callbacks ∗u)
  Set up an identified best-effort broadcast connection.
- void ibc_close (struct ibc_conn ∗c)
  Close an ibc connection.
- int ibc_send (struct ibc_conn ∗c)
  Send an anonymous best-effort broadcast packet.

9.33 core/net/rime/mesh.c File Reference

9.33.1 Detailed Description

A mesh routing protocol.

Author:
Adam Dunkels <adam@sics.se>

Definition in file mesh.c.

`#include "contiki.h"`

`#include "net/rime.h"`

`#include "net/rime/route.h"`

`#include "net/rime/mesh.h"`

`#include <stddef.h>`

9.34 core/net/rime/mesh.h File Reference

9.34.1 Detailed Description

Header file for the Rime mesh routing protocol.

Author:

Adam Dunkels <adam@sics.se>

Definition in file mesh.h.

`#include "net/rime.h"`

`#include "net/rime/mh.h"`

`#include "net/rime/route-discovery.h"`

Functions

- void mesh_open (struct mesh_conn ∗c, u16_t channels, const struct mesh_callbacks ∗callbacks)

  Open a mesh connection.

- void mesh_close (struct mesh_conn ∗c)

  Close an mesh connection.

- int mesh_send (struct mesh_conn ∗c, rimeaddr_t ∗dest)

  Send a mesh packet.

9.35 core/net/rime/mh.c File Reference

9.35.1 Detailed Description

Multihop forwarding.

Author:

Adam Dunkels <adam@sics.se>

Definition in file mh.c.

`#include "contiki.h"`

`#include "net/rime.h"`

`#include "net/rime/mh.h"`

`#include "net/rime/route.h"`

9.36 core/net/rime/mh.h File Reference

9.36.1 Detailed Description

Multihop forwarding header file.

Author:
Adam Dunkels <adam@sics.se>

Definition in file mh.h.

```
#include "net/rime/abc.h"
#include "net/rime/rimeaddr.h"
```

9.37 core/net/rime/neighbor.c File Reference

9.37.1 Detailed Description

Radio neighborhood management.

Author:
Adam Dunkels <adam@sics.se>

Definition in file neighbor.c.

```
#include <limits.h>
#include <stdio.h>
#include "contiki.h"
#include "net/rime/neighbor.h"
#include "net/rime/ctimer.h"
```

9.38 core/net/rime/neighbor.h File Reference

9.38.1 Detailed Description

Header file for the Contiki radio neighborhood management.

Author:
Adam Dunkels <adam@sics.se>

Definition in file neighbor.h.

```
#include "net/rime/rimeaddr.h"
```

9.39 core/net/rime/nf.c File Reference

9.39.1 Detailed Description

Best-effort network flooding (nf).

Author:
Adam Dunkels <adam@sics.se>

Definition in file nf.c.

```
#include "net/rime/nf.h"
```

```
#include "net/rime.h"
#include "lib/rand.h"
#include <string.h>
#include <stdio.h>
```

9.40 core/net/rime/nf.h File Reference

9.40.1 Detailed Description

Header file for the best-effort network flooding (nf).

Author:
Adam Dunkels <adam@sics.se>

Definition in file nf.h.

```
#include "net/rime/ctimer.h"
#include "net/rime/queuebuf.h"
#include "net/rime/ipolite.h"
```

9.41 core/net/rime/queuebuf.c File Reference

9.41.1 Detailed Description

Implementation of the Rime queue buffers.

Author:
Adam Dunkels <adam@sics.se>

Definition in file queuebuf.c.

```
#include "contiki-net.h"
#include <string.h>
```

9.42 core/net/rime/queuebuf.h File Reference

9.42.1 Detailed Description

Header file for the Rime queue buffer management.

Author:
Adam Dunkels <adam@sics.se>

Definition in file queuebuf.h.

```
#include "net/rime/rimebuf.h"
```

9.43 core/net/rime/rimeaddr.c File Reference

9.43.1 Detailed Description

Functions for manipulating Rime addresses.

Author:

Adam Dunkels <adam@sics.se>

Definition in file rimeaddr.c.

```
#include "net/rime/rimeaddr.h"
```

Variables

- rimeaddr_t rimeaddr_node_addr

  The Rime address of the node.

- const rimeaddr_t rimeaddr_null

  The null Rime address.

9.44 core/net/rime/rimeaddr.h File Reference

9.44.1 Detailed Description

Header file for the Rime address repressentation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file rimeaddr.h.

Functions

- void rimeaddr_copy (rimeaddr_t ∗dest, const rimeaddr_t ∗from)

  Copy a Rime address.

- int rimeaddr_cmp (const rimeaddr_t ∗addr1, const rimeaddr_t ∗addr2)

  Compare two Rime addresses.

- void rimeaddr_set_node_addr (rimeaddr_t ∗addr)

  Set the address of the current node.

Variables

- rimeaddr_t rimeaddr_node_addr

  The Rime address of the node.

- const rimeaddr_t rimeaddr_null

    The null Rime address.

9.45 core/net/rime/rimebuf.c File Reference

9.45.1 Detailed Description

Rime buffer (rimebuf) management.

Author:

Adam Dunkels <adam@sics.se>

Definition in file rimebuf.c.

```
#include <string.h>
#include "contiki-net.h"
#include "net/rime/rimebuf.h"
#include "net/rime.h"
```

9.46 core/net/rime/rimebuf.h File Reference

9.46.1 Detailed Description

Header file for the Rime buffer (rimebuf) management.

Author:

Adam Dunkels <adam@sics.se>

Definition in file rimebuf.h.

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

Defines

- #define RIMEBUF_SIZE 128

    The size of the rimebuf, in bytes.

- #define RIMEBUF_HDR_SIZE 32

    The size of the rimebuf header, in bytes.

Functions

- void rimebuf_clear (void)

    Clear and reset the rimebuf.

- void * rimebuf_dataptr (void)

    Get a pointer to the data in the rimebuf.

- void ∗ rimebuf_hdrptr (void)

  Get a pointer to the header in the rimebuf, for outbound packets.

- u8_t rimebuf_hdrlen (void)

  Get the length of the header in the rimebuf, for outbound packets.

- u16_t rimebuf_datalen (void)

  Get the length of the data in the rimebuf.

- u16_t rimebuf_totlen (void)

  Get the total length of the header and data in the rimebuf.

- void rimebuf_set_datalen (u16_t len)

  Set the length of the data in the rimebuf.

- void rimebuf_reference (void ∗ptr, u16_t len)

  Point the rimebuf to external data.

- int rimebuf_is_reference (void)

  Check if the rimebuf references external data.

- void ∗ rimebuf_reference_ptr (void)

  Get a pointer to external data referenced by the rimebuf.

- void rimebuf_compact (void)

  Compact the rimebuf.

- int rimebuf_copyfrom (u8_t ∗from, u16_t len)

  Copy from external data into the rimebuf.

- int rimebuf_copyto (u8_t ∗to)

  Copy the entire rimebuf to an external buffer.

- int rimebuf_copyto_hdr (u8_t ∗to)

  Copy the header portion of the rimebuf to an external buffer.

- int rimebuf_hdralloc (int size)

  Extend the header of the rimebuf, for outbound packets.

- int rimebuf_hdrreduce (int size)

  Reduce the header in the rimebuf, for incoming packets.

9.47 core/net/rime/route-discovery.c File Reference

9.47.1 Detailed Description

Route discovery protocol.

Author:
Adam Dunkels <adam@sics.se>

Definition in file route-discovery.c.

```
#include "contiki.h"
#include "net/rime.h"
#include "net/rime/route.h"
#include "net/rime/route-discovery.h"
#include <stddef.h>
```

9.48 core/net/rime/route-discovery.h File Reference

9.48.1 Detailed Description

Header file for the Rime mesh routing protocol.

Author:
Adam Dunkels <adam@sics.se>

Definition in file route-discovery.h.

```
#include "net/rime.h"
#include "net/rime/nf.h"
```

9.49 core/net/rime/route.c File Reference

9.49.1 Detailed Description

Rime route table.

Author:
Adam Dunkels <adam@sics.se>

Definition in file route.c.

```
#include <stdio.h>
#include "net/rime/route.h"
```

9.50 core/net/rime/route.h File Reference

9.50.1 Detailed Description

Header file for the Rime route table.

Author:
Adam Dunkels <adam@sics.se>

Definition in file route.h.

```
#include "contiki-net.h"
#include "net/rime.h"
```

9.51 core/net/rime/ruc.c File Reference

9.51.1 Detailed Description

Reliable unicast.

Author:

Adam Dunkels <adam@sics.se>

Definition in file ruc.c.

```
#include "net/rime/ruc.h"
#include "net/rime/neighbor.h"
#include "net/rime.h"
#include <string.h>
```

9.52 core/net/rime/ruc.h File Reference

9.52.1 Detailed Description

Reliable unicast header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file ruc.h.

```
#include "net/rime/suc.h"
```

9.53 core/net/rime/rudolph0.c File Reference

9.53.1 Detailed Description

Rudolph0: a simple block data flooding protocol.

Author:

Adam Dunkels <adam@sics.se>

Definition in file rudolph0.c.

```
#include <stddef.h>
#include "net/rime.h"
#include "net/rime/rudolph0.h"
#include <stdio.h>
```

9.54 core/net/rime/rudolph0.h File Reference

9.54.1 Detailed Description

Header file for the single-hop reliable bulk data transfer module.

Author:
Adam Dunkels <adam@sics.se>

Definition in file rudolph0.h.

```
#include "net/rime.h"
#include "net/rime/sabc.h"
#include "net/rime/polite.h"
#include "contiki-net.h"
```

9.55 core/net/rime/rudolph1.c File Reference

9.55.1 Detailed Description

Rudolph1: a simple block data flooding protocol.

Author:
Adam Dunkels <adam@sics.se>

Definition in file rudolph1.c.

```
#include <stdio.h>
#include <stddef.h>
#include "net/rime.h"
#include "net/rime/rudolph1.h"
#include "cfs/cfs.h"
```

9.56 core/net/rime/rudolph1.h File Reference

9.56.1 Detailed Description

Header file for the multi-hop reliable bulk data transfer mechanism.

Author:
Adam Dunkels <adam@sics.se>

Definition in file rudolph1.h.

```
#include "net/rime.h"
#include "net/rime/trickle.h"
#include "net/rime/uabc.h"
#include "contiki-net.h"
```

9.57 core/net/rime/sabc.c File Reference

9.57.1 Detailed Description

Implementation of the Rime module Stubborn Anonymous BroadCast (sabc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file sabc.c.

```
#include "net/rime/sabc.h"
#include "net/rime.h"
#include <string.h>
```

9.58 core/net/rime/sabc.h File Reference

9.58.1 Detailed Description

Header file for the Rime module Stubborn Anonymous BroadCast (sabc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file sabc.h.

```
#include "net/rime/uc.h"
#include "net/rime/ctimer.h"
#include "net/rime/queuebuf.h"
```

Functions

- void sabc_open (struct sabc_conn ∗c, u16_t channel, const struct sabc_callbacks ∗u)

  Set up a sabc connection.
- int sabc_send_stubborn (struct sabc_conn ∗c, clock_time_t t)

  Send a stubborn message.
- void sabc_cancel (struct sabc_conn ∗c)

  Cancel the current stubborn message.
- void sabc_set_timer (struct sabc_conn ∗c, clock_time_t t)

  Set the retransmission time of the current stubborn message.

9.59 core/net/rime/sibc.c File Reference

9.59.1 Detailed Description

Implementation of the Rime module Stubborn Identified BroadCast (sibc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file sibc.c.

```
#include "net/rime/sibc.h"
#include "net/rime.h"
#include <string.h>
```

9.60 core/net/rime/sibc.h File Reference

9.60.1 Detailed Description

Header file for the Rime module Stubborn Identified BroadCast (sibc).

Author:
Adam Dunkels <adam@sics.se>

Definition in file sibc.h.

```
#include "net/rime/uc.h"
#include "net/rime/ctimer.h"
#include "net/rime/queuebuf.h"
```

9.61 core/net/rime/suc.c File Reference

9.61.1 Detailed Description

Stubborn unicast.

Author:
Adam Dunkels <adam@sics.se>

Definition in file suc.c.

```
#include "net/rime/suc.h"
#include "net/rime.h"
#include <string.h>
```

9.62 core/net/rime/suc.h File Reference

9.62.1 Detailed Description

Stubborn unicast header file.

Author:
Adam Dunkels <adam@sics.se>

Definition in file suc.h.

```
#include "net/rime/uc.h"
#include "net/rime/ctimer.h"
#include "net/rime/queuebuf.h"
```

9.63 core/net/rime/tree.c File Reference

9.63.1 Detailed Description

Tree-based hop-by-hop reliable data collection.

Author:

Adam Dunkels <adam@sics.se>

Definition in file tree.c.

```
#include "contiki.h"
#include "net/rime.h"
#include "net/rime/neighbor.h"
#include "net/rime/nf.h"
#include "net/rime/tree.h"
#include "dev/radio-sensor.h"
#include <string.h>
#include <stdio.h>
#include <stddef.h>
```

9.64 core/net/rime/tree.h File Reference

9.64.1 Detailed Description

Header file for hop-by-hop reliable data collection.

Author:

Adam Dunkels <adam@sics.se>

Definition in file tree.h.

```
#include "net/rime/ipolite.h"
#include "net/rime/ruc.h"
```

9.65 core/net/rime/trickle.c File Reference

9.65.1 Detailed Description

Trickle (reliable single source flooding) for Rime.

Author:

Adam Dunkels <adam@sics.se>

Definition in file trickle.c.

```
#include "net/rime/trickle.h"
```

9.66 core/net/rime/trickle.h File Reference

9.66.1 Detailed Description

Header file for Trickle (reliable single source flooding) for Rime.

Author:

Adam Dunkels <adam@sics.se>

Definition in file trickle.h.

```
#include "net/rime.h"
#include "net/rime/nf.h"
```

9.67 core/net/rime/uabc.c File Reference

9.67.1 Detailed Description

Unique Anonymous best effort local area BroadCast (uabc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file uabc.c.

```
#include "net/rime.h"
#include "net/rime/uabc.h"
#include "lib/rand.h"
#include <string.h>
```

9.68 core/net/rime/uabc.h File Reference

9.68.1 Detailed Description

Header file for Unique Anonymous best effort local area BroadCast (uabc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file uabc.h.

```
#include "net/rime.h"
```

9.69 core/net/rime/uc.c File Reference

9.69.1 Detailed Description

Single-hop unicast.

Author:

Adam Dunkels <adam@sics.se>

Definition in file uc.c.

```
#include "net/rime.h"
#include "net/rime/uc.h"
#include <string.h>
```

9.70 core/net/rime/uc.h File Reference

9.70.1 Detailed Description

Header file for Rime's single-hop unicast.

Author:

Adam Dunkels <adam@sics.se>

Definition in file uc.h.

```
#include "net/rime/ibc.h"
```

9.71 core/net/rime/uibc.c File Reference

9.71.1 Detailed Description

Unique Identified best effort local area BroadCast (uibc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file uibc.c.

```
#include "net/rime.h"
#include "net/rime/uibc.h"
#include "lib/rand.h"
#include <string.h>
```

9.72 core/net/rime/uibc.h File Reference

9.72.1 Detailed Description

Header file for Unique Identified best effort local area BroadCast (uibc).

Author:

Adam Dunkels <adam@sics.se>

Definition in file uibc.h.

```
#include "net/rime.h"
```

9.73 core/net/tcpip.h File Reference

9.73.1 Detailed Description

Header for the Contiki/uIP interface.

Author:

Adam Dunkels <adam@sics.se>

Definition in file tcpip.h.

```
#include "contiki.h"
#include "net/uip.h"
```

TCP functions

- CCIF void tcp_attach (struct uip_conn ∗conn, void ∗appstate)

  Attach a TCP connection to the current process.

- CCIF void tcp_listen (u16_t port)

  Open a TCP port.

- CCIF void tcp_unlisten (u16_t port)

  Close a listening TCP port.

- CCIF struct uip_conn ∗ tcp_connect (uip_ipaddr_t ∗ripaddr, u16_t port, void ∗appstate)

  Open a TCP connection to the specified IP address and port.

- void tcpip_poll_tcp (struct uip_conn ∗conn)

  Cause a specified TCP connection to be polled.

UDP functions

- #define udp_bind(conn, port) uip_udp_bind(conn, port)

  Bind a UDP connection to a local port.

- void udp_attach (struct uip_udp_conn ∗conn, void ∗appstate)

  Attach the current process to a UDP connection.

- CCIF struct uip_udp_conn ∗ udp_new (const uip_ipaddr_t ∗ripaddr, u16_t port, void ∗appstate)

  Create a new UDP connection.

- uip_udp_conn ∗ udp_broadcast_new (u16_t port, void ∗appstate)

  Create a new UDP broadcast connection.

- CCIF void tcpip_poll_udp (struct uip_udp_conn ∗conn)

  Cause a specified UDP connection to be polled.

TCP/IP packet processing

- CCIF void tcpip_input (void)

  Deliver an incoming packet to the TCP/IP stack.

Defines

- #define UIP_APPCALL tcpip_uipcall

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

Typedefs

- typedef tcpip_uipstate uip_udp_appstate_t

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

- typedef tcpip_uipstate uip_tcp_appstate_t

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

Variables

- CCIF process_event_t tcpip_event

  The uIP event.

9.74 core/net/uip-fw.c File Reference

9.74.1 Detailed Description

uIP packet forwarding.

Author:

Adam Dunkels <adam@sics.se>

This file implements a number of simple functions which do packet forwarding over multiple network interfaces with uIP.

Definition in file uip-fw.c.

```
#include <string.h>
#include "contiki-conf.h"
#include "net/uip.h"
#include "net/uip_arch.h"
#include "net/uip-fw.h"
```

9.75 core/net/uip-fw.h File Reference

9.75.1 Detailed Description

uIP packet forwarding header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file uip-fw.h.

```
#include "net/uip.h"
```

Defines

- #define UIP_FW_NETIF(ip1, ip2, ip3, ip4, nm1, nm2, nm3, nm4, outputfunc)

  Intantiating macro for a uIP network interface.

- #define uip_fw_setipaddr(netif, addr)

  Set the IP address of a network interface.

- #define uip_fw_setnetmask(netif, addr)

  Set the netmask of a network interface.

- #define UIP_FW_LOCAL

  A non-error message that indicates that a packet should be processed locally.

- #define UIP_FW_OK

  A non-error message that indicates that something went OK.

- #define UIP_FW_FORWARDED

  A non-error message that indicates that a packet was forwarded.

- #define UIP_FW_ZEROLEN

  A non-error message that indicates that a zero-length packet transmission was attempted, and that no packet was sent.

- #define UIP_FW_TOOLARGE

  An error message that indicates that a packet that was too large for the outbound network interface was detected.

- #define UIP_FW_NOROUTE

  An error message that indicates that no suitable interface could be found for an outbound packet.

- #define UIP_FW_DROPPED

  An error message that indicates that a packet that should be forwarded or output was dropped.

Functions

- void uip_fw_init (void)
  Initialize the uIP packet forwarding module.
- u8_t uip_fw_forward (void)
  Forward an IP packet in the uip_buf buffer.
- u8_t uip_fw_output (void)
  Output an IP packet on the correct network interface.
- void uip_fw_register (struct uip_fw_netif ∗netif)
  Register a network interface with the forwarding module.
- void uip_fw_default (struct uip_fw_netif ∗netif)
  Register a default network interface.
- void uip_fw_periodic (void)
  Perform periodic processing.

9.76 core/net/uip-split.h File Reference

9.76.1 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.

9.77 core/net/uip.c File Reference

9.77.1 Detailed Description

The uIP TCP/IP stack code.

Author:
Adam Dunkels <adam@dunkels.com>

Definition in file uip.c.

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

Functions

- void uip_setipid (u16_t id)
  uIP initialization function.
- void uip_init (void)
  uIP initialization function.
- uip_udp_conn ∗ uip_udp_new (const 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

- 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_udp_conn ∗ uip_udp_conn
  The current UDP connection.

- u8_t uip_acc32 [4]

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

9.78 core/net/uip.h File Reference

9.78.1 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 "net/uipopt.h"
#include "net/tcpip.h"
```

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_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_datalen()

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

- #define uip_urgdatalen()

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

- #define uip_close()

  Close the current connection.

- #define uip_abort()

  Abort the current connection.

- #define uip_stop()

  Tell the sending host to stop sending data.

- #define uip_stopped(conn)

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

- #define uip_restart()

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

- #define uip_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.

- #define uip_ipaddr_to_quad(a)

    Convert an IP address to four bytes separated by commas.

- #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 UIP_APPDATA_SIZE
  The buffer size available for user data in the uip_buf buffer.

Typedefs

- typedef uip_ip4addr_t uip_ip4addr_t
  Representation of an IP address.

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.
- CCIF void uip_send (const void ∗data, int len)
  Send data on the current connection.
- uip_udp_conn ∗ uip_udp_new (const uip_ipaddr_t ∗ripaddr, u16_t rport)
  Set up a new UDP connection.
- CCIF u16_t htons (u16_t val)
  Convert 16-bit quantity from host byte order to network byte order.
- 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

- CCIF u8_t uip_buf [UIP_BUFSIZE+2]

  The uIP packet buffer.

- CCIF void ∗ uip_appdata

  Pointer to the application data in the packet buffer.

- CCIF u16_t uip_len

  The length of the packet in the uip_buf buffer.

- CCIF struct uip_conn ∗ uip_conn

  Pointer to the current TCP connection.

- 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_stats uip_stat

  The uIP TCP/IP statistics.

9.79 core/net/uip_arp.c File Reference

9.79.1 Detailed Description

Implementation of the ARP Address Resolution Protocol.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file uip_arp.c.

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

9.80 core/net/uip_arp.h File Reference

9.80.1 Detailed Description

Macros and definitions for the ARP module.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file uip_arp.h.

```
#include "net/uip.h"
```

Defines

- #define uip_setethaddr(eaddr)

  Specifiy 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.

9.81 core/net/uiplib.h File Reference

9.81.1 Detailed Description

Various uIP library functions.

Author:

Adam Dunkels <adam@sics.se>

Definition in file uiplib.h.

Functions

- CCIF unsigned char uiplib_ipaddrconv (char *addrstr, unsigned char *addr)

  Convert a textual representation of an IP address to a numerical representation.

9.82 core/net/uipopt.h File Reference

9.82.1 Detailed Description

Configuration options for uIP.

Author:

Adam Dunkels <adam@dunkels.com>

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

Definition in file uipopt.h.

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

Defines

- #define UIP_FIXEDADDR

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

- #define UIP_PINGADDRCONF

  Ping IP address asignment.

- #define UIP_FIXEDETHADDR

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

- #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.

- #define UIP_UDP

  Toggles wether UDP support should be compiled in or not.

- #define UIP_UDP_CHECKSUMS

  Toggles if UDP checksums should be used or not.

- #define UIP_UDP_CONNS

  The maximum amount of concurrent UDP connections.

- #define UIP_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.

- #define UIP_ARPTAB_SIZE

  The size of the ARP table.

- #define UIP_ARP_MAXAGE 120

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

- #define UIP_BUFSIZE

  The size of the uIP packet buffer.

- #define UIP_STATISTICS

  Determines if statistics support should be compiled in.

- #define UIP_LOGGING

  Determines if logging of certain events should be compiled in.

- #define UIP_BROADCAST

  Broadcast support.

- #define UIP_LLH_LEN

  The link level header length.

- #define UIP_BYTE_ORDER

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

Functions

- void uip_log (char *msg)

  Print out a uIP log message.

9.83 core/sys/arg.c File Reference

9.83.1 Detailed Description

Argument buffer for passing arguments when starting processes.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file arg.c.

```
#include "contiki.h"
#include "sys/arg.h"
```

Functions

- char * arg_alloc (char size)

  Allocates an argument buffer.

- void arg_free (char *arg)

  Deallocates an argument buffer.

9.84 core/sys/cc.h File Reference

9.84.1 Detailed Description

Default definitions of C compiler quirk work-arounds.

Author:

Adam Dunkels <adam@dunkels.com>

This file is used for making use of extra functionality of some C compilers used for Contiki, and defining work-arounds for various quirks and problems with some other C compilers.

Definition in file cc.h.

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

Defines

- #define CC_REGISTER_ARG

  Configure if the C compiler supports the "register" keyword for function arguments.

- #define CC_FUNCTION_POINTER_ARGS 0

  Configure if the C compiler supports the arguments for function pointers.

- #define CC_FASTCALL

  Configure if the C compiler supports fastcall function declarations.

- #define CC_UNSIGNED_CHAR_BUGS 0

  Configure work-around for unsigned char bugs with sdcc.

- #define CC_DOUBLE_HASH 0

  Configure if C compiler supports double hash marks in C macros.

9.85 core/sys/dsc.h File Reference

9.85.1 Detailed Description

Declaration of the DSC program description structure.

Author:

Adam Dunkels <adam@dunkels.com>

Definition in file dsc.h.

```
#include "ctk/ctk.h"
```

Defines

- #define DSC(dscname, description, prgname, process, icon) CLIF const struct dsc dscname = {description, prgname, icon}

  Intantiating macro for the DSC structure.

9.86 core/sys/etimer.c File Reference

9.86.1 Detailed Description

Event timer library implementation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file etimer.c.

```
#include "contiki-conf.h"
#include "sys/etimer.h"
#include "sys/process.h"
```

Functions called from timer interrupts, by the system

- void etimer_request_poll (void)
  Make the event timer aware that the clock has changed.
- int etimer_pending (void)
  Check if there are any non-expired event timers.
- clock_time_t etimer_next_expiration_time (void)
  Get next event timer expiration time.

Functions called from application programs

- void etimer_set (struct etimer ∗et, clock_time_t interval)
  Set an event timer.
- void etimer_reset (struct etimer ∗et)
  Reset an event timer with the same interval as was previously set.
- void etimer_restart (struct etimer ∗et)
  Restart an event timer from the current point in time.
- void etimer_adjust (struct etimer ∗et, int timediff)
  Adjust the expiration time for an event timer.
- int etimer_expired (struct etimer ∗et)
  Check if an event timer has expired.
- clock_time_t etimer_expiration_time (struct etimer ∗et)
  Get the expiration time for the event timer.
- clock_time_t etimer_start_time (struct etimer ∗et)
  Get the start time for the event timer.
- void etimer_stop (struct etimer ∗et)
  Stop a pending event timer.

9.87 core/sys/etimer.h File Reference

9.87.1 Detailed Description

Event timer header file.

Author:
Adam Dunkels <adam@sics.se>

Definition in file etimer.h.

```
#include "sys/timer.h"
#include "sys/process.h"
```

9.88 core/sys/lc-addrlabels.h File Reference

9.88.1 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 dividuum for finding the nice local scope label implementation.

Definition in file lc-addrlabels.h.

9.89 core/sys/lc-switch.h File Reference

9.89.1 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.

Typedefs

- typedef unsigned short lc_t

  The local continuation type.

9.90 core/sys/lc.h File Reference

9.90.1 Detailed Description

Local continuations.

Author:

Adam Dunkels <adam@sics.se>

Definition in file lc.h.

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

Defines

- #define LC_INIT(lc)
  Initialize a local continuation.
- #define LC_SET(lc)
  Set a local continuation.
- #define LC_RESUME(lc)
  Resume a local continuation.
- #define LC_END(lc)
  Mark the end of local continuation usage.

9.91 core/sys/loader.h File Reference

9.91.1 Detailed Description

Default definitions and error values for the Contiki program loader.

Author:
Adam Dunkels <adam@dunkels.com>

Definition in file loader.h.

Defines

- #define LOADER_OK 0
  No error.
- #define LOADER_ERR_READ 1
  Read error.
- #define LOADER_ERR_HDR 2
  Header error.
- #define LOADER_ERR_OS 3
  Wrong OS.
- #define LOADER_ERR_FMT 4
  Data format error.
- #define LOADER_ERR_MEM 5
  Not enough memory.
- #define LOADER_ERR_OPEN 6
  Could not open file.

- #define LOADER_ERR_ARCH 7

  Wrong architecture.

- #define LOADER_ERR_VERSION 8

  Wrong OS version.

- #define LOADER_ERR_NOLOADER 9

  Program loading not supported.

- #define LOADER_LOAD(name, arg) LOADER_ERR_NOLOADER

  Load and execute a program.

- #define LOADER_UNLOAD()

  Unload a program from memory.

- #define LOADER_LOAD_DSC(name) NULL

  Load a DSC (program description).

- #define LOADER_UNLOAD_DSC(dsc)

  Unload a DSC (program description).

9.92 core/sys/mt.c File Reference

9.92.1 Detailed Description

Implementation of the archtecture agnostic parts of the preemptive multithreading library for Contiki.

Author:

Adam Dunkels <adam@sics.se>

Definition in file mt.c.

```
#include "contiki.h"
#include "sys/mt.h"
#include "sys/cc.h"
```

Functions

- void mt_init (void)

  Initializes the multithreading library.

- void mt_remove (void)

  Uninstalls library and cleans up.

- void mt_start (struct mt_thread *thread, void(*function)(void *), void *data)

  Starts a multithreading thread.

- void mt_exec (struct mt_thread *thread)

Execute parts of a thread.

- void mt_yield (void)

  Voluntarily give up the processor.

- void mt_exit (void)

  Exit a thread.

- void mt_stop (struct mt_thread ∗thread)

  Stop a thread.

9.93 core/sys/mt.h File Reference

9.93.1 Detailed Description

Header file for the preemptive multitasking library for Contiki.

Author:

Adam Dunkels <adam@sics.se>

Definition in file mt.h.

```
#include "contiki.h"
#include "mtarch.h"
```

Defines

- #define MT_OK

  No error.

Functions

- void mtarch_init (void)

  Initialize the architecture specific support functions for the multi-thread library.

- void mtarch_remove (void)

  Uninstall library and clean up.

- void mtarch_start (struct mtarch_thread ∗thread, void(∗function)(void ∗data), void ∗data)

  Setup the stack frame for a thread that is being started.

- void mtarch_exec (struct mtarch_thread ∗thread)

  Start executing a thread.

- void mtarch_yield (void)

  Yield the processor.

- void mtarch_stop (struct mtarch_thread ∗thread)
  Clean up the stack of a thread.
- void mt_init (void)
  Initializes the multithreading library.
- void mt_remove (void)
  Uninstalls library and cleans up.
- void mt_start (struct mt_thread ∗thread, void(∗function)(void ∗), void ∗data)
  Starts a multithreading thread.
- void mt_exec (struct mt_thread ∗thread)
  Execute parts of a thread.
- void mt_yield (void)
  Voluntarily give up the processor.
- void mt_exit (void)
  Exit a thread.
- void mt_stop (struct mt_thread ∗thread)
  Stop a thread.

9.94 core/sys/process.c File Reference

9.94.1 Detailed Description

Implementation of the Contiki process kernel.

Author:
Adam Dunkels <adam@sics.se>

Definition in file process.c.

```
#include <stdio.h>
#include "sys/process.h"
#include "sys/arg.h"
```

Functions called from application programs

- process_event_t process_alloc_event (void)
  Allocate a global event number.
- void process_start (struct process ∗p, char ∗arg)
  Start a process.
- void process_exit (struct process ∗p)

Cause a process to exit.

- int process_post (struct process ∗p, process_event_t ev, process_data_t data)

  Post an asynchronous event.

- void process_post_synch (struct process ∗p, process_event_t ev, process_data_t data)

  Post a synchronous event to a process.

Functions called by the system and boot-up code

- void process_init (void)

  Initialize the process module.

- int process_run (void)

  Run the system once - call poll handlers and process one event.

- int process_nevents (void)

  Number of events waiting to be processed.

Functions called from device drivers

- void process_poll (struct process ∗p)

  Request a process to be polled.

9.95 core/sys/process.h File Reference

9.95.1 Detailed Description

Header file for the Contiki process interface.

Author:

Adam Dunkels <adam@sics.se>

Definition in file process.h.

```
#include "sys/pt.h"
#include "sys/cc.h"
```

Return values

- #define PROCESS_ERR_OK 0

  Return value indicating that an operation was successful.

- #define PROCESS_ERR_FULL 1

  Return value indicating that the event queue was full.

Process protothread functions

- #define PROCESS_BEGIN()
  Define the beginning of a process.
- #define PROCESS_END()
  Define the end of a process.
- #define PROCESS_WAIT_EVENT()
  Wait for an event to be posted to the process.
- #define PROCESS_WAIT_EVENT_UNTIL(c)
  Wait for an event to be posted to the process, with an extra condition.
- #define PROCESS_YIELD()
  Yield the currently running process.
- #define PROCESS_YIELD_UNTIL(c)
  Yield the currently running process until a condition occurs.
- #define PROCESS_WAIT_UNTIL(c)
  Wait for a condition to occur.
- #define PROCESS_EXIT()
  Exit the currently running process.
- #define PROCESS_PT_SPAWN(pt, thread)
  Spawn a protothread from the process.
- #define PROCESS_PAUSE()
  Yield the process for a short while.

Poll and exit handlers

- #define PROCESS_POLLHANDLER(handler)
  Specify an action when a process is polled.
- #define PROCESS_EXITHANDLER(handler)
  Specify an action when a process exits.

Process declaration and definion

- #define PROCESS_THREAD(name, ev, data)
  Define the body of a process.
- #define PROCESS_NAME(name)
  Declare the name of a process.

- #define PROCESS(name, strname)

  Declare a process.

Functions called from application programs

- #define PROCESS_CURRENT()

  Get a pointer to the currently running process.

- #define PROCESS_CONTEXT_BEGIN(p)

  Switch context to another process.

- #define PROCESS_CONTEXT_END(p) process_current = tmp_current; }

  End a context switch.

9.96 core/sys/pt-sem.h File Reference

9.96.1 Detailed Description

Counting semaphores implemented on protothreads.

Author:

Adam Dunkels <adam@sics.se>

Definition in file pt-sem.h.

`#include "sys/pt.h"`

Defines

- #define PT_SEM_INIT(s, c)

  Initialize a semaphore.

- #define PT_SEM_WAIT(pt, s)

  Wait for a semaphore.

- #define PT_SEM_SIGNAL(pt, s)

  Signal a semaphore.

9.97 core/sys/pt.h File Reference

9.97.1 Detailed Description

Protothreads implementation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file pt.h.

```
#include "sys/lc.h"
```

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.

9.98 core/sys/timer.c File Reference

9.98.1 Detailed Description

Timer library implementation.

Author:

Adam Dunkels <adam@sics.se>

Definition in file timer.c.

```
#include "contiki-conf.h"
#include "sys/clock.h"
#include "sys/timer.h"
```

9.99 core/sys/timer.h File Reference

9.99.1 Detailed Description

Timer library header file.

Author:

Adam Dunkels <adam@sics.se>

Definition in file timer.h.

```
#include "sys/clock.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.

9.100 platform/esb/dev/beep.h File Reference

9.100.1 Detailed Description

Interface to the beeper.

Author:

Adam Dunkels <adam@sics.se>

Definition in file beep.h.

```
#include "sys/clock.h"
```

Functions

- void beep_beep (int len)

  Beep for a specified time.

- void beep_alarm (int alarmmode, int len)

  Beep an alarm for a specified time.

- void beep (void)

  Produces a quick click-like beep.

- void beep_down (int len)

  A beep with a pitch-bend down.

- void beep_on (void)

  Turn the beeper on.

- void beep_off (void)

  Turn the beeper off.

- void beep_spinup (void)

  Produce a sound similar to a hard-drive spinup.

- void beep_long (clock_time_t len)

  Beep for a long time (seconds).

9.101 platform/esb/dev/eeprom.c File Reference

9.101.1 Detailed Description

EEPROM functions.

Author:
Adam Dunkels <adam@sics.se>

Definition in file eeprom.c.

```
#include <msp430x14x.h>
#include <io.h>
#include "dev/eeprom.h"
```

Defines

- #define SDA_HIGH (P5OUT |= 0x04)
  EEPROM data line high.
- #define SDA_LOW (P5OUT &= 0xFB)
  EEPROM data line low.
- #define SCL_HIGH (P5OUT |= 0x08)
  EEPROM clock line high.
- #define SCL_LOW (P5OUT &= 0xF7)
  EEPROM clock line low.

Functions

- void eeprom_read (unsigned short addr, unsigned char *buf, int size)
  Read bytes from the EEPROM using sequential read.
- void eeprom_write (unsigned short addr, unsigned char *buf, int size)
  Write bytes to EEPROM using sequencial write.

9.102 platform/esb/dev/rs232.c File Reference

9.102.1 Detailed Description

RS232 communication device driver for the MSP430.

Author:
Adam Dunkels <adam@sics.se>

This file contains an RS232 device driver for the MSP430 microcontroller.

Definition in file rs232.c.

```
#include <io.h>
#include <signal.h>
#include <string.h>
#include "contiki-esb.h"
```

9.103 platform/esb/dev/rs232.h File Reference

9.103.1 Detailed Description

Header file for MSP430 RS232 driver.

Author:

Adam Dunkels <adam@sics.se>

Definition in file rs232.h.

Functions

- void rs232_init (void)

  Initialize the RS232 module.

- void rs232_set_input (int(∗f)(unsigned char))

  Set an input handler for incoming RS232 data.

- void rs232_set_speed (unsigned char speed)

  Configure the speed of the RS232 hardware.

- void rs232_print (char ∗str)

  Print a text string on RS232.

- void rs232_send (char c)

  Print a character on RS232.

9.104 platform/esb/dev/tr1001.c File Reference

9.104.1 Detailed Description

Device driver and packet framing for the RFM-TR1001 radio module.

Author:

Adam Dunkels <adam@sics.se>

This file implements a device driver for the RFM-TR1001 radio tranciever.

Definition in file tr1001.c.

```
#include "contiki-esb.h"
#include "lib/me.h"
#include "lib/crc16.h"
#include "net/tr1001-drv.h"
#include <io.h>
#include <signal.h>
#include <string.h>
```

10 Contiki 2.x Example Documentation

10.1 code-style.c

```
/**
 * \defgroup coding-style Coding style
 *
 * This is how a Doxygen module is documented - start with a \defgroup
 * Doxygen keyword at the beginning of the file to define a module,
 * and use the \addtogroup Doxygen keyword in all other files that
 * belong to the same module. Typically, the \defgroup is placed in
 * the .h file and \addtogroup in the .c file.
 *
 * @{
 */

/**
 * \file
 *         A brief description of what this file is.
 * \author
 *         Adam Dunkels <adam@sics.se>
 *
 *         Every file that is part of a documented module has to have
 *         a \file block, else it will not show up in the Doxygen
 *         "Modules" * section.
 */

/* Single line comments look like this. */

/*
 * Multi-line comments look like this. Comments should prefferably be
 * full sentences, filled to look like real paragraphs.
 */

#include "contiki.h"

/*
 * Make sure that non-global variables are all maked with the static
 * keyword. This keeps the size of the symbol table down.
 */
static int flag;

/*
 * All variables and functions that are visible outside of the file
 * should have the module name prepended to them. This makes it easy
 * to know where to look for function and variable definitions.
 *
 * Put dividers (a single-line comment consisting only of dashes)
 * between functions.
 */
/*---------------------------------------------------------------------------*/
```

```
/**
 * \brief      Use Doxygen documentation for functions.
 * \param c    Briefly describe all parameters.
 * \return     Briefly describe the return value.
 * \retval 0   Functions that return a few specified values
 * \retval 1   can use the \retval keyword instead of \return.
 *
 *             Put a longer description of what the function does
 *             after the preamble of Doxygen keywords.
 *
 *             This template should always be used to document
 *             functions. The text following the introduction is used
 *             as the function’s documentation.
 *
 *             Function prototypes have the return type on one line,
 *             the name and arguments on one line (with no space
 *             between the name and the first parenthesis), followed
 *             by a single curly bracket on its own line.
 */
void
code_style_example_function(void)
{
  /*
   * Local variables should always be declared at the start of the
   * function.
   */
  int i;                   /* Use short variable names for loop
                              counters. */

  /*
   * There should be no space between keywords and the first
   * parenthesis. There should be spaces around binary operators, no
   * spaces between a unary operator and its operand.
   *
   * Curly brackets following for(), if(), do, and case() statements
   * should follow the statement on the same line.
   */
  for(i = 0; i < 10; ++i) {
    /*
     * Always use full blocks (curly brackets) after if(), for(), and
     * while() statements, even though the statement is a single line
     * of code. This makes the code easier to read and modifications
     * are less error prone.
     */
    if(i == c) {
      return c;            /* No parentesis around return values. */
    } else {               /* The else keyword is placed inbetween
                              curly brackers, always on its own line. */
      c++;
    }
  }
}
/*---------------------------------------------------------------------------*/
/*
 * Static (non-global) functions do not need Doxygen comments. The
 * name should not be prepended with the module name - doing so would
 * create confusion.
 */
static void
an_example_function(void)
{

}
/*---------------------------------------------------------------------------*/

/* The following stuff ends the \defgroup block at the beginning of
   the file: */
```

```
/** @} */
```

10.2 example-list.c

```
#include "list.h"

struct example_list_struct {
  struct *next;
  int number;
};

LIST(example_list);

void
example_function(void)
{
  struct example_list_struct *s;
  struct example_list_struct element1, element2;

  list_init(example_list);

  list_add(example_list, &element1);
  list_add(example_list, &element2);

  for(s = list_head(example_list);
      s != NULL;
      s = s->next) {
    printf("List element number %d\n", s->number);
  }

}
```

10.3 example-packet-drv.c

```
/*
 * This is an example of how to write a network device driver ("packet
 * driver") for Contiki. A packet driver is a regular Contiki process
 * that does two things:
 * # Checks for incoming packets and delivers those to the TCP/IP stack
 * # Provides an output function that transmits packets
 *
 * The output function is registered with the Contiki TCP/IP stack,
 * whereas incoming packets must be checked inside a Contiki process.
 * We use the same process for checking for incoming packets and for
 * registering the output function.
 */

/*
 * We include the "contiki-net.h" file to get all the network functions.
 */
#include "contiki-net.h"

/*---------------------------------------------------------------------------*/
/*
 * We declare the process that we use to register with the TCP/IP stack,
 * and to check for incoming packets.
 */
PROCESS(example_packet_driver_process, "Example packet driver process");
/*---------------------------------------------------------------------------*/
/*
 * Next, we define the function that transmits packets. This function
 * is called from the TCP/IP stack when a packet is to be transmitted.
 * The packet is located in the uip_buf[] buffer, and the length of the
```

```
 * packet is in the uip_len variable.
 */
u8_t
example_packet_driver_output(void)
{
  let_the_hardware_send_the_packet(uip_buf, uip_len);

  /*
   * A network device driver returns always zero.
   */
  return 0;
}
/*---------------------------------------------------------------------------*/
/*
 * This is the poll handler function in the process below. This poll
 * handler function checks for incoming packets and delivers them to
 * the TCP/IP stack.
 */
static void
pollhandler(void)
{
  /*
   * We assume that we have some hardware device that notifies us when
   * a new packet has arrived. We also assume that we have a function
   * that pulls out the new packet (here called
   * check_and_copy_packet()) and puts it in the uip_buf[] buffer. The
   * function returns the length of the incoming packet, and we store
   * it in the global uip_len variable. If the packet is longer than
   * zero bytes, we hand it over to the TCP/IP stack.
   */
  uip_len = check_and_copy_packet();

  /*
   * The function tcpip_input() delivers the packet in the uip_buf[]
   * buffer to the TCP/IP stack.
   */
  if(uip_len > 0) {
    tcpip_input();
  }

  /*
   * Now we'll make sure that the poll handler is executed repeatedly.
   * We do this by calling process_poll() with this process as its
   * argument.
   *
   * In many cases, the hardware will cause an interrupt to be executed
   * when a new packet arrives. For such hardware devices, the interrupt
   * handler calls process_poll() (which is safe to use in an interrupt
   * context) instead.
   */
  process_poll(&example_packet_driver_process);
}
/*---------------------------------------------------------------------------*/
/*
 * Finally, we define the process that does the work.
 */
PROCESS_THREAD(example_packet_driver_process, ev, data)
{
  /*
   * This process has a poll handler, so we declare it here. Note that
   * the PROCESS_POLLHANDLER() macro must come before the PROCESS_BEGIN()
   * macro.
   */
  PROCESS_POLLHANDLER(pollhandler());

  /*
   * This process has an exit handler, so we declare it here. Note that
```

```
   * the PROCESS_EXITHANDLER() macro must come before the PROCESS_BEGIN()
   * macro.
   */
  PROCESS_EXITHANDLER(exithandler());

  /*
   * The process begins here.
   */
  PROCESS_BEGIN();

  /*
   * We start with initializing the hardware.
   */
  initialize_the_hardware();

  /*
   * Register the driver. This will cause any previously registered driver
   * to be ignored by the TCP/IP stack.
   */
  tcpip_set_outputfunc(example_packet_driver_output);

  /*
   * Now we'll make sure that the poll handler is executed initially. We do
   * this by calling process_poll() with this process as its argument.
   */
  process_poll(&example_packet_driver_process);

  /*
   * And we wait for the process to exit.
   */
  PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_EXIT);

  /*
   * Now we shutdown the hardware.
   */
  shutdown_the_hardware();

  /*
   * Here ends the process.
   */
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.4 example-pollhandler.c

```
#include "contiki.h"

PROCESS(example_pollhandler, "Pollhandler example");

static void
exithandler(void)
{
  printf("Process exited\n");
}

static void
pollhandler(void)
{
  printf("Process polled\n");
}

PROCESS_THREAD(example_pollhandler, ev, data)
{

  PROCESS_POLLHANDLER(pollhandler());
```

```
  PROCESS_EXITHANDLER(exithandler());

  PROCESS_BEGIN();

  while(1) {
    PROCESS_WAIT_EVENT();
  }

  PROCESS_END();
}
```

10.5 example-program.c

```
/*
 * This file contains an example of how a Contiki program looks.
 *
 * The program opens a UDP broadcast connection and sends one packet
 * every second.
 */

#include "contiki.h"
#include "contiki-net.h"

/*
 * All Contiki programs must have a process, and we declare it here.
 */
PROCESS(example_program_process, "Example process");

/*
 * To make the program send a packet once every second, we use an
 * event timer (etimer).
 */
static struct etimer timer;

/*---------------------------------------------------------------------------*/
/*
 * Here we implement the process. The process is run whenever an event
 * occurs, and the parameters "ev" and "data" will we set to the event
 * type and any data that may be passed along with the event.
 */
PROCESS_THREAD(example_program_process, ev, data)
{
  /*
   * Declare the UDP connection. Note that this *MUST* be declared
   * static, or otherwise the contents may be destroyed. The reason
   * for this is that the process runs as a protothread, and
   * protothreads do not support stack variables.
   */
  static struct uip_udp_conn *c;

  /*
   * A process thread starts with PROCESS_BEGIN() and ends with
   * PROCESS_END().
   */
  PROCESS_BEGIN();

  /*
   * We create the UDP connection to port 4321. We don’t want to
   * attach any special data to the connection, so we pass it a NULL
   * parameter.
   */
  c = udp_broadcast_new(HTONS(4321), NULL);

  /*
   * Loop for ever.
   */
```

```
  while(1) {

    /*
     * We set a timer that wakes us up once every second.
     */
    etimer_set(&timer, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

    /*
     * Now, this is a the tricky bit: in order for us to send a UDP
     * packet, we must call upon the uIP TCP/IP stack process to call
     * us. (uIP works under the Hollywood principle: "Don’t call us,
     * we’ll call you".) We use the function tcpip_poll_udp() to tell
     * uIP to call us, and then we wait for the uIP event to come.
     */
    tcpip_poll_udp(c);
    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);

    /*
     * We can now send our packet.
     */
    uip_send("Hello", 5);

    /*
     * We’re done now, so we’ll just loop again.
     */
  }

  /*
   * The process ends here. Even though our program sits is a while(1)
   * loop, we must put the PROCESS_END() at the end of the process, or
   * else the program won’t compile.
   */
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.6 example-psock-server.c

```
/*
 * This is a small example of how to write a TCP server using
 * Contiki’s protosockets. It is a simple server that accepts one line
 * of text from the TCP connection, and echoes back the first 10 bytes
 * of the string, and then closes the connection.
 *
 * The server only handles one connection at a time.
 *
 */

#include <string.h>

/*
 * We include "contiki-net.h" to get all network definitions and
 * declarations.
 */
#include "contiki-net.h"

/*
 * We define one protosocket since we’ve decided to only handle one
 * connection at a time. If we want to be able to handle more than one
 * connection at a time, each parallell connection needs its own
 * protosocket.
 */
static struct psock ps;

/*
```

```
 * We must have somewhere to put incoming data, and we use a 10 byte
 * buffer for this purpose.
 */
static char buffer[10];

/*---------------------------------------------------------------------------*/
/*
 * A protosocket always requires a protothread. The protothread
 * contains the code that uses the protosocket. We define the
 * protothread here.
 */
static
PT_THREAD(handle_connection(struct psock *p))
{
  /*
   * A protosocket's protothread must start with a PSOCK_BEGIN(), with
   * the protosocket as argument.
   *
   * Remember that the same rules as for protothreads apply: do NOT
   * use local variables unless you are very sure what you are doing!
   * Local (stack) variables are not preserved when the protothread
   * blocks.
   */
  PSOCK_BEGIN(p);

  /*
   * We start by sending out a welcoming message. The message is sent
   * using the PSOCK_SEND_STR() function that sends a null-terminated
   * string.
   */
  PSOCK_SEND_STR(p, "Welcome, please type something and press return.\n");

  /*
   * Next, we use the PSOCK_READTO() function to read incoming data
   * from the TCP connection until we get a newline character. The
   * number of bytes that we actually keep is dependant of the length
   * of the input buffer that we use. Since we only have a 10 byte
   * buffer here (the buffer[] array), we can only remember the first
   * 10 bytes received. The rest of the line up to the newline simply
   * is discarded.
   */
  PSOCK_READTO(p, '\n');

  /*
   * And we send back the contents of the buffer. The PSOCK_DATALEN()
   * function provides us with the length of the data that we've
   * received. Note that this length will not be longer than the input
   * buffer we're using.
   */
  PSOCK_SEND_STR(p, "Got the following data: ");
  PSOCK_SEND(p, buffer, PSOCK_DATALEN(p));
  PSOCK_SEND_STR(p, "Good bye!\r\n");

  /*
   * We close the protosocket.
   */
  PSOCK_CLOSE(p);

  /*
   * And end the protosocket's protothread.
   */
  PSOCK_END(p);
}
/*---------------------------------------------------------------------------*/
/*
 * We declare the process.
 */
```

```
PROCESS(example_psock_server_process, "Example protosocket server");
/*---------------------------------------------------------------------------*/
/*
 * The definition of the process.
 */
PROCESS_THREAD(example_psock_server_process, ev, data)
{
  /*
   * The process begins here.
   */
  PROCESS_BEGIN();

  /*
   * We start with setting up a listening TCP port. Note how we're
   * using the HTONS() macro to convert the port number (1010) to
   * network byte order as required by the tcp_listen() function.
   */
  tcp_listen(HTONS(1010));

  /*
   * We loop for ever, accepting new connections.
   */
  while(1) {

    /*
     * We wait until we get the first TCP/IP event, which probably
     * comes because someone connected to us.
     */
    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);

    /*
     * If a peer connected with us, we'll initialize the protosocket
     * with PSOCK_INIT().
     */
    if(uip_connected()) {

      /*
       * The PSOCK_INIT() function initializes the protosocket and
       * binds the input buffer to the protosocket.
       */
      PSOCK_INIT(&ps, buffer, sizeof(buffer));

      /*
       * We loop until the connection is aborted, closed, or times out.
       */
      while(!(uip_aborted() || uip_closed() || uip_timedout())) {

        /*
         * We wait until we get a TCP/IP event. Remember that we
         * always need to wait for events inside a process, to let
         * other processes run while we are waiting.
         */
        PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);

        /*
         * Here is where the real work is taking place: we call the
         * handle_connection() protothread that we defined above. This
         * protothread uses the protosocket to receive the data that
         * we want it to.
         */
        handle_connection(&ps);
      }
    }
  }

  /*
   * We must always declare the end of a process.
```

```
   */
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.7 test-abc.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ‘‘AS IS’’ AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: test-abc.c,v 1.3 2007/03/25 12:10:29 adamdunkels Exp $
 */

/**
 * \file
 *         Testing the abc layer in Rime
 * \author
 *         Adam Dunkels <adam@sics.se>
 */

#include "contiki.h"
#include "net/rime.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include <stdio.h>
/*---------------------------------------------------------------------------*/
PROCESS(test_abc_process, "ABC test");
AUTOSTART_PROCESSES(&test_abc_process);
/*---------------------------------------------------------------------------*/
static void
abc_recv(struct abc_conn *c)
{
  printf("abc message received ’%s’\n", (char *)rimebuf_dataptr());
}
const static struct abc_callbacks abc_call = {abc_recv};
static struct abc_conn abc;
```

```
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(test_abc_process, ev, data)
{
  PROCESS_EXITHANDLER(abc_close(&abc);)

  PROCESS_BEGIN();

  abc_open(&abc, 128, &abc_call);

  while(1) {
    static struct etimer et;

    etimer_set(&et, CLOCK_SECOND * 2);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

    rimebuf_copyfrom("Hej", 4);
    abc_send(&abc);

  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.8 test-meshroute.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: test-meshroute.c,v 1.3 2007/03/25 12:10:29 adamdunkels Exp $
 */

/**
 * \file
 *         A brief description of what this file is.
 * \author
 *         Adam Dunkels <adam@sics.se>
 */
```

```
#include "contiki.h"
#include "net/rime.h"
#include "net/rime/mesh.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include <stdio.h>

static struct mesh_conn mesh;
/*---------------------------------------------------------------------------*/
PROCESS(test_mesh_process, "Mesh test");
AUTOSTART_PROCESSES(&test_mesh_process);
/*---------------------------------------------------------------------------*/
static void
sent(struct mesh_conn *c)
{
  printf("packet sent\n");
}
static void
timedout(struct mesh_conn *c)
{
  printf("packet timedout\n");
}
static void
recv(struct mesh_conn *c, rimeaddr_t *from)
{
  printf("Data received from %d: %.*s (%d)\n", from->u16[0],
         rimebuf_datalen(), (char *)rimebuf_dataptr(), rimebuf_datalen());

  rimebuf_copyfrom("Hopp", 4);
  mesh_send(&mesh, from);
}

const static struct mesh_callbacks callbacks = {recv, sent, timedout};
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(test_mesh_process, ev, data)
{
  PROCESS_EXITHANDLER(mesh_close(&mesh);)
  PROCESS_BEGIN();

  mesh_open(&mesh, 128, &callbacks);

  button_sensor.activate();

  while(1) {
    rimeaddr_t addr;
    static struct etimer et;

    /*    etimer_set(&et, CLOCK_SECOND * 4);*/
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et) ||
                             (ev == sensors_event && data == &button_sensor));

    printf("Button\n");

    /*
     * Send a message containing "Hej" (3 characters) to node number
     * 6.
     */

    rimebuf_copyfrom("Hej", 3);
    addr.u8[0] = 161;
    addr.u8[1] = 161;
    mesh_send(&mesh, &addr);
  }
```

```
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.9 test-rudolph0.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: test-rudolph0.c,v 1.5 2007/05/22 21:04:19 adamdunkels Exp $
 */

/**
 * \file
 *         Testing the rudolph0 code in Rime
 * \author
 *         Adam Dunkels <adam@sics.se>
 */

#include "contiki.h"
#include "net/rime/rudolph0.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include <stdio.h>

#define FILESIZE 200

/*---------------------------------------------------------------------------*/
PROCESS(test_rudolph0_process, "Rudolph0 test");
AUTOSTART_PROCESSES(&test_rudolph0_process);
/*---------------------------------------------------------------------------*/
static void
write_chunk(struct rudolph0_conn *c, int offset, int flag,
            char *data, int datalen)
{
  int fd;
```

```
  if(flag == RUDOLPH0_FLAG_NEWFILE) {
    /*    printf("+++ rudolph0 new file incoming at %lu\n", clock_time());*/
    leds_on(LEDS_RED);
    fd = cfs_open("codeprop.out", CFS_WRITE);
  } else {
    fd = cfs_open("codeprop.out", CFS_WRITE + CFS_APPEND);
  }

  if(datalen > 0) {
    int ret;
    cfs_seek(fd, offset);
    ret = cfs_write(fd, data, datalen);
    /*    printf("write_chunk wrote %d bytes at %d, %d\n", ret, offset, (unsigned char)data[0]);*/
  }

  cfs_close(fd);

  if(flag == RUDOLPH0_FLAG_LASTCHUNK) {
    int i;
    /*    printf("+++ rudolph0 entire file received at %lu\n", clock_time());*/
    leds_off(LEDS_RED);
    leds_on(LEDS_YELLOW);
    fd = cfs_open("hej", CFS_READ);
    for(i = 0; i < FILESIZE; ++i) {
      unsigned char buf;
      cfs_read(fd, &buf, 1);
      if(buf != (unsigned char)i) {
        printf("error: diff at %d, %d != %d\n", i, i, buf);
        break;
      }
    }
    cfs_close(fd);
  }
}
static int
read_chunk(struct rudolph0_conn *c, int offset, char *to, int maxsize)
{
  int fd;
  int ret;

  fd = cfs_open("hej", CFS_READ);

  cfs_seek(fd, offset);
  ret = cfs_read(fd, to, maxsize);
  /*  printf("read_chunk %d bytes at %d, %d\n", ret, offset, (unsigned char)to[0]);*/
  cfs_close(fd);
  return ret;
}
const static struct rudolph0_callbacks rudolph0_call = {write_chunk,
                                                         read_chunk};
static struct rudolph0_conn rudolph0;
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(test_rudolph0_process, ev, data)
{
  static int fd;

  PROCESS_EXITHANDLER(rudolph0_close(&rudolph0);)

  PROCESS_BEGIN();

  PROCESS_PAUSE();


  rudolph0_open(&rudolph0, 128, &rudolph0_call);
  button_sensor.activate();
```

```
  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                             data == &button_sensor);
    {
      int i;

      fd = cfs_open("hej", CFS_WRITE);
      for(i = 0; i < FILESIZE; i++) {
        unsigned char buf = i;
        cfs_write(fd, &buf, 1);
      }
      cfs_close(fd);
    }
    rudolph0_send(&rudolph0, CLOCK_SECOND / 4);

    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                             data == &button_sensor);
    rudolph0_stop(&rudolph0);

  }
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.10 test-rudolph1.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ‘‘AS IS’’ AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: test-rudolph1.c,v 1.7 2007/05/15 08:10:32 adamdunkels Exp $
 */

/**
 * \file
 *         Testing the rudolph1 code in Rime
 * \author
 *         Adam Dunkels <adam@sics.se>
 */
```

```
#include "contiki.h"
#include "net/rime/rudolph1.h"

#include "dev/button-sensor.h"

#include "dev/leds.h"

#include "cfs/cfs.h"

#include "sys/rtimer.h"

#include <stdio.h>

#define FILESIZE 2000

/*---------------------------------------------------------------------------*/
PROCESS(test_rudolph1_process, "Rudolph1 test");
AUTOSTART_PROCESSES(&test_rudolph1_process);
/*---------------------------------------------------------------------------*/
static void
write_chunk(struct rudolph1_conn *c, int offset, int flag,
            char *data, int datalen)
{
  int fd;
#if NETSIM
  {
    char buf[100];
    sprintf(buf, "%d%%", (100 * (offset + datalen)) / FILESIZE);
    ether_set_text(buf);
  }
#endif /* NETSIM */

  if(flag == RUDOLPH1_FLAG_NEWFILE) {
    /*printf("+++ rudolph1 new file incoming at %lu\n", clock_time());*/
    leds_on(LEDS_RED);
    fd = cfs_open("codeprop.out", CFS_WRITE);
  } else {
    fd = cfs_open("codeprop.out", CFS_WRITE + CFS_APPEND);
  }

  if(datalen > 0) {
    int ret;
    cfs_seek(fd, offset);
    ret = cfs_write(fd, data, datalen);
  }

  cfs_close(fd);

  if(flag == RUDOLPH1_FLAG_LASTCHUNK) {
    int i;
    printf("+++ rudolph1 entire file received at %d, %d\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1]);
    leds_off(LEDS_RED);
    leds_on(LEDS_YELLOW);

    fd = cfs_open("hej", CFS_READ);
    for(i = 0; i < FILESIZE; ++i) {
      unsigned char buf;
      cfs_read(fd, &buf, 1);
      if(buf != (unsigned char)i) {
        printf("%d.%d: error: diff at %d, %d != %d\n",
               rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
               i, i, buf);
        break;
      }
    }
#if NETSIM
```

```
    ether_send_done();
#endif
    cfs_close(fd);
  }
}
static int
read_chunk(struct rudolph1_conn *c, int offset, char *to, int maxsize)
{
  int fd;
  int ret;

  fd = cfs_open("hej", CFS_READ);

  cfs_seek(fd, offset);
  ret = cfs_read(fd, to, maxsize);
  /*  printf("%d.%d: read_chunk %d bytes at %d, %d\n",
         rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
         ret, offset, (unsigned char)to[0]);*/
  cfs_close(fd);
  return ret;
}
const static struct rudolph1_callbacks rudolph1_call = {write_chunk,
                                                        read_chunk};
static struct rudolph1_conn rudolph1;
/*---------------------------------------------------------------------------*/
static void
log_queuelen(struct rtimer *t, void *ptr)
{
#if NETSIM
  extern u8_t queuebuf_len, queuebuf_ref_len;
  node_log("%d %d\n",
           queuebuf_len,
           queuebuf_ref_len);
  rtimer_set(t, RTIMER_TIME(t) + RTIMER_ARCH_SECOND, 1,
             log_queuelen, ptr);
#endif /* NETSIM */
}
/*---------------------------------------------------------------------------*/

PROCESS_THREAD(test_rudolph1_process, ev, data)
{
  static int fd;
  static struct rtimer t;
  PROCESS_EXITHANDLER(rudolph1_close(&rudolph1);)
  PROCESS_BEGIN();

  PROCESS_PAUSE();


  rudolph1_open(&rudolph1, 128, &rudolph1_call);
  button_sensor.activate();

  rtimer_set(&t, RTIMER_NOW() + RTIMER_ARCH_SECOND, 1,
             log_queuelen, NULL);

  PROCESS_PAUSE();

  if(rimeaddr_node_addr.u8[0] == 1 &&
     rimeaddr_node_addr.u8[1] == 1) {
    {
      int i;

      fd = cfs_open("hej", CFS_WRITE);
      for(i = 0; i < FILESIZE; i++) {
        unsigned char buf = i;
        cfs_write(fd, &buf, 1);
      }
```

```
      cfs_close(fd);
    }
    rudolph1_send(&rudolph1, CLOCK_SECOND * 2);
#if NETSIM
    ether_send_done();
#endif /* NETSIM */

  }

  while(1) {

    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                             data == &button_sensor);
    rudolph1_stop(&rudolph1);

  }
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.11 test-treeroute.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: test-treeroute.c,v 1.5 2007/05/22 21:04:34 adamdunkels Exp $
 */

/**
 * \file
 *         A brief description of what this file is.
 * \author
 *         Adam Dunkels <adam@sics.se>
 */

#include "contiki.h"
#include "net/rime.h"
#include "net/rime/tree.h"
#include "dev/leds.h"
```

```
#include "dev/pir-sensor.h"
#include "dev/button-sensor.h"

#include <stdio.h>

static struct tree_conn tc;

/*---------------------------------------------------------------------------*/
PROCESS(test_tree_process, "Test tree process");
PROCESS(depth_blink_process, "Depth indicator");
AUTOSTART_PROCESSES(&test_tree_process, &depth_blink_process);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(depth_blink_process, ev, data)
{
  static struct etimer et;
  static int count;

  PROCESS_BEGIN();

  while(1) {
    etimer_set(&et, CLOCK_SECOND * 1);
    PROCESS_WAIT_UNTIL(etimer_expired(&et));
    count = tree_depth(&tc);
    if(count == TREE_MAX_DEPTH) {
      leds_on(LEDS_RED);
    } else {
      leds_off(LEDS_RED);
      while(count > 0) {
        leds_on(LEDS_RED);
        etimer_set(&et, CLOCK_SECOND / 10);
        PROCESS_WAIT_UNTIL(etimer_expired(&et));
        leds_off(LEDS_RED);
        etimer_set(&et, CLOCK_SECOND / 10);
        PROCESS_WAIT_UNTIL(etimer_expired(&et));
        --count;
      }
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
static void
recv(rimeaddr_t *originator, u8_t seqno, u8_t hops)
{
  printf("Sink got message from %d.%d, seqno %d, hops %d: len %d '%s'\n",
         originator->u8[0], originator->u8[1],
         seqno, hops,
         rimebuf_datalen(),
         (char *)rimebuf_dataptr());

}
/*---------------------------------------------------------------------------*/
static const struct tree_callbacks callbacks = { recv };
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(test_tree_process, ev, data)
{
  PROCESS_BEGIN();

  tree_open(&tc, 128, &callbacks);

  while(1) {

    PROCESS_WAIT_EVENT();

    if(ev == sensors_event) {
```

```
      if(data == &pir_sensor) {
        rimebuf_clear();
        rimebuf_set_datalen(sprintf(rimebuf_dataptr(),
                                    "%d", pir_sensor.value(0)));
        tree_send(&tc, 10);
      }

      if(data == &button_sensor) {
        printf("Button\n");
        tree_set_sink(&tc, 1);
      }
    }

  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.12 test-trickle.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: test-trickle.c,v 1.5 2007/05/15 08:10:32 adamdunkels Exp $
 */

/**
 * \file
 *         Testing the trickle code in Rime
 * \author
 *         Adam Dunkels <adam@sics.se>
 */

#include "contiki.h"
#include "net/rime/trickle.h"

#include "dev/button-sensor.h"
```

```
#include "dev/leds.h"

#include <stdio.h>
/*---------------------------------------------------------------------------*/
PROCESS(test_trickle_process, "TRICKLE test");
AUTOSTART_PROCESSES(&test_trickle_process);
/*---------------------------------------------------------------------------*/
static void
trickle_recv(struct trickle_conn *c)
{
  printf("%d.%d: trickle message received '%s'\n",
         rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
         (char *)rimebuf_dataptr());
}
const static struct trickle_callbacks trickle_call = {trickle_recv};
static struct trickle_conn trickle;
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(test_trickle_process, ev, data)
{
  PROCESS_EXITHANDLER(trickle_close(&trickle);)
  PROCESS_BEGIN();

  trickle_open(&trickle, CLOCK_SECOND, 128, &trickle_call);
  button_sensor.activate();

  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                             data == &button_sensor);

    rimebuf_copyfrom("Hello, world", 13);
    trickle_send(&trickle);

  }
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

10.13 test-uabc.c

10.14 test-uibc.c

Index