

# Contiki 1.2-devel0 Reference Manual

Generated by Doxygen 1.3.6

Tue Sep 14 01:03:39 2004



# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>The Contiki Operating System</b>            | <b>1</b> |
| <b>2</b> | <b>Contiki 1.2-devel0 Module Index</b>         | <b>3</b> |
| 2.1      | Contiki 1.2-devel0 Modules . . . . .           | 3        |
| <b>3</b> | <b>Contiki 1.2-devel0 Data Structure Index</b> | <b>5</b> |
| 3.1      | Contiki 1.2-devel0 Data Structures . . . . .   | 5        |
| <b>4</b> | <b>Contiki 1.2-devel0 File Index</b>           | <b>7</b> |
| 4.1      | Contiki 1.2-devel0 File List . . . . .         | 7        |
| <b>5</b> | <b>Contiki 1.2-devel0 Module Documentation</b> | <b>9</b> |
| 5.1      | System events . . . . .                        | 9        |
| 5.2      | The Contiki event kernel . . . . .             | 11       |
| 5.3      | The Contiki program loader . . . . .           | 16       |
| 5.4      | Protothreads . . . . .                         | 19       |
| 5.5      | Local continuations . . . . .                  | 24       |
| 5.6      | Protothread semaphores . . . . .               | 25       |
| 5.7      | CTK application functions . . . . .            | 28       |
| 5.8      | The CTK graphical user interface. . . . .      | 42       |
| 5.9      | CTK device driver functions . . . . .          | 46       |
| 5.10     | The uIP TCP/IP stack . . . . .                 | 51       |
| 5.11     | uIP configuration functions . . . . .          | 61       |
| 5.12     | uIP initialization functions . . . . .         | 63       |
| 5.13     | uIP device driver functions . . . . .          | 64       |
| 5.14     | uIP application functions . . . . .            | 68       |
| 5.15     | uIP conversion functions . . . . .             | 75       |
| 5.16     | uIP Address Resolution Protocol . . . . .      | 80       |
| 5.17     | uIP TCP throughput booster hack . . . . .      | 82       |
| 5.18     | uIP hostname resolver functions . . . . .      | 83       |

|          |   |            |
|----------|---|------------|
| 5.19     | Socket library . . . . .                                | 85         |
| 5.20     | Memory block management functions . . . . .             | 91         |
| 5.21     | Peemotive multi-threading . . . . .                     | 94         |
| 5.22     | Architecture support for multi-threading . . . . .      | 98         |
| 5.23     | Multi-threading library convenience functions . . . . . | 100        |
| 5.24     | System signals . . . . .                                | 102        |
| 5.25     | Uiparch . . . . .                                       | 104        |
| <b>6</b> | <b>Contiki 1.2-devel0 Data Structure Documentation</b>  | <b>105</b> |
| 6.1      | ctk_menu Struct Reference . . . . .                     | 105        |
| 6.2      | ctk_menuitem Struct Reference . . . . .                 | 107        |
| 6.3      | ctk_menus Struct Reference . . . . .                    | 108        |
| 6.4      | ctk_widget Struct Reference . . . . .                   | 109        |
| 6.5      | ctk_window Struct Reference . . . . .                   | 111        |
| 6.6      | dsc Struct Reference . . . . .                          | 113        |
| 6.7      | socket Struct Reference . . . . .                       | 114        |
| 6.8      | uip_conn Struct Reference . . . . .                     | 115        |
| 6.9      | uip_eth_addr Struct Reference . . . . .                 | 117        |
| 6.10     | uip_eth_hdr Struct Reference . . . . .                  | 118        |
| 6.11     | uip_stats Struct Reference . . . . .                    | 119        |
| 6.12     | uip_udp_conn Struct Reference . . . . .                 | 122        |
| <b>7</b> | <b>Contiki 1.2-devel0 File Documentation</b>            | <b>123</b> |
| 7.1      | apps/program-handler.c File Reference . . . . .         | 123        |
| 7.2      | conf/uip-conf.h.example File Reference . . . . .        | 126        |
| 7.3      | conf/www-conf.h.example File Reference . . . . .        | 128        |
| 7.4      | ctk/ctk-draw.h File Reference . . . . .                 | 130        |
| 7.5      | ctk/ctk.c File Reference . . . . .                      | 131        |
| 7.6      | ctk/ctk.h File Reference . . . . .                      | 134        |
| 7.7      | ek/arg.c File Reference . . . . .                       | 139        |
| 7.8      | ek/dsc.h File Reference . . . . .                       | 140        |
| 7.9      | ek/ek.c File Reference . . . . .                        | 141        |
| 7.10     | ek/loader.h File Reference . . . . .                    | 143        |
| 7.11     | ek/mt.c File Reference . . . . .                        | 145        |
| 7.12     | ek/mt.h File Reference . . . . .                        | 146        |
| 7.13     | ek/pt-sem.h File Reference . . . . .                    | 148        |
| 7.14     | ek/pt.h File Reference . . . . .                        | 149        |

|  |     |
|--|-----|
| 7.15 lib/cc.h File Reference . . . . .           | 150 |
| 7.16 lib/ctk-textedit.c File Reference . . . . . | 151 |
| 7.17 lib/ctk-textedit.h File Reference . . . . . | 152 |
| 7.18 lib/memb.c File Reference . . . . .         | 154 |
| 7.19 lib/memb.h File Reference . . . . .         | 155 |
| 7.20 lib/petsciiconv.h File Reference . . . . .  | 156 |
| 7.21 uip/resolv.c File Reference . . . . .       | 157 |
| 7.22 uip/resolv.h File Reference . . . . .       | 158 |
| 7.23 uip/socket.h File Reference . . . . .       | 159 |
| 7.24 uip/uip-split.h File Reference . . . . .    | 161 |
| 7.25 uip/uip.c File Reference . . . . .          | 162 |
| 7.26 uip/uip.h File Reference . . . . .          | 163 |
| 7.27 uip/uip_arp.c File Reference . . . . .      | 168 |
| 7.28 uip/uip_arp.h File Reference . . . . .      | 169 |
| 7.29 uip/uiplib.h File Reference . . . . .       | 170 |



# Chapter 1

## The Contiki Operating System

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

The Contiki operating system is a highly portable, minimalistic operating system for a variety of constrained systems ranging from modern 8-bit microcontrollers for embedded systems to old 8-bit homecomputers. Contiki provides a simple event driven kernel with optional preemptive multithreading, interprocess communication using message passing signals, a dynamic process structure and support for loading and unloading programs, native TCP/IP support using the uIP TCP/IP stack, and a graphical subsystem with either direct graphic support for directly connected terminals or networked virtual display with VNC or Telnet.

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/~adam/contiki/>





## Chapter 2

# Contiki 1.2-devel0 Module Index

### 2.1 Contiki 1.2-devel0 Modules

Here is a list of all modules:

|   |     |
|---|-----|
| System events . . . . .                                 | 9   |
| The Contiki program loader . . . . .                    | 16  |
| Protothreads . . . . .                                  | 19  |
| Local continuations . . . . .                           | 24  |
| Protothread semaphores . . . . .                        | 25  |
| The CTK graphical user interface. . . . .               | 42  |
| CTK application functions . . . . .                     | 28  |
| CTK device driver functions . . . . .                   | 46  |
| The uIP TCP/IP stack . . . . .                          | 51  |
| uIP configuration functions . . . . .                   | 61  |
| uIP initialization functions . . . . .                  | 63  |
| uIP device driver functions . . . . .                   | 64  |
| uIP application functions . . . . .                     | 68  |
| uIP conversion functions . . . . .                      | 75  |
| uIP Address Resolution Protocol . . . . .               | 80  |
| uIP TCP throughput booster hack . . . . .               | 82  |
| uIP hostname resolver functions . . . . .               | 83  |
| Uiparch . . . . .                                       | 104 |
| Socket library . . . . .                                | 85  |
| Memory block management functions . . . . .             | 91  |
| Peemptive multi-threading . . . . .                     | 94  |
| Architecture support for multi-threading . . . . .      | 98  |
| Multi-threading library convenience functions . . . . . | 100 |
| System signals . . . . .                                | 102 |



## Chapter 3

# Contiki 1.2-devel0 Data Structure Index

### 3.1 Contiki 1.2-devel0 Data Structures

Here are the data structures with brief descriptions:

|   |     |
|---|-----|
| <a href="#">ctk_menu</a> (Representation of an individual menu ) . . . . .  | 105 |
| <a href="#">ctk_menuitem</a> (Representation of an individual menu item ) . . . . .   | 107 |
| <a href="#">ctk_menus</a> (Representation of the menu bar ) . . . . .   | 108 |
| <a href="#">ctk_widget</a> (The generic CTK widget structure that contains all other widget structures ) . . . . .                    | 109 |
| <a href="#">ctk_window</a> (Representation of a CTK window ) . . . . .  | 111 |
| <a href="#">dsc</a> (The DSC program description structure ) . . . . .  | 113 |
| <a href="#">socket</a> (The representation of a socket ) . . . . .  | 114 |
| <a href="#">uip_conn</a> (Representation of a uIP TCP connection ) . . . . .  | 115 |
| <a href="#">uip_eth_addr</a> (Representation of a 48-bit Ethernet address ) . . . . .   | 117 |
| <a href="#">uip_eth_hdr</a> (The Ethernet header ) . . . . .  | 118 |
| <a href="#">uip_stats</a> (The structure holding the TCP/IP statistics that are gathered if UIP_STATISTICS is<br>set to 1 ) . . . . . | 119 |
| <a href="#">uip_udp_conn</a> (Representation of a uIP UDP connection ) . . . . .  | 122 |



## Chapter 4

# Contiki 1.2-devel0 File Index

### 4.1 Contiki 1.2-devel0 File List

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

|   |     |
|---|-----|
| apps/ <a href="#">program-handler.c</a> (The program handler, used for loading programs and starting the screen-saver ) . . . . .               | 123 |
| conf/ <a href="#">uip-conf.h.example</a> (UIP configuration file ) . . . . .  | 126 |
| conf/ <a href="#">www-conf.h.example</a> (The Contiki web browser configuration file ) . . . . .  | 128 |
| ctk/ <a href="#">ctk-draw.h</a> (CTK screen drawing module interface, ctk-draw ) . . . . .  | 130 |
| ctk/ <a href="#">ctk.c</a> (The Contiki Toolkit CTK, the Contiki GUI ) . . . . .  | 131 |
| ctk/ <a href="#">ctk.h</a> (CTK header file ) . . . . .   | 134 |
| ek/ <a href="#">arg.c</a> (Argument buffer for passing arguments when starting processes ) . . . . .  | 139 |
| ek/ <a href="#">dsc.h</a> (Declaration of the DSC program description structure ) . . . . .   | 140 |
| ek/ <a href="#">ek.c</a> (Event kernel, event dispatcher and handler of uIP events ) . . . . .  | 141 |
| ek/ <a href="#">ek.h</a> . . . . .  | ??  |
| ek/ <a href="#">loader.h</a> (Default definitions and error values for the Contiki program loader ) . . . . .                                   | 143 |
| ek/ <a href="#">mt.c</a> (Implementation of the architecture agnostic parts of the preemptive multithreading library for Contiki ) . . . . .    | 145 |
| ek/ <a href="#">mt.h</a> (Header file for the preemptive multitasking library for Contiki ) . . . . .   | 146 |
| ek/ <a href="#">pt-sem.h</a> (Counting semaphores implemented on protothreads ) . . . . .   | 148 |
| ek/ <a href="#">pt.h</a> (Protothreads implementation ) . . . . .   | 149 |
| lib/ <a href="#">cc.h</a> (Default definitions of C compiler quirk work-arounds ) . . . . .   | 150 |
| lib/ <a href="#">ctk-textedit.c</a> (An experimental CTK text edit widget ) . . . . .   | 151 |
| lib/ <a href="#">ctk-textedit.h</a> (Header file for the experimental application level CTK textedit widget ) . . . . .                         | 152 |
| lib/ <a href="#">memb.c</a> (Memory block allocation routines ) . . . . .   | 154 |
| lib/ <a href="#">memb.h</a> (Memory block allocation routines ) . . . . .   | 155 |
| lib/ <a href="#">petsciiconv.h</a> (PETSCII/ASCII conversion functions ) . . . . .  | 156 |
| uip/ <a href="#">resolv.c</a> (DNS host name to IP address resolver ) . . . . .   | 157 |
| uip/ <a href="#">resolv.h</a> (UIP DNS resolver code header file ) . . . . .  | 158 |
| uip/ <a href="#">socket.h</a> (Socket library header file ) . . . . .   | 159 |
| uip/ <a href="#">uip-split.h</a> (Module for splitting outbound TCP segments in two to avoid the delayed ACK throughput degradation ) . . . . . | 161 |
| uip/ <a href="#">uip.c</a> (The uIP TCP/IP stack code ) . . . . .   | 162 |
| uip/ <a href="#">uip.h</a> (Header file for the uIP TCP/IP stack ) . . . . .  | 163 |
| uip/ <a href="#">uip_arp.c</a> (Implementation of the ARP Address Resolution Protocol ) . . . . .   | 168 |
| uip/ <a href="#">uip_arp.h</a> (Macros and definitions for the ARP module ) . . . . .   | 169 |
| uip/ <a href="#">uilib.h</a> (Various uIP library functions ) . . . . .   | 170 |



# Chapter 5

## Contiki 1.2-devel0 Module Documentation

### 5.1 System events

#### 5.1.1 Detailed Description

The Contiki system defines a number of default events that can be delivered to processes.

#### Variables

- `ek_event_t ek_event_quit`  
*The "quit" event.*
- `ek_event_t ek_event_msg`  
*A generic message event.*

#### 5.1.2 Variable Documentation

##### 5.1.2.1 `ek_event_t ek_event_msg`

A generic message event.

This event may be used to send messages between processes. The actual interpretation of the message is up to the applications to decide.

##### 5.1.2.2 `ek_event_t ek_event_quit`

The "quit" event.

This event is posted to a process in order to tell it to remove itself from the system. Since each program may have allocated system resources that must be released before the process quits, each program must implement the event handler by itself. A process that receives this event must call `LOADER_UNLOAD()` to unload itself after doing all necessary clean ups (such as closing open windows, deallocate allocated memory, etc.). The following code shows how this can be implemented:

```
static struct ctk_window mainwindow;
static EK_EVENTHANDLER(example_eventhandler, s, data);

static
EK_EVENTHANDLER(example_eventhandler, s, data)
{
    EK_EVENTHANDLER_ARGS(s, data);

    if(s == ek_event_quit) {
        ctk_window_close(&mainwindow);
        ek_exit(&p);
        LOADER_UNLOAD();
    }
}
```



## 5.2 The Contiki event kernel

### 5.2.1 Detailed Description

At the heart of the Contiki desktop environment is the event driven Contiki kernel. Using non-preemptive multitasking, the Contiki event kernel makes it possible to run several programs in parallel. It also provides message passing mechanisms to the running programs.

The Contiki kernel is a simple event driven dispatcher which handles processes, events and uIP events. All code execution is initiated by the dispatcher, and applications are implemented as C functions that must return within a short time after being called. It therefore is not possible to implement processes with, e.g., long-lasting while() loops such as the infamous while(1); loop.

### Modules

- group [The Contiki event kernel](#)

### Functions

- `ek_event_t ek_alloc_event` (void)  
*Allocates a event number.*
- `ek_id_t ek_start` (CC\_REGISTER\_ARG struct ek\_proc \*p)  
*Starts a new process.*
- `void ek_exit` (void)  
*Exit the currently running process.*
- `ek_proc * ek_process` (ek\_id\_t id)  
*Finds the process structure for a specific process ID.*
- `void ek_init` (void)  
*Initializes the dispatcher module.*
- `void ek_process_event` (void)  
*Process the next event in the event queue and deliver it to listening processes.*
- `void ek_process_poll` (void)  
*Call each process' poll handler.*
- `int ek_run` (void)  
*Run the system once - call poll handlers and process one event.*
- `ek_err_t ek_post` (ek\_id\_t id, ek\_event\_t s, ek\_data\_t data)  
*Post an asynchronous event.*
- `void ek_post_synch` (ek\_id\_t id, ek\_event\_t ev, ek\_data\_t data)  
*Post a synchronous event.*
- `char * arg_alloc` (char size)

*Allocates an argument buffer.*

- void [arg\\_free](#) (char \*arg)

*Deallocates an argument buffer.*

### 5.2.1.1 The dispatcher

The dispatcher is the initiator of all program execution in Contiki. After the system has been initialized by the boot up code, the [ek\\_run\(\)](#) function is called. This function never returns, but will sit in a loop in which it does two things.

- Pulls the first event of the event queue and dispatches this to all listening processes ([ek\\_process\\_event\(\)](#)).
- Executes the "poll" handlers of all processes that have registered ([ek\\_process\\_poll\(\)](#)).

Only one event is processed at a time, and the poll handlers of all processes are called between two events are handled.

A process is defined by an initialization function, a event handler, a uIP event handler, and an poll handler. The event handler is called when a event has been posted, for which the process is currently listening. The uIP event handler is called when the uIP TCP/IP stack has an event to deliver to the process. Such events can be that new data has arrived on a connection, that previously sent data has been acknowledged or that a connection has been closed. The poll handler is periodically called by the system.

A process is started by calling the [ek\\_start\(\)](#) function. This function must be called by the initialization function before any other dispatcher function is called. When the function returns, the new process is running.

The initialization function is declared with the special `LOADER_INIT()` macro. The initialization function takes a single argument; a char \* pointer.

The function [ek\\_exit\(\)](#) is used to tell the dispatcher that a process has exited. This function must be called by the process itself, and must be called the process unloads itself.

#### Note:

It is not possible to call [ek\\_exit\(\)](#) on behalf of another process - instead, post the event [ek\\_event\\_quit](#) with the process as a receiver. The other process should then listen for this event, and call [ek\\_exit\(\)](#) when the event is received.

The dispatcher can pass events between different processes. Events are simple messages that consist of a event number and a generic data pointer called the event data. The event data can be used to pass messages between processes. In order for a event to be delivered to a process, the process must be listening for the event number.

If a process has registered an poll handler, the dispatcher will call it as often as possible. The poll handler can be used to implement timer based functionality (by checking the [ek\\_clock\(\)](#) function), or other background processing. The poll handler must return to the caller within a short time, or otherwise the system will feel sluggish.

The uIP TCP/IP stack will call the dispatcher when a uIP event has occurred. The dispatcher will find the right process for which the event is intended and call the process' uIP handler function.

### 5.2.1.2 Argument buffer

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' address 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.

## 5.2.2 Function Documentation

### 5.2.2.1 `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.

### 5.2.2.2 `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.

### 5.2.2.3 `ek_event_t ek_alloc_event (void)`

Allocates a event number.

**Returns:**

The allocated event number or EK\_EVENT\_NONE if no event number could be allocated.

### 5.2.2.4 `void ek_exit (void)`

Exit the currently running process.

This function causes the currently running process to exit. The function must be called by the process before it unloads itself, or the system will crash.

#### 5.2.2.5 void ek\_init (void)

Initializes the dispatcher module.

Must be called during the initialization of Contiki.

#### 5.2.2.6 ek\_err\_t ek\_post (ek\_id\_t id, ek\_event\_t s, ek\_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:

*s* The event to be posted.

*data* The auxillary data to be sent with the event

*id* The process ID to which the event should be posted, or EK\_BROADCAST if the event should be posted to all processes.

##### Return values:

**EK\_ERR\_OK** The event could be posted.

**EK\_ERR\_FULL** The event queue was full and the event could not be posted.

#### 5.2.2.7 void ek\_post\_synch (ek\_id\_t id, ek\_event\_t ev, ek\_data\_t data)

Post a synchronous event.

This function emits a event and calls the listening processes' event handlers immediately, before returning to the caller. This function requires more call stack space than the ek\_emit() function and should be used with care, and only in situations where the exact implications are known.

In most situations, the ek\_emit() function should be used instead.

##### Parameters:

*s* The event to be emitted.

*data* The auxillary data to be sent with the event

*id* The process ID to which the event should be emitted, or EK\_BROADCAST if the event should be emitted to all processes listening for the event.

#### 5.2.2.8 struct ek\_proc\* ek\_process (ek\_id\_t id)

Finds the process structure for a specific process ID.

##### Parameters:

*id* The process ID for the process.

##### Returns:

The process structure for the process, or NULL if there process ID was not found.

### 5.2.2.9 int ek\_run (void)

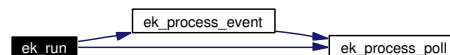
Run the system once - call poll handlers and process one event.

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.

Here is the call graph for this function:



### 5.2.2.10 ek\_id\_t ek\_start (CC\_REGISTER\_ARG struct ek\_proc \* p)

Starts a new process.

Is called by a program in order to start a new process for the program. This function should be called quite early in the initialization procedure of a new process. In particular, it must be called before any other dispatcher functions, or functions of other modules that make use of dispatcher functions. Most CTK functions call dispatcher functions, and should therefore not be called before ek\_start() is called.

Example:

```

static void app_poll(void);
static EK_EVENTHANDLER(app_eventhandler, s, data);
static struct ek_proc p =
    {EK_PROC("Generic applications", app_poll, app_eventhandler, NULL)};
static ek_id_t id = EK_ID_NONE;

LOADER_INIT_FUNC(app_init, arg)
{
    arg_free(arg);

    if(id == EK_ID_NONE) {
        id = ek_start(&p);

        rest_of_initialization();
    }
}
  
```

#### Parameters:

*p* A pointer to a ek\_proc struct that must be found in the process own memory space.

#### Returns:

The process identifier for the new process or EK\_ID\_NONE if the process could not be started.

Here is the call graph for this function:



## 5.3 The Contiki program loader

### 5.3.1 Detailed Description

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

#### Data Structures

- struct `dsc`  
*The DSC program description structure.*

#### Defines

- #define `DSC(dscname, description, prgname, initfunc, icon)` const struct `dsc` dscname = { description, prgname, icon }  
*Instantiating macro for the DSC structure.*
- #define `LOADER_OK` 0 /\*\*< No error. \*/  
*No error.*
- #define `LOADER_ERR_READ` 1 /\*\*< Read error. \*/  
*Read error.*
- #define `LOADER_ERR_HDR` 2 /\*\*< Header error. \*/  
*Header error.*
- #define `LOADER_ERR_OS` 3 /\*\*< Wrong OS. \*/  
*Wrong OS.*
- #define `LOADER_ERR_FMT` 4 /\*\*< Data format error. \*/  
*Data format error.*
- #define `LOADER_ERR_MEM` 5 /\*\*< Not enough memory. \*/  
*Not enough memory.*
- #define `LOADER_ERR_OPEN` 6 /\*\*< Could not open file. \*/  
*Could not open file.*
- #define `LOADER_ERR_ARCH` 7 /\*\*< Wrong architecture. \*/  
*Wrong architecture.*
- #define `LOADER_ERR_VERSION` 8 /\*\*< Wrong OS version. \*/  
*Wrong OS version.*
- #define `LOADER_ERR_NOLOADER` 9 /\*\*< Program loading not supported. \*/  
*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).*

### 5.3.1.1 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.

## 5.3.2 Define Documentation

### 5.3.2.1 `#define DSC(dscname, description, prgname, initfunc, icon) const struct dsc dscname = {description, prgname, icon}`

Instantiating 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.

### 5.3.2.2 `#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.

**5.3.2.3 #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.

**5.3.2.4 #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.

**5.3.2.5 #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.



## 5.4 Protothreads

### 5.4.1 Detailed Description

Protothreads are lightweight stackless threads that is used to provide blocking contexts in event-driven systems. This is useful for implementing sequential control flow, without requiring ordinary threads and multiple stacks. Protothreads provides conditional blocking inside a C function.

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.

Because protothreads are stackless, a protothread can only run within a single C function. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls are made by spawning a separate protothread for each potentially blocking function.

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.

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.

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()`.

### Files

- file `pt.h`  
*Protothreads implementation.*

### Modules

- group `Local continuations`
- group `Protothread semaphores`

### Defines

- `#define PT_THREAD(name_args)`  
*Declaration of a protothread.*
- `#define PT_INIT(pt)`  
*Initialize a protothread.*
- `#define PT_BEGIN(pt)`  
*Start a protothread.*
- `#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.*
- #define [PT\\_WAIT\\_THREAD](#)(pt, thread)  
*Block and wait until a child protothread completes.*
- #define [PT\\_SPAWN](#)(pt, thread)  
*Spawn a child protothread and wait until it exits.*
- #define [PT\\_RESTART](#)(pt)  
*Restart the protothread.*
- #define [PT\\_EXIT](#)(pt)  
*Exit the protothread.*
- #define [PT\\_END](#)(pt)  
*Declare the end of a protothread.*

## 5.4.2 Define Documentation

### 5.4.2.1 #define PT\_BEGIN(pt)

Start a protothread.

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

#### Parameters:

*pt* A pointer to the protothread control structure.

Example:

```
PT_THREAD(producer(struct pt *p, int event)) {
    int empty;
    empty = (event == CONSUMED || event == DROPPED);

    PT_BEGIN(p);

    PT_WAIT_UNTIL(empty);
    produce();
    PT_WAIT_UNTIL(event == PRODUCED);

    PT_EXIT(p);
}
```

### 5.4.2.2 #define PT\_END(pt)

Declare the end of a protothread.

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

**Parameters:**

*pt* A pointer to the protothread control structure.

**5.4.2.3 #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.

**5.4.2.4 #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.

Example:

```
int main(void) {
    struct pt p;
    int event;

    PT_INIT(&p);
    while(PT_RUNNING(consumer(&p, event))) {
        event = get_event();
    }
}
```

**5.4.2.5 #define PT\_RESTART(pt)**

Restart the protothread.

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

**Parameters:**

*pt* A pointer to the protothread control structure.

**5.4.2.6 #define PT\_SPAWN(pt, 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.

*thread* The child protothread with arguments

#### 5.4.2.7 #define PT\_THREAD(name\_args)

Declaration of a protothread.

This macro is used to declare a protothread.

Example:

```
PT_THREAD(consumer(struct pt *p, int event)) {
    PT_BEGIN(p);
    PT_WAIT_UNTIL(event == AVAILABLE);
    consume();
    PT_WAIT_UNTIL(event == CONSUMED);
    acknowledge_consumed();
    PT_END(p);
}
```

#### 5.4.2.8 #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

Example:

```
PT_THREAD(child(struct pt *p, int event)) {
    PT_BEGIN(p);

    PT_WAIT_UNTIL(event == EVENT1);

    PT_END(p);
}

PT_THREAD(parent(struct pt *p, struct pt *child_pt, int event)) {
    PT_BEGIN(p);

    PT_INIT(child_pt);

    PT_WAIT_THREAD(p, child(child_pt, event));

    PT_END(p);
}
```

#### 5.4.2.9 #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.

Example:

```
PT_THREAD(seconds(struct pt *p)) {  
    PT_BEGIN(p);  
  
    PT_WAIT_UNTIL(p, time >= 2 * SECOND);  
    printf("Two seconds have passed\n");  
  
    PT_EXIT(p);  
}
```

**5.4.2.10 #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.

## 5.5 Local continuations

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.

## 5.6 Protothread semaphores

### 5.6.1 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 if the buffer is empty, and the "full" semaphore is used to block the producer if 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](#)

*Couting 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.*

## 5.6.2 Define Documentation

### 5.6.2.1 #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.

### 5.6.2.2 #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

### 5.6.2.3 #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

## 5.7 CTK application functions

### 5.7.1 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) do {memset((e) → text, 0, (e) → len); (e) → xpos = 0;} while(0);  
*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, id) ctk\_icon\_add((struct [ctk\\_widget](#) \*)icon, id)  
*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

- void `ctk_widget_redraw` (struct `ctk_widget *`w)  
*Redraws a widget.*
- void `ctk_desktop_redraw` (struct `ctk_desktop *`d)  
*Redraw the entire desktop.*
- 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, ek\_id\_t id)  
*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

- ek\_event\_t [ctk\\_signal\\_keypress](#)  
*Emitted for every key being pressed.*
- ek\_event\_t [ctk\\_signal\\_widget\\_activate](#)  
*Emitted when a widget is activated (pressed).*
- ek\_event\_t [ctk\\_signal\\_widget\\_select](#)  
*Emitted when a widget is selected.*

- `ek_event_t ctk_signal_menu_activate`  
*Emitted when a menu item is activated.*
- `ek_event_t ctk_signal_window_close`  
*Emitted when a window is closed.*
- `ek_event_t ctk_signal_pointer_move`  
*Emitted when the mouse pointer is moved.*
- `ek_event_t ctk_signal_pointer_button`  
*Emitted when a mouse button is pressed.*
- `ek_event_t ctk_signal_button_activate`  
*Same as `ctk_signal_widget_activate`.*
- `ek_event_t ctk_signal_button_hover`  
*Same as `ctk_signal_widget_select`.*
- `ek_event_t ctk_signal_hyperlink_activate`  
*Emitted when a hyperlink is activated.*
- `ek_event_t ctk_signal_hyperlink_hover`  
*Same as `ctk_signal_widget_select`.*

## 5.7.2 Define Documentation

### 5.7.2.1 `#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.

### 5.7.2.2 `#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.

### 5.7.2.3 **#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.

### 5.7.2.4 **#define CTK\_ICON(*title*, *bitmap*, *textmap*)**

#### Value:

```
NULL, NULL, 0, 0, CTK_WIDGET_ICON, 2, 4, CTK_WIDGET_FLAG_INITIALIZER(0) \
    title, EK_ID_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.

### 5.7.2.5 **#define CTK\_ICON\_ADD(*icon*, *id*) ctk\_icon\_add((struct [ctk\\_widget](#) \*)*icon*, *id*)**

Add an icon to the desktop.

#### Parameters:

- icon* The icon to be added.
- id* The process ID of the process that owns the icon.

#### 5.7.2.6 **#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.

#### 5.7.2.7 **#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.

#### 5.7.2.8 **#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.

#### 5.7.2.9 **#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.

**5.7.2.10 #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
```

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

**5.7.2.11 #define CTK\_TEXTENTRY\_CLEAR(*e*) do {memset((*e*) → *text*, 0, (*e*) → *len*); (*e*) → *xpos* = 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.

**5.7.2.12 #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.

**5.7.2.13 #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



**5.7.2.14 #define CTK\_WIDGET\_REDRAW(widget) ctk\_widget\_redraw((struct ctk\_widget \*)widget)**

Add a widget to the redraw queue.

**Parameters:**

*widget* The widget to be redrawn.

**5.7.2.15 #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.

**5.7.2.16 #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.

**5.7.2.17 #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.

**5.7.2.18 #define CTK\_WIDGET\_TYPE(w) ((w) → type)**

Obtain the type of a widget.

**Parameters:**

*w* The widget.

**5.7.2.19 #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.

**5.7.2.20 #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.

### 5.7.3 Function Documentation

**5.7.3.1 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.

**5.7.3.2 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.

**5.7.3.3 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.

**5.7.3.4 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.

**5.7.3.5 void ctk\_dialog\_open (struct ctk\_window \* *d*)**

Open a dialog box.

**Parameters:**

*d* The dialog to be opened.

**5.7.3.6 void ctk\_icon\_add (CC\_REGISTER\_ARG struct ctk\_widget \* *icon*, ek\_id\_t *id*)**

Add an icon to the desktop.

**Parameters:**

*icon* The icon to be added.

*id* The process ID of the process that owns the icon.

Here is the call graph for this function:



**5.7.3.7 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.

**5.7.3.8 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.

**5.7.3.9 void ctk\_menu\_remove (struct [ctk\\_menu](#) \* *menu*)**

Remove a menu from the menu bar.

**Parameters:**

*menu* The menu to be removed.

**5.7.3.10 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.

**5.7.3.11 unsigned char ctk\_mode\_get (void)**

Retrieves the current CTK mode.

**Returns:**

The current CTK mode.

### 5.7.3.12 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.

### 5.7.3.13 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 widget should be added.

*widget* The widget to be added.

### 5.7.3.14 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.

### 5.7.3.15 void ctk\_window\_clear (struct ctk\_window \* *w*)

Remove all widgets from a window.

**Parameters:**

*w* The window to be cleared.

### 5.7.3.16 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.

### 5.7.3.17 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.

### 5.7.3.18 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.

### 5.7.3.19 void ctk\_window\_redraw (struct [ctk\\_window](#) \* *w*)

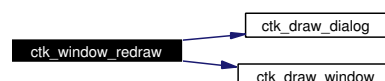
Redraw a window.

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

#### Parameters:

*w* The window to be redrawn.

Here is the call graph for this function:



## 5.7.4 Variable Documentation

### 5.7.4.1 `ek_event_t ctk_signal_hyperlink_activate`

Emitted when a hyperlink is activated.

The signal is broadcast to all listeners.

### 5.7.4.2 `ek_event_t ctk_signal_keypress`

Emitted for every key being pressed.

The key is passed as signal data.

### 5.7.4.3 `ek_event_t ctk_signal_menu_activate`

Emitted when a menu item is activated.

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

### 5.7.4.4 `ek_event_t ctk_signal_pointer_button`

Emitted when a mouse button is pressed.

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

### 5.7.4.5 `ek_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.

### 5.7.4.6 `ek_event_t ctk_signal_widget_activate`

Emitted when a widget is activated (pressed).

A pointer to the widget is passed as signal data.

### 5.7.4.7 `ek_event_t ctk_signal_widget_select`

Emitted when a widget is selected.

A pointer to the widget is passed as signal data.

### 5.7.4.8 `ek_event_t ctk_signal_window_close`

Emitted when a window is closed.

A pointer to the window is passed as signal data.

## 5.8 The CTK graphical user interface.

### 5.8.1 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

- group [CTK application functions](#)
- group [CTK device driver functions](#)

#### Functions

- void [ctk\\_init](#) (void)  
*Initializes the Contiki Toolkit.*
- void [ctk\\_mode\\_set](#) (unsigned char mode)  
*Sets the current CTK mode.*
- unsigned char [ctk\\_mode\\_get](#) (void)  
*Retrieves the current CTK mode.*
- void [ctk\\_window\\_new](#) (struct [ctk\\_window](#) \*window, unsigned char w, unsigned char h, char \*title)  
*Create a new window.*
- void [ctk\\_window\\_clear](#) (struct [ctk\\_window](#) \*w)  
*Remove all widgets from a window.*
- void [ctk\\_window\\_close](#) (struct [ctk\\_window](#) \*w)  
*Close a window if it is open.*
- void [ctk\\_window\\_redraw](#) (struct [ctk\\_window](#) \*w)  
*Redraw a window.*
- 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_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.*

## 5.8.2 Function Documentation

### 5.8.2.1 void `ctk_dialog_open` (struct `ctk_window` \* d)

Open a dialog box.

**Parameters:**

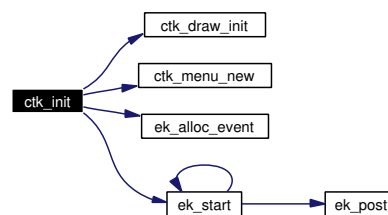
*d* The dialog to be opened.

### 5.8.2.2 void `ctk_init` (void)

Initializes the Contiki Toolkit.

This function must be called before any other CTK function, but after the initialization of the dispatcher module.

Here is the call graph for this function:



### 5.8.2.3 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.

**5.8.2.4 void ctk\_menu\_remove (struct [ctk\\_menu](#) \* *menu*)**

Remove a menu from the menu bar.

**Parameters:**

*menu* The menu to be removed.

**5.8.2.5 unsigned char ctk\_mode\_get (void)**

Retrieves the current CTK mode.

**Returns:**

The current CTK mode.

**5.8.2.6 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.

**5.8.2.7 void ctk\_window\_clear (struct [ctk\\_window](#) \* *w*)**

Remove all widgets from a window.

**Parameters:**

*w* The window to be cleared.

**5.8.2.8 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.

**5.8.2.9 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.

**5.8.2.10 void ctk\_window\_redraw (struct ctk\_window \* *w*)**

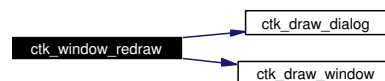
Redraw a window.

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

**Parameters:**

*w* The window to be redrawn.

Here is the call graph for this function:



## 5.9 CTK device driver functions

### 5.9.1 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.*

## 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)  
*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.*

### 5.9.1.1 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_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.

### 5.9.1.2 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()`.

## 5.9.2 Function Documentation

### 5.9.2.1 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.

**5.9.2.2 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.

**5.9.2.3 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.

**5.9.2.4 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.

**5.9.2.5 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 screen 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.

#### 5.9.2.6 void ctk\_draw\_window (struct [ctk\\_window](#) \* window, unsigned char focus, unsigned char clipy1, unsigned char clipy2)

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)



## 5.10 The uIP TCP/IP stack

### Files

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

### Modules

- group [uIP configuration functions](#)
- group [uIP initialization functions](#)
- group [uIP device driver functions](#)
- group [uIP application functions](#)
- group [uIP conversion functions](#)
- group [uIP Address Resolution Protocol](#)
- group [uIP TCP throughput booster hack](#)
- group [uIP hostname resolver functions](#)
- group [Uiparch](#)

### Data Structures

- struct [uip\\_conn](#)  
*Representation of a uIP TCP connection.*
- struct [uip\\_udp\\_conn](#)  
*Representation of a uIP UDP connection.*
- struct [uip\\_stats](#)  
*The structure holding the TCP/IP statistics that are gathered if `UIP_STATISTICS` is set to 1.*

### Functions

- void [uip\\_init](#) (void)  
*uIP initialization function.*
- [uip\\_conn](#) \* [uip\\_connect](#) (u16\_t \*ripaddr, u16\_t rport)  
*Connect to a remote host using TCP.*
- [uip\\_udp\\_conn](#) \* [uip\\_udp\\_new](#) (u16\_t \*ripaddr, u16\_t rport)  
*Set up a new UDP connection.*
- void [uip\\_unlisten](#) (u16\_t port)  
*Stop listening to the specified port.*

- void `uip_listen` (u16\_t port)  
*Start listening to the specified port.*
- u16\_t `htons` (u16\_t val)  
*Convert 16-bit quantity from host byte order to network byte order.*

## Variables

- u8\_t \* `uip_appdata`  
*Pointer to the application data in the packet buffer.*
- `uip_stats` `uip_stat`  
*The uIP TCP/IP statistics.*
- u8\_t `uip_buf` [UIP\_BUFSIZE+2]  
*The uIP packet buffer.*
- u8\_t \* `uip_appdata`  
*Pointer to the application data in the packet buffer.*
- u8\_t `uip_acc32` [4]  
*4-byte array used for the 32-bit sequence number calculations.*

### 5.10.0.7 The uIP TCP/IP stack

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

### 5.10.1 uIP introduction

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

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and does not implement UDP, but focuses on the IP, ICMP and TCP protocols. uIP is written in the C programming language.

Many other TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with

another equally limited device, e.g., when running distributed peer-to-peer services and protocols. uIP is designed to be RFC compliant in order to let the embedded devices to act as first-class network citizens. The uIP TCP/IP implementation that is not tailored for any specific application.

### 5.10.2 TCP/IP communication

The full TCP/IP suite consists of numerous protocols, ranging from low level protocols such as ARP which translates IP addresses to MAC addresses, to application level protocols such as SMTP that is used to transfer e-mail. The uIP is mostly concerned with the TCP and IP protocols and upper layer protocols will be referred to as “the application”. Lower layer protocols are often implemented in hardware or firmware and will be referred to as “the network device” that are controlled by the network device driver.

TCP provides a reliable byte stream to the upper layer protocols. It breaks the byte stream into appropriately sized segments and each segment is sent in its own IP packet. The IP packets are sent out on the network by the network device driver. If the destination is not on the physically connected network, the IP packet is forwarded onto another network by a router that is situated between the two networks. If the maximum packet size of the other network is smaller than the size of the IP packet, the packet is fragmented into smaller packets by the router. If possible, the size of the TCP segments are chosen so that fragmentation is minimized. The final recipient of the packet will have to reassemble any fragmented IP packets before they can be passed to higher layers.

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

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

In our implementations, we have implemented all RFC requirements that affect host-to-host communication. 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.

### 5.10.3 Memory management

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

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be

queued, either by the network device or by the device driver. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

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

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

#### 5.10.4 Application program interface (API)

The Application Program Interface (API) defines the way the application program interacts with the TCP/IP stack. The most commonly used API for TCP/IP is the BSD socket API which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in the intended uIP target architectures, the BSD socket interface is not suitable for our purposes.

Instead, uIP uses an event driven interface where the application is invoked in response to certain events. An application running on top of uIP is implemented as a C function that is called by uIP in response to certain events. uIP calls the application when data is received, when data has been successfully delivered to the other end of the connection, when a new connection has been set up, or when data has to be retransmitted. The application is also periodically polled for new data. The application program provides only one callback function; it is up to the application to deal with mapping different network services to different ports and connections. Because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

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

In order to reduce memory usage, uIP utilizes the fact that the application may be able to regenerate sent data and lets the application take part in retransmissions. uIP does not keep track of packet contents after they have been sent by the device driver, and uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be retransmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing

a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

#### 5.10.4.1 Application events

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

#### 5.10.4.2 The connection pointer

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

#### 5.10.4.3 Receiving data

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

#### 5.10.4.4 Sending data

When sending data, the application must check the number of available bytes in the send window and adjust the length of the data to be sent accordingly. The size of the send window is dictated by the memory configuration as well as the buffer space announced by the remote host. If no buffer space is available, the application has to defer the send and wait until later.

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.

#### 5.10.4.5 Retransmitting data

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

segment should be retransmitted, the application function is called with the `uip_rexmit()` flag set, indicating that a retransmission is required.

The application must check the `uip_rexmit()` flag and produce the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. 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.

#### 5.10.4.6 Closing connections

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

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

#### 5.10.4.7 Reporting errors

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

#### 5.10.4.8 Polling

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

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

#### 5.10.4.9 Listening ports

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

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

#### 5.10.4.10 Opening connections

New connections can be opened from within uIP by the function `uip_connect()`. This function allocates a new connection and sets a flag in the connection state which will open a TCP connection to the specified IP address and port the next time the connection is polled by uIP. The `uip_connect()` function returns a pointer

to the `uip_conn` structure for the new connection. If there are no free connection slots, the function returns `NULL`.

The function `uip_ipaddr()` may be used to pack an IP address into the two element 16-bit array used by uIP to represent IP addresses.

Two examples of usage are shown below. The first example shows how to open a connection to TCP port 8080 of the remote end of the current connection. If there are not enough TCP connection slots to allow a new connection to be opened, the `uip_connect()` function returns `NULL` and the current connection is aborted by `uip_abort()`.

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

The second example shows how to open a new connection to a specific IP address. No error checks are made in this example.

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

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

### 5.10.5 uIP device drivers

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

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

### 5.10.6 Checksum calculation

### 5.10.7 Function Documentation

#### 5.10.7.1 `u16_t htons (u16_t val)`

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

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

#### 5.10.7.2 `struct uip_conn* uip_connect (u16_t *ripaddr, u16_t port)`

Connect to a remote host using TCP.

This function is used to start a new connection to the specified port on the specified host. It allocates a new connection identifier, sets the connection to the `SYN_SENT` state and sets the retransmission timer to 0.

This will cause a TCP SYN segment to be sent out the next time this connection is periodically processed, which usually is done within 0.5 seconds after the call to `uip_connect()`.

**Note:**

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

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

```
u16_t ipaddr[2];

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

**Parameters:**

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

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

**Returns:**

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

Here is the call graph for this function:



### 5.10.7.3 void uip\_init (void)

uIP initialization function.

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

### 5.10.7.4 void uip\_listen (u16\_t port)

Start listening to the specified port.

**Note:**

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

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

**Parameters:**

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

### 5.10.7.5 struct uip\_udp\_conn\* uip\_udp\_new (u16\_t \* ripaddr, u16\_t rport)

Set up a new UDP connection.



**Parameters:**

*ripaddr* A pointer to a 4-byte structure representing the IP address of the remote host.  
*rport* The remote port number in network byte order.

**Returns:**

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

**5.10.7.6 void uip\_unlisten (u16\_t port)**

Stop listening to the specified port.

**Note:**

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

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

**Parameters:**

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

**5.10.8 Variable Documentation****5.10.8.1 u8\_t\* uip\_appdata**

Pointer to the application data in the packet buffer.

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

**5.10.8.2 u8\_t\* uip\_appdata**

Pointer to the application data in the packet buffer.

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

**5.10.8.3 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);
    hwsend(&uip_buf[UIP_LLH_LEN], 40);
    hwsend(uip_appdata, uip_len - 40 - UIP_LLH_LEN);
}
```

#### 5.10.8.4 struct `uip_stats uip_stat`

The uIP TCP/IP statistics.

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

## 5.11 uIP configuration functions

### 5.11.1 Detailed Description

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

#### Defines

- #define `uip_sethostaddr(addr)`  
*Set the IP address of this host.*
- #define `uip_gethostaddr(addr)`  
*Get the IP address of this host.*
- #define `uip_setdraddr(addr)`  
*Set the default router's IP address.*
- #define `uip_setnetmask(addr)`  
*Set the netmask.*
- #define `uip_getdraddr(addr)`  
*Get the default router's IP address.*
- #define `uip_getnetmask(addr)`  
*Get the netmask.*
- #define `uip_setethaddr(eaddr)`  
*Specify the Ethernet MAC address.*

### 5.11.2 Define Documentation

#### 5.11.2.1 #define `uip_getdraddr(addr)`

Get the default router's IP address.

##### Parameters:

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

#### 5.11.2.2 #define `uip_gethostaddr(addr)`

Get the IP address of this host.

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

##### Parameters:

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

### 5.11.2.3 **#define uip\_getnetmask(addr)**

Get the netmask.

**Parameters:**

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

### 5.11.2.4 **#define uip\_setdraddr(addr)**

Set the default router's IP address.

**Parameters:**

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

### 5.11.2.5 **#define uip\_setethaddr(eaddr)**

Specify the Ethernet MAC address.

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

**Note:**

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

**Parameters:**

*eaddr* A pointer to a struct [uip\\_eth\\_addr](#) containing the Ethernet MAC address of the Ethernet card.

### 5.11.2.6 **#define uip\_sethostaddr(addr)**

Set the IP address of this host.

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

**Parameters:**

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

### 5.11.2.7 **#define uip\_setnetmask(addr)**

Set the netmask.

**Parameters:**

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

## 5.12 uIP initialization functions

### 5.12.1 Detailed Description

The uIP initialization functions are used for booting uIP.

#### Functions

- void `uip_init` (void)  
*uIP initialization function.*

### 5.12.2 Function Documentation

#### 5.12.2.1 void `uip_init` (void)

uIP initialization function.

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

## 5.13 uIP device driver functions

### 5.13.1 Detailed Description

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

#### Defines

- `#define uip_input()`  
*Process an incoming packet.*
- `#define uip_periodic(conn)`  
*Periodic processing for a connection identified by its number.*
- `#define uip_periodic_conn(conn)`  
*Periodic processing for a connection identified by a pointer to its structure.*
- `#define uip_udp_periodic(conn)`  
*Periodic processing for a UDP connection identified by its number.*
- `#define uip_udp_periodic_conn(conn)`  
*Periodic processing for a UDP connection identified by a pointer to its structure.*

#### Variables

- `u8_t uip_buf [UIP_BUFSIZE+2]`  
*The uIP packet buffer.*

### 5.13.2 Define Documentation

#### 5.13.2.1 `#define uip_input()`

Process an incoming packet.

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

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

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

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

**Note:**

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

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

**5.13.2.2 #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 connection, and should be called when the periodic uIP timer goes off. It should be called for every connection, regardless of whether they are open or closed.

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

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

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

**Note:**

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

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

**Parameters:**

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

### 5.13.2.3 #define uip\_periodic\_conn(conn)

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

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

**Parameters:**

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

### 5.13.2.4 #define uip\_udp\_periodic(conn)

Periodic processing for a UDP connection identified by its number.

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

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

**Note:**

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

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

**Parameters:**

*conn* The number of the UDP connection to be processed.

### 5.13.2.5 #define uip\_udp\_periodic\_conn(conn)

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

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

**Parameters:**

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

## 5.13.3 Variable Documentation

### 5.13.3.1 u8\_t uip\_buf[UIP\_BUFSIZE+2]

The uIP packet buffer.



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

**Note:**

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

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

## 5.14 uIP application functions

### 5.14.1 Detailed Description

Functions used by an application running on top of uIP.

#### Defines

- #define `uip_send(data, len)`  
*Send data on the current connection.*
- #define `uip_datalen()`  
*The length of any incoming data that is currently available (if available) in the `uip_appdata` buffer.*
- #define `uip_urgdatalen()`  
*The length of any out-of-band data (urgent data) that has arrived on the connection.*
- #define `uip_close()`  
*Close the current connection.*
- #define `uip_abort()`  
*Abort the current connection.*
- #define `uip_stop()`  
*Tell the sending host to stop sending data.*
- #define `uip_stopped(conn)`  
*Find out if the current connection has been previously stopped with `uip_stop()`.*
- #define `uip_restart()`  
*Restart the current connection, if it has previously been stopped with `uip_stop()`.*
- #define `uip_udpconnection()`  
*Is the current connection a UDP connection?*
- #define `uip_newdata()`  
*Is new incoming data available?*
- #define `uip_acked()`  
*Has previously sent data been acknowledged?*
- #define `uip_connected()`  
*Has the connection just been connected?*
- #define `uip_closed()`  
*Has the connection been closed by the other end?*
- #define `uip_aborted()`  
*Has the connection been aborted by the other end?*

- `#define uip_timedout()`  
*Has the connection timed out?*
- `#define uip_rexmit()`  
*Do we need to retransmit previously data?*
- `#define uip_poll()`  
*Is the connection being polled by uIP?*
- `#define uip_initialmss()`  
*Get the initial maxium segment size (MSS) of the current connection.*
- `#define uip_mss()`  
*Get the current maxium segment size that can be sent on the current connection.*
- `#define uip_udp_remove(conn)`  
*Removed a UDP connection.*
- `#define uip_udp_bind(conn, port)`  
*Bind a UDP connection to a local port.*
- `#define uip_udp_send(len)`  
*Send a UDP datagram of length len on the current connection.*

## Functions

- `void uip_listen (u16_t port)`  
*Start listening to the specified port.*
- `void uip_unlisten (u16_t port)`  
*Stop listening to the specified port.*
- `uip_conn * uip_connect (u16_t *ripaddr, u16_t port)`  
*Connect to a remote host using TCP.*
- `uip_udp_conn * uip_udp_new (u16_t *ripaddr, u16_t rport)`  
*Set up a new UDP connection.*

### 5.14.2 Define Documentation

#### 5.14.2.1 `#define uip_abort()`

Abort the current connection.

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

**5.14.2.2 #define uip\_aborted()**

Has the connection been aborted by the other end?

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

**5.14.2.3 #define uip\_acked()**

Has previously sent data been acknowledged?

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

**5.14.2.4 #define uip\_close()**

Close the current connection.

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

**5.14.2.5 #define uip\_closed()**

Has the connection been closed by the other end?

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

**5.14.2.6 #define uip\_connected()**

Has the connection just been connected?

Reduces to non-zero if the current connection has been connected to a remote host. This will happen both if the connection has been actively opened (with [uip\\_connect\(\)](#)) or passively opened (with [uip\\_listen\(\)](#)).

**5.14.2.7 #define uip\_datalen()**

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

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

**5.14.2.8 #define uip\_mss()**

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

The current maximum segment size that can be sent on the connection is computed from the receiver's window and the MSS of the connection (which also is available by calling [uip\\_initialmss\(\)](#)).

**5.14.2.9 #define uip\_newdata()**

Is new incoming data available?

Will reduce to non-zero if there is new data for the application present at the uip\_appdata pointer. The size of the data is available through the uip\_len variable.

**5.14.2.10 #define uip\_poll()**

Is the connection being polled by uIP?

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

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

**5.14.2.11 #define uip\_restart()**

Restart the current connection, if it has previously been stopped with [uip\\_stop\(\)](#).

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

**5.14.2.12 #define uip\_rexmit()**

Do we need to retransmit previously data?

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

**5.14.2.13 #define uip\_send(data, len)**

Send data on the current connection.

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

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

**Note:**

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

**Parameters:**

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

*len* The maximum amount of data bytes to be sent.

**5.14.2.14 #define uip\_stop()**

Tell the sending host to stop sending data.

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

**5.14.2.15 #define uip\_timeout()**

Has the connection timed out?

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

**5.14.2.16 #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.

**5.14.2.17 #define uip\_udp\_remove(conn)**

Removed a UDP connection.

**Parameters:**

*conn* A pointer to the [uip\\_udp\\_conn](#) structure for the connection.

**5.14.2.18 #define uip\_udp\_send(len)**

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

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

**Parameters:**

*len* The length of the data in the uip\_buf buffer.

**5.14.2.19 #define uip\_udpconnection()**

Is the current connection a UDP connection?

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

**5.14.2.20 #define uip\_urgdatalen()**

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

**Note:**

The configuration parameter UIP\_URGDATA must be set for this function to be enabled.

### 5.14.3 Function Documentation

#### 5.14.3.1 struct `uip_conn*` `uip_connect (u16_t * ripaddr, u16_t port)`

Connect to a remote host using TCP.

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

**Note:**

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

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

```
u16_t ipaddr[2];  
  
uip_ipaddr(ipaddr, 192,168,1,2);  
uip_connect(ipaddr, HTONS(80));
```

**Parameters:**

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

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

**Returns:**

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

Here is the call graph for this function:



#### 5.14.3.2 void `uip_listen (u16_t port)`

Start listening to the specified port.

**Note:**

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

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

**Parameters:**

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

### 5.14.3.3 `struct uip_udp_conn* uip_udp_new (u16_t * ripaddr, u16_t rport)`

Set up a new UDP connection.

**Parameters:**

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

*rport* The remote port number in network byte order.

**Returns:**

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

### 5.14.3.4 `void uip_unlisten (u16_t port)`

Stop listening to the specified port.

**Note:**

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

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

**Parameters:**

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



## 5.15 uIP conversion functions

### 5.15.1 Detailed Description

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

#### Defines

- #define `uip_ipaddr(addr, addr0, addr1, addr2, addr3)`  
*Pack an IP address into a 4-byte array which is used by uIP to represent IP addresses.*
- #define `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

- `u16_t htons (u16_t val)`  
*Convert 16-bit quantity from host byte order to network byte order.*
- unsigned char `uiplib_ipaddrconv (char *addrstr, unsigned char *addr)`  
*Convert a textual representation of an IP address to a numerical representation.*

## 5.15.2 Define Documentation

### 5.15.2.1 #define HTONS(n)

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

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

### 5.15.2.2 #define uip\_ipaddr(addr, addr0, addr1, addr2, addr3)

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

Example:

```
u16_t ipaddr[2];

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

#### Parameters:

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

### 5.15.2.3 #define uip\_ipaddr1(addr)

Pick the first octet of an IP address.

Picks out the first octet of an IP address.

Example:

```
u16_t ipaddr[2];
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.

### 5.15.2.4 #define uip\_ipaddr2(addr)

Pick the second octet of an IP address.

Picks out the second octet of an IP address.

Example:

```
u16_t ipaddr[2];
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.

#### 5.15.2.5 #define uip\_ipaddr3(addr)

Pick the third octet of an IP address.

Picks out the third octet of an IP address.

Example:

```
u16_t ipaddr[2];
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.

#### 5.15.2.6 #define uip\_ipaddr4(addr)

Pick the fourth octet of an IP address.

Picks out the fourth octet of an IP address.

Example:

```
u16_t ipaddr[2];
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.

#### 5.15.2.7 #define uip\_ipaddr\_cmp(addr1, addr2)

Compare two IP addresses.

Compares two IP addresses.

Example:

```
u16_t ipaddr1[2], ipaddr2[2];

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.

#### 5.15.2.8 #define uip\_ipaddr\_copy(dest, src)

Copy an IP address to another IP address.

Copies an IP address from one place to another.

Example:

```

ul6_t ipaddr1[2], ipaddr2[2];

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.

**5.15.2.9 #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:

```

ul6_t ipaddr1[2], ipaddr2[2], netmask[2];

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.

**5.15.2.10 #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:

```

ul6_t ipaddr1[2], ipaddr2[2], mask[2];

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.

### 5.15.3 Function Documentation

#### 5.15.3.1 u16\_t htons (u16\_t val)

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

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

#### 5.15.3.2 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.

## 5.16 uIP Address Resolution Protocol

### 5.16.1 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\\_addr](#)  
*Representation of a 48-bit Ethernet address.*
- struct [uip\\_eth\\_hdr](#)  
*The Ethernet header.*

### Functions

- void [uip\\_arp\\_init](#) (void)  
*Initialize the ARP module.*
- void [uip\\_arp\\_arpin](#) (void)  
*ARP processing for incoming ARP packets.*
- void [uip\\_arp\\_out](#) (void)  
*Prepend Ethernet header to an outbound IP packet and see if we need to send out an ARP request.*
- void [uip\\_arp\\_timer](#) (void)  
*Periodic ARP processing function.*

## 5.16.2 Function Documentation

### 5.16.2.1 void uip\_arp\_arpin (void)

ARP processing for incoming ARP packets.

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

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

This function expects an ARP packet with a prepended Ethernet header in the uip\_buf[] buffer, and the length of the packet in the global variable uip\_len.

### 5.16.2.2 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.

### 5.16.2.3 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.

## 5.17 uIP TCP throughput booster hack

### 5.17.1 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.*

### 5.17.2 Function Documentation

#### 5.17.2.1 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.



## 5.18 uIP hostname resolver functions

### 5.18.1 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 signal `resolv_signal_found` is emitted when a hostname has been resolved. The signal is emitted to all processes listening for the signal, and 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.*
- u16\_t \* `resolv_getserver` (void)  
*Obtain the currently configured DNS server.*
- void `resolv_conf` (u16\_t \*dnsserver)  
*Configure a DNS server.*
- void `resolv_init` (char \*arg)  
*Initialize the resolver.*

### Variables

- ek\_event\_t `resolv_event_found`  
*Signal that is sent when a DNS name has been resolved.*

### 5.18.2 Function Documentation

#### 5.18.2.1 void `resolv_conf` (u16\_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.

Here is the call graph for this function:



#### 5.18.2.2 `u16_t* resolv_getserver (void)`

Obtain the currently configured DNS server.

**Returns:**

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

#### 5.18.2.3 `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.

#### 5.18.2.4 `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.

## 5.19 Socket library

### 5.19.1 Detailed Description

The socket 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 socket library are executed in a sequential fashion and does not have to be implemented as explicit state machines.

Sockets only work with TCP connections.

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

The socket 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 socket runs as a protothread, the socket has to be started with a call to [SOCKET\\_BEGIN\(\)](#) at the start of the function in which the socket is used. Similarly, the socket protothread can be terminated by a call to [SOCKET\\_EXIT\(\)](#).

The example code below illustrates how to use the socket library. The program implements a simple SMTP client that sends a short email. The program is divided into two functions, one uIP event handler (`smtp_uipcall()`) and one function that runs the socket protothread and performs the SMTP communication (`smtp_socketthread()`).

An SMTP connection is represented by a `smtp_state` structure containing a struct socket and a small input buffer. The input buffer only needs to be 3 bytes long to accomodate the 3 byte status codes used by SMTP. Connection structures can be allocated from the memory buffer called `connections`, which is declared with the [MEMB\(\)](#) macro.

The convenience macro `SEND_STRING()` is defined in order to simplify the code, as it mostly involves sending strings.

The function `smtp_socketthread()` is declared as a protothread using the [PT\\_THREAD\(\)](#) macro. The [SOCKET\\_BEGIN\(\)](#) call at the first line of the `smtp_socketthread()` function starts the protothread. SMTP specifies that the server will start with sending a welcome message that should include the status code 220 if the server is ready to accept messages. Therefore, the `smtp_socketthread()` first calls [SOCKET\\_READTO\(\)](#) to read all incoming data up to the first newline. If the status code was anything else but 220, the socket is closed and the socket's protothread is terminated with the call to [SOCKET\\_CLOSE\\_EXIT\(\)](#).

If the connection is accepted by the server, `smtp_socketthread()` continues with sending the HELO message. If this gets a positive reply (a status code beginning with a 2), the protothread moves on with the rest of the SMTP procedure. Finally, after all headers and data is sent, the program sends a QUIT before it finally closes the socket and exits the socket's protothread.

```
#include <string.h>

#include "socket.h"
#include "memb.h"

struct smtp_state {
    struct socket socket;
    char inputbuffer[3];
};

MEMB(connections, sizeof(struct smtp_state), 2);
```

```

#define SEND_STRING(s, str) SOCKET_SEND(s, str, strlen(str))

static
PT_THREAD(smtp_socketthread(struct smtp_state *s))
{
    SOCKET_BEGIN(&s->socket);

    SOCKET_READTO(&s->socket, '\n');

    if(strncmp(s->inputbuffer, "220", 3) != 0) {
        SOCKET_CLOSE_EXIT(&s->socket);
    }

    SEND_STRING(&s->socket, "HELO contiki.example.com\r\n");

    SOCKET_READTO(&s->socket, '\n');
    if(s->inputbuffer[0] != '2') {
        SOCKET_CLOSE_EXIT(&s->socket);
    }

    SEND_STRING(&s->socket, "MAIL FROM: contiki@example.com\r\n");

    SOCKET_READTO(&s->socket, '\n');
    if(s->inputbuffer[0] != '2') {
        SOCKET_CLOSE_EXIT(&s->socket);
    }

    SEND_STRING(&s->socket, "RCPT TO: contiki@example.com\r\n");

    SOCKET_READTO(&s->socket, '\n');
    if(s->inputbuffer[0] != '2') {
        SOCKET_CLOSE_EXIT(&s->socket);
    }

    SEND_STRING(&s->socket, "DATA\r\n");

    SOCKET_READTO(&s->socket, '\n');
    if(s->inputbuffer[0] != '3') {
        SOCKET_CLOSE_EXIT(&s->socket);
    }

    SEND_STRING(&s->socket, "To: contiki@example.com\r\n");
    SEND_STRING(&s->socket, "From: contiki@example.com\r\n");
    SEND_STRING(&s->socket, "Subject: Example\r\n");

    SEND_STRING(&s->socket, "A test message from Contiki.\r\n");

    SEND_STRING(&s->socket, "\r\n.\r\n");

    SOCKET_READTO(&s->socket, '\n');
    if(s->inputbuffer[0] != '2') {
        SOCKET_CLOSE_EXIT(&s->socket);
    }

    SEND_STRING(&s->socket, "QUIT\r\n");

    SOCKET_END(&s->socket);
}

void
smtp_uipcall(void *state)
{
    struct smtp_state *s = (struct smtp_state *)state;

    if(uip_closed() || uip_aborted() || uip_timedout()) {
        memb_free(&connections, s);
    } else if(uip_connected()) {

```

```
    SOCKET_INIT(s, s->inputbuffer, sizeof(s->inputbuffer));  
  } else {  
    smtp_socketthread(s);  
  }  
}
```

## Files

- file [socket.h](#)  
*Socket library header file.*

## Data Structures

- struct [socket](#)  
*The representation of a socket.*

## Defines

- #define [SOCKET\\_INIT](#)(socket, buffer, buffersize)  
*Initialize a socket.*
- #define [SOCKET\\_BEGIN](#)(socket)  
*Start the socket protothread in a function.*
- #define [SOCKET\\_SEND](#)(socket, data, datalen)  
*Send data.*
- #define [SOCKET\\_CLOSE](#)(socket)  
*Close a socket.*
- #define [SOCKET\\_READTO](#)(socket, c)  
*Read data up to a specified character.*
- #define [SOCKET\\_DATALEN](#)(socket)  
*The length of the data that was previously read.*
- #define [SOCKET\\_EXIT](#)(socket)  
*Exit the socket's protothread.*
- #define [SOCKET\\_CLOSE\\_EXIT](#)(socket)  
*Close a socket and exit the socket's protothread.*
- #define [SOCKET\\_NEWDATA](#)(socket)  
*Check if new data has arrived on a socket.*
- #define [SOCKET\\_WAIT\\_UNTIL](#)(socket, condition)  
*Wait until data arrives or until a condition is true.*

## 5.19.2 Define Documentation

### 5.19.2.1 #define SOCKET\_BEGIN([socket](#))

Start the socket protothread in a function.

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

**Parameters:**

*socket* (struct socket \*) A pointer to the socket to be started.

### 5.19.2.2 #define SOCKET\_CLOSE([socket](#))

Close a socket.

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

**Parameters:**

*socket* (struct socket \*) A pointer to the socket that is to be closed.

### 5.19.2.3 #define SOCKET\_CLOSE\_EXIT([socket](#))

Close a socket and exit the socket's protothread.

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

**Parameters:**

*socket* (struct socket \*) A pointer to the socket.

### 5.19.2.4 #define SOCKET\_DATALEN([socket](#))

The length of the data that was previously read.

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

**Parameters:**

*socket* (struct socket \*) A pointer to the socket holding the data.

### 5.19.2.5 #define SOCKET\_EXIT([socket](#))

Exit the socket's protothread.

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

**See also:**

[SOCKET\\_CLOSE\\_EXIT\(\)](#)

**Parameters:**

*socket* (struct socket \*) A pointer to the socket.

#### 5.19.2.6 #define SOCKET\_INIT(socket, buffer, buffersize)

Initialize a socket.

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

**Parameters:**

*socket* (struct socket \*) A pointer to the socket to be initialized

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

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

#### 5.19.2.7 #define SOCKET\_NEWDATA(socket)

Check if new data has arrived on a socket.

This macro is used in conjunction with the [SOCKET\\_WAIT\\_UNTIL\(\)](#) macro to check if data has arrived on a socket.

**Parameters:**

*socket* (struct socket \*) A pointer to the socket.

#### 5.19.2.8 #define SOCKET\_READTO(socket, 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 [SOCKET\\_INIT\(\)](#). Data is only read until the specified character appears in the data stream.

**Parameters:**

*socket* (struct socket \*) A pointer to the socket from which data should be read.

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

#### 5.19.2.9 #define SOCKET\_SEND(socket, data, datalen)

Send data.

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

**Parameters:**

*socket* (struct socket \*) A pointer to the socket 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.

### 5.19.2.10 `#define SOCKET_WAIT_UNTIL(socket, condition)`

Wait until data arrives or until a condition is true.

This macro blocks the protothread until new data arrives on the socket or until the specified condition is true. After the protothread unblocks, the macro `SOCKET_NEWDATA()` must be used to check whether the protothread unblocked because of new data arrived or (only) if the condition was true.

Typically, this macro is used as follows:

```
PT_THREAD(thread(struct socket *s, struct timer *t))
{
    SOCKET_BEGIN(s);

    SOCKET_WAIT_UNTIL(s, timer_expired(t) || something_else());

    if(SOCKET_NEWDATA(s)) {
        SOCKET_READTO(s, '\n');
    } else {
        handle_timed_out(s);
    }

    SOCKET_END(s);
}
```

#### Parameters:

*socket* (struct socket \*) A pointer to the socket.

*condition* The condition to wait for.



## 5.20 Memory block management functions

### 5.20.1 Detailed Description

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

A set of memory blocks is statically declared with the [MEMB\(\)](#) macro. Memory blocks are allocated from the declared memory by the [memb\\_alloc\(\)](#) function, and are deallocated with the [memb\\_free\(\)](#) function.

**Note:**

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

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

```
#include "memb.h"

MEMB(cmem, 20, 8);

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

    memb_init(&cmem);

    ptr = memb_alloc(&cmem);

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

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

### Files

- file [memb.h](#)  
*Memory block allocation routines.*
- file [memb.c](#)  
*Memory block allocation routines.*

### Defines

- #define [MEMB](#)(name, size, num)  
*Declare a memory block.*

## Functions

- void [memb\\_init](#) (struct memb\_blocks \*m)  
*Initialize a memory block that was declared with [MEMB\(\)](#).*
- char \* [memb\\_alloc](#) (struct memb\_blocks \*m)  
*Allocate a memory block from a block of memory declared with [MEMB\(\)](#).*
- char [memb\\_ref](#) (struct memb\_blocks \*m, char \*ptr)  
*Increase the reference count for a memory chunk.*
- char [memb\\_free](#) (struct memb\_blocks \*m, void \*ptr)  
*Deallocate a memory block from a memory block previously declared with [MEMB\(\)](#).*

### 5.20.2 Define Documentation

#### 5.20.2.1 #define MEMB(name, size, num)

##### Value:

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

Declare a memory block.

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

Example:

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

##### Parameters:

- name** The name of the memory block (later used with [memb\\_init\(\)](#), [memb\\_alloc\(\)](#) and [memb\\_free\(\)](#)).
- size** The size of each memory chunk, in bytes.
- num** The total number of memory chunks in the block.

### 5.20.3 Function Documentation

#### 5.20.3.1 char \* memb\_alloc (struct memb\_blocks \* m)

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

##### Parameters:

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

**5.20.3.2 char memb\_free (struct memb\_blocks \* *m*, void \* *ptr*)**

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

**Parameters:**

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

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

**Returns:**

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

**5.20.3.3 void memb\_init (struct memb\_blocks \* *m*)**

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

**Parameters:**

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

**5.20.3.4 char memb\_ref (struct memb\_blocks \* *m*, char \* *ptr*)**

Increase the reference count for a memory chunk.

**Note:**

No sanity checks are currently made.

**Parameters:**

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

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

**Returns:**

The new reference count.

## 5.21 Preemptive multi-threading

### 5.21.1 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 consists 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

- group [Architecture support for multi-threading](#)
- group [Multi-threading library convenience functions](#)

#### 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)  
*Start executing a thread.*
- void `mt_exec_event` (struct mt\_thread \*thread, ek\_event\_t s, ek\_data\_t data)  
*Post an event to a thread.*
- void `mt_yield` (void)  
*Voluntarily give up the processor.*
- void `mt_post` (ek\_id\_t id, ek\_event\_t s, ek\_data\_t data)  
*Emit a signal to another process.*
- void `mt_wait` (ek\_event\_t \*s, ek\_data\_t \*data)  
*Block and wait for an event to occur.*
- void `mt_exit` (void)  
*Exit a thread.*

## 5.21.2 Function Documentation

### 5.21.2.1 void mt\_exec (struct mt\_thread \* *thread*)

Start executing a thread.

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

**Note:**

The thread 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.

Here is the call graph for this function:



### 5.21.2.2 void mt\_exec\_event (struct mt\_thread \* *thread*, ek\_event\_t *s*, ek\_data\_t *data*)

Post an event to a thread.

This function posts an event to a thread. The thread will be scheduled if the thread currently is waiting for the posted event number. If the thread is not waiting for the event, this function does nothing.

**Note:**

The thread 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.

*s* The event that is posted to the thread.

Here is the call graph for this function:



### 5.21.2.3 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.

Here is the call graph for this function:



#### 5.21.2.4 void mt\_post (ek\_id\_t id, ek\_event\_t s, ek\_data\_t data)

Emit a signal to another process.

This function is called by a running thread and will emit a signal to another Contiki process. This will cause the currently executing thread to yield.

##### Parameters:

*s* The signal to be emitted.

*data* A pointer to a message that is to be delivered together with the signal.

*id* The process ID of the receiver of the signal, or EK\_ID\_ALL for a broadcast signal.

Here is the call graph for this function:



#### 5.21.2.5 void mt\_start (struct mt\_thread \* thread, void(\* function)(void \*), 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.

Here is the call graph for this function:



#### 5.21.2.6 void mt\_wait (ek\_event\_t \* s, ek\_data\_t \* data)

Block and wait for an event to occur.

This function can be called by a running thread in order to block and wait for an event. The function returns when an event has occurred. The event number and the associated data are placed in the variables pointed to by the function arguments.

Here is the call graph for this function:



#### 5.21.2.7 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.

Here is the call graph for this function:



## 5.22 Architecture support for multi-threading

### 5.22.1 Detailed Description

The Contiki multi-threading library requires some architecture specific support for setting up and switching stacks. This support requires three 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, and `mtarch_yield()`, which restores the kernel stack from a thread's stack. Additionally, two functions for controlling the preemption (if any) must be implemented: `mtarch_preemption_start()` and `mtarch_preemption_stop()`. 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 initialization of timer interrupts, or any other mechanisms required for correct operation of the architecture specific support functions.

### 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_yield` (void)

*Yield the processor.*

- void `mtarch_exec` (struct `mtarch_thread` \*thread)

*Start executing a thread.*

### 5.22.2 Function Documentation

#### 5.22.2.1 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.



#### 5.22.2.2 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 preemption timers or other architecture specific mechanisms required for the operation of the library.

#### 5.22.2.3 void mtarch\_start (struct mtarch\_thread \* *thread*, void(\* *function*)(void \**data*), 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.

#### 5.22.2.4 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.

## 5.23 Multi-threading library convenience functions

### 5.23.1 Detailed Description

The Contiki multi-threading library has an interface that might be hard to use. Therefore, the mtp module provides a simpler interface.

Example:

```
static void
example_thread_code(void *data)
{
    while(1) {
        printf("Test\n");
        mt_yield();
    }
}
MTP(example_thread, "Example thread", p1, t1, t1_idle);

int
main(int argc, char *argv[])
{
    mtp_start(&example_thread, example_thread_code, NULL);
}
```

### Defines

- #define **MTP**(thread, proc, name)  
*Declare a thread.*

### Functions

- void **mtp\_start** (struct mtp\_thread \*t, void(\*function)(void \*), void \*data)  
*Start a thread.*

### 5.23.2 Define Documentation

#### 5.23.2.1 #define MTP(thread, proc, name)

Declare a thread.

This macro is used to conveniently declare a thread, and the process in which the thread should execute. The names of the variables provided to the macro should be chosen to be unique within the file that the thread is used.

Example:

```
MTP(example_thread, "Example thread", p1, t1, t1_idle);
```

#### Parameters:

**thread** The name of the thread.

**name** A string that specifies the user-visible name of the process in which the thread will run.

*name\_p* The name of the variable holding the process' state.

*name\_t* The name of the variable holding the threads' state.

*name\_idle* The name of the function that is to execute the threads' code.

### 5.23.3 Function Documentation

#### 5.23.3.1 void mtp\_start (struct mtp\_thread \* *t*, void(\**function*)(void \*), void \* *data*)

Start a thread.

This function starts the process in which the thread is to run, and also sets up the thread to run within the process. The function should be passed variable names declared with the [MTP\(\)](#) macro.

Example:

```
mtp_start(&t, example_thread_code, NULL);
```

##### Parameters:

*t* A pointer to a thread structure previously declared with [MTP\(\)](#).

*function* A pointer to the function that the thread should start executing.

*data* A pointer that the function should be passed when first invoked.

Here is the call graph for this function:



## 5.24 System signals

### Variables

- `ek_event_t ctk_signal_keypress`  
*Emitted for every key being pressed.*
- `ek_event_t ctk_signal_widget_activate`  
*Emitted when a widget is activated (pressed).*
- `ek_event_t ctk_signal_button_activate`  
*Same as `ctk_signal_widget_activate`.*
- `ek_event_t ctk_signal_widget_select`  
*Emitted when a widget is selected.*
- `ek_event_t ctk_signal_button_hover`  
*Same as `ctk_signal_widget_select`.*
- `ek_event_t ctk_signal_hyperlink_activate`  
*Emitted when a hyperlink is activated.*
- `ek_event_t ctk_signal_hyperlink_hover`  
*Same as `ctk_signal_widget_select`.*
- `ek_event_t ctk_signal_menu_activate`  
*Emitted when a menu item is activated.*
- `ek_event_t ctk_signal_window_close`  
*Emitted when a window is closed.*
- `ek_event_t ctk_signal_pointer_move`  
*Emitted when the mouse pointer is moved.*
- `ek_event_t ctk_signal_pointer_button`  
*Emitted when a mouse button is pressed.*
- `ek_event_t resolv_event_found`  
*Signal that is sent when a DNS name has been resolved.*

### 5.24.1 Variable Documentation

#### 5.24.1.1 `ek_event_t ctk_signal_hyperlink_activate`

Emitted when a hyperlink is activated.

The signal is broadcast to all listeners.

**5.24.1.2 `ek_event_t ctk_signal_keypress`**

Emitted for every key being pressed.

The key is passed as signal data.

**5.24.1.3 `ek_event_t ctk_signal_menu_activate`**

Emitted when a menu item is activated.

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

**5.24.1.4 `ek_event_t ctk_signal_pointer_button`**

Emitted when a mouse button is pressed.

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

**5.24.1.5 `ek_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.

**5.24.1.6 `ek_event_t ctk_signal_widget_activate`**

Emitted when a widget is activated (pressed).

A pointer to the widget is passed as signal data.

**5.24.1.7 `ek_event_t ctk_signal_widget_select`**

Emitted when a widget is selected.

A pointer to the widget is passed as signal data.

**5.24.1.8 `ek_event_t ctk_signal_window_close`**

Emitted when a window is closed.

A pointer to the window is passed as signal data.

## 5.25 Uiparch

### Variables

- `u8_t uip_acc32` [4]

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

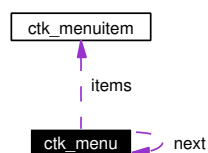
## Chapter 6

# Contiki 1.2-devel0 Data Structure Documentation

### 6.1 ctk\_menu Struct Reference

```
#include <ctk.h>
```

Collaboration diagram for ctk\_menu:



#### 6.1.1 Detailed Description

Representation of an individual menu.

#### 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\_CONF\_MAXMENUITEMS]  
*The array which contains all the menu items.*

## 6.1.2 Field Documentation

### 6.1.2.1 unsigned char `ctk_menu::titlelen`

The length of the title in characters.

Cached for speed reasons.



## 6.2 ctk\_menuitem Struct Reference

```
#include <ctk.h>
```

### 6.2.1 Detailed Description

Representation of an individual menu item.

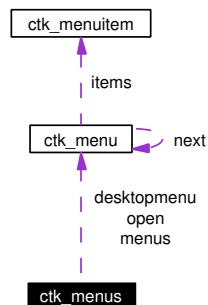
#### Data Fields

- char \* [title](#)  
*The menu items text.*
- unsigned char [titlelen](#)  
*The length of the item text, cached for speed.*

## 6.3 ctk\_menus Struct Reference

```
#include <ctk.h>
```

Collaboration diagram for ctk\_menus:



### 6.3.1 Detailed Description

Representation of the menu bar.

#### 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).

### 6.3.2 Field Documentation

#### 6.3.2.1 struct `ctk_menu*` `ctk_menus::open`

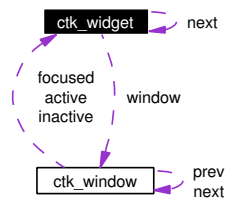
The currently open menu, if any.

If all menus are closed, this item is NULL:

## 6.4 ctk\_widget Struct Reference

```
#include <ctk.h>
```

Collaboration diagram for ctk\_widget:



### 6.4.1 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.

### Data Fields

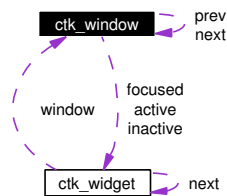
- `ctk_widget * next`  
The next widget in the linked list of widgets that is contained in the `ctk_window` structure.
- `ctk_window * window`  
The window in which the widget is contained.
- unsigned char `x`  
The *x* position of the widget within the containing window, in character coordinates.
- 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.*

## 6.5 ctk\_window Struct Reference

```
#include <ctk.h>
```

Collaboration diagram for ctk\_window:



### 6.5.1 Detailed Description

Representation of a CTK window.

For the CTK, each window is represented 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 are contained in the window as well as a pointer to the currently selected widget.

### 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.*
- `ek_id_t 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.*

## 6.5.2 Field Documentation

### 6.5.2.1 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.

### 6.5.2.2 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.

### 6.5.2.3 **ek\_id\_t ctk\_window::owner**

The process that owns the window.

This process will be the receiver of all CTK signals that pertain to this window.

### 6.5.2.4 **char\* ctk\_window::title**

The title of the window.

Used for constructing the "Dekstop" menu.

## 6.6 dsc Struct Reference

```
#include <dsc.h>
```

### 6.6.1 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.

### 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.*

### 6.6.2 Field Documentation

#### 6.6.2.1 `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.

## 6.7 socket Struct Reference

```
#include <socket.h>
```

### 6.7.1 Detailed Description

The representation of a socket.

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



## 6.8 uip\_conn Struct Reference

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

### 6.8.1 Detailed Description

Representation of a uIP TCP connection.

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

### Data Fields

- u16\_t [ripaddr](#) [2]  
*The IP address of the remote host.*
- u16\_t [lport](#)  
*The local TCP port, in network byte order.*
- u16\_t [rport](#)  
*The local remote TCP port, in network byte order.*
- u8\_t [rcv\\_nxt](#) [4]  
*The sequence number that we expect to receive next.*
- u8\_t [snd\\_nxt](#) [4]  
*The sequence number that was last sent by us.*
- u16\_t [len](#)  
*Length of the data that was previously sent.*
- u16\_t [mss](#)  
*Current maximum segment size for the connection.*
- u16\_t [initialmss](#)  
*Initial maximum segment size for the connection.*
- u8\_t [sa](#)  
*Retransmission time-out calculation state variable.*
- u8\_t [sv](#)  
*Retransmission time-out calculation state variable.*
- u8\_t [rto](#)  
*Retransmission time-out.*
- u8\_t [tcpstateflags](#)

*TCP state and flags.*

- `u8_t timer`

*The retransmission timer.*

- `u8_t nrtx`

*The number of retransmissions for the last segment sent.*

- `u8_t appstate [UIP_APPSTATE_SIZE]`

*The application state.*

## 6.9 uip\_eth\_addr Struct Reference

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

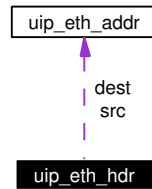
### 6.9.1 Detailed Description

Representation of a 48-bit Ethernet address.

## 6.10 uip\_eth\_hdr Struct Reference

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

Collaboration diagram for uip\_eth\_hdr:



### 6.10.1 Detailed Description

The Ethernet header.

## 6.11 uip\_stats Struct Reference

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

### 6.11.1 Detailed Description

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

#### Data Fields

- struct {
    - uip\_stats\_t **drop**
    - uip\_stats\_t **recv**
    - uip\_stats\_t **sent**
    - uip\_stats\_t **vherr**
    - uip\_stats\_t **hblenerr**
    - uip\_stats\_t **lblenerr**
    - uip\_stats\_t **fragerr**
    - uip\_stats\_t **chkerr**
    - uip\_stats\_t **protoerr**
- } [ip](#)

*IP statistics.*

- struct {
    - uip\_stats\_t **drop**
    - uip\_stats\_t **recv**
    - uip\_stats\_t **sent**
    - uip\_stats\_t **typeerr**
- } [icmp](#)

*ICMP statistics.*

- struct {
    - uip\_stats\_t **drop**
    - uip\_stats\_t **recv**
    - uip\_stats\_t **sent**
    - uip\_stats\_t **chkerr**
    - uip\_stats\_t **ackerr**
    - uip\_stats\_t **rst**
    - uip\_stats\_t **rexmit**
    - uip\_stats\_t **syndrop**
    - uip\_stats\_t **synrst**
- } [tcp](#)

*TCP statistics.*

## 6.11.2 Field Documentation

### 6.11.2.1 `uip_stats_t uip_stats::ackerr`

Number of TCP segments with a bad ACK number.

### 6.11.2.2 `uip_stats_t uip_stats::chkerr`

Number of TCP segments with a bad checksum.

### 6.11.2.3 `uip_stats_t uip_stats::drop`

Number of dropped TCP segments.

### 6.11.2.4 `uip_stats_t uip_stats::fragerr`

Number of packets dropped since they were IP fragments.

### 6.11.2.5 `uip_stats_t uip_stats::hblenerr`

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

### 6.11.2.6 `uip_stats_t uip_stats::lbenerr`

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

### 6.11.2.7 `uip_stats_t uip_stats::protoerr`

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

### 6.11.2.8 `uip_stats_t uip_stats::recv`

Number of received TCP segments.

### 6.11.2.9 `uip_stats_t uip_stats::rexmit`

Number of retransmitted TCP segments.

### 6.11.2.10 `uip_stats_t uip_stats::rst`

Number of received TCP RST (reset) segments.

### 6.11.2.11 `uip_stats_t uip_stats::sent`

Number of sent TCP segments.

**6.11.2.12 uip\_stats\_t uip\_stats::syndrop**

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

**6.11.2.13 uip\_stats\_t uip\_stats::synrst**

Number of SYNs for closed ports, triggering a RST.

**6.11.2.14 uip\_stats\_t uip\_stats::typeerr**

Number of ICMP packets with a wrong type.

**6.11.2.15 uip\_stats\_t uip\_stats::vhlerr**

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

## 6.12 uip\_udp\_conn Struct Reference

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

### 6.12.1 Detailed Description

Representation of a uIP UDP connection.

#### Data Fields

- `u16_t ripaddr` [2]  
*The IP address of the remote peer.*
- `u16_t lport`  
*The local port number in network byte order.*
- `u16_t rport`  
*The remote port number in network byte order.*
- `u8_t appstate` [UIP\_APPSTATE\_SIZE]  
*The application state.*



# Chapter 7

## Contiki 1.2-devel0 File Documentation

### 7.1 apps/program-handler.c File Reference

#### 7.1.1 Detailed Description

The program handler, used for loading programs and starting the screensaver.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto: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.

```
#include <string.h>
#include "ek.h"
#include "petsciiconv.h"
#include "ctk.h"
#include "ctk-draw.h"
#include "ctk-conf.h"
#include "log.h"
#include "loader.h"
#include "program-handler.h"
```

#### Functions

- void [program\\_handler\\_add](#) (struct [dsc](#) \*[dsc](#), char \*menuname, unsigned char desktop)  
*Add a program to the program handler.*
- void [program\\_handler\\_init](#) (void)  
*Initializes the program handler.*

- void `program_handler_load` (char \*name, char \*arg)  
*Loads a program and displays a dialog telling the user about it.*
- void `program_handler_screensaver` (char \*name)  
*Configures the name of the screensaver to be loaded when appropriate.*

## 7.1.2 Function Documentation

### 7.1.2.1 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.

Here is the call graph for this function:

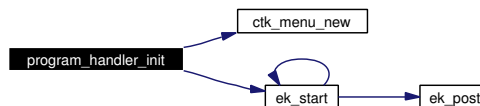


### 7.1.2.2 void `program_handler_init` (void)

Initializes the program handler.

Is called by the initialization before any programs have been added with `program_handler_add()`.

Here is the call graph for this function:



### 7.1.2.3 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.

Here is the call graph for this function:



**7.1.2.4 void program\_handler\_screensaver (char \* *name*)**

Configures the name of the screensaver to be loaded when appropriate.

**Parameters:**

*name* The name of the screensaver or NULL if no screensaver should be used.

## 7.2 conf/uiplib-conf.h.example File Reference

### 7.2.1 Detailed Description

uIP configuration file.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

This file contains configuration options for the uIP TCP/IP stack. Each Contiki port will contain its own uip-conf.h file containing architecture specific configuration options.

### Defines

- `#define UIP_CONF_MAX_CONNECTIONS 40`  
*The maximum number of TCP connections.*
- `#define UIP_CONF_MAX_LISTENPORTS 40`  
*The maximum number of listening TCP ports.*
- `#define UIP_CONF_BUFFER_SIZE 400`  
*The size of the uIP packet buffer.*
- `#define UIP_CONF_BYTE_ORDER LITTLE_ENDIAN`  
*The host byte order.*
- `#define UIP_CONF_PINGADDRCONF 0`  
*IP address configuration through ping.*

### 7.2.2 Define Documentation

#### 7.2.2.1 `#define UIP_CONF_BUFFER_SIZE 400`

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.

#### 7.2.2.2 `#define UIP_CONF_BYTE_ORDER LITTLE_ENDIAN`

The host byte order.

Used for telling uIP if the architecture has `LITTLE_ENDIAN` or `BIG_ENDIAN` byte order. x86 CPUs have `LITTLE_ENDIAN` byte order, whereas Motorola CPUs have `BIG_ENDIAN`. Check the documentation of the CPU to find out the byte order.

**7.2.2.3 #define UIP\_CONF\_MAX\_CONNECTIONS 40**

The maximum number of TCP connections.

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

**7.2.2.4 #define UIP\_CONF\_MAX\_LISTENPORTS 40**

The maximum number of listening TCP ports.

Each listening TCP port requires 2 bytes of memory.

**7.2.2.5 #define UIP\_CONF\_PINGADDRCONF 0**

IP address configuration through ping.

uIP features IP address configuration using an ICMP echo (ping) packet. In this mode, the destination IP address of the first ICMP echo packet that is received is used to set the host IP address.

## 7.3 conf/www-conf.h.example File Reference

### 7.3.1 Detailed Description

The Contiki web browser configuration file.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

#### Defines

- #define `WWW_CONF_WEBPAGE_WIDTH` 36  
*The width of the web page viewing area, measured in characters.*
- #define `WWW_CONF_WEBPAGE_HEIGHT` 17  
*The height of the web page viewing area, measured in characters.*
- #define `WWW_CONF_HISTORY_SIZE` 4  
*The size of the "Back" history.*
- #define `WWW_CONF_MAX_URLLEN` 100  
*The maximum length of the URLs the web browser will handle.*
- #define `WWW_CONF_MAX_NUMPAGEWIDGETS` 20  
*The maximum number of widgets (i.e., hyperlinks, form elements) on a single web page view.*
- #define `WWW_CONF_RENDERSTATE` 1  
*Turns support for the <center> tag on or off, and must be on for HTML forms to work.*
- #define `WWW_CONF_FORMS` 1  
*Toggles support for HTML forms.*
- #define `WWW_CONF_MAX_FORMACTIONLEN` 40  
*Maximum length of HTML form action URLs.*
- #define `WWW_CONF_MAX_FORMNAMELEN` 20  
*Maximum length of HTML form name.*
- #define `WWW_CONF_MAX_INPUTNAMELEN` 20  
*Maximum length of HTML form input name.*
- #define `WWW_CONF_MAX_INPUTVALUELEN` (WWW\_CONF\_WEBPAGE\_WIDTH - 1)  
*Maximum length of HTML form input value.*
- #define `WWW_CONF_HOMEPAGE` "http://contiki.c64.org/"  
*The default home page.*

## 7.3.2 Define Documentation

### 7.3.2.1 `#define WWW_CONF_MAX_NUMPAGEWIDGETS 20`

The maximum number of widgets (i.e., hyperlinks, form elements) on a single web page view.

**Note:**

This does not limit the total number of widgets in a web page, only the number of widget that are visible simultaneously.

## 7.4 ctk/ctk-draw.h File Reference

### 7.4.1 Detailed Description

CTK screen drawing module interface, ctk-draw.

**Author:**

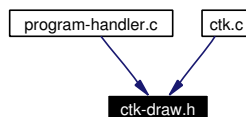
Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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.

```
#include "ctk.h"
```

```
#include "ctk-arch.h"
```

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



### 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)  
*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.*



## 7.5 ctk/ctk.c File Reference

### 7.5.1 Detailed Description

The Contiki Toolkit CTK, the Contiki GUI.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

```
#include "ek.h"
#include "cc.h"
#include "ctk.h"
#include "ctk-draw.h"
#include "ctk-conf.h"
#include "ctk-mouse.h"
#include "timer.h"
#include <string.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, ek\_id\_t id)  
*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

- ek\_event\_t [ctk\\_signal\\_keypress](#)  
*Emitted for every key being pressed.*
- ek\_event\_t [ctk\\_signal\\_widget\\_activate](#)  
*Emitted when a widget is activated (pressed).*
- ek\_event\_t [ctk\\_signal\\_button\\_activate](#)  
*Same as [ctk\\_signal\\_widget\\_activate](#).*
- ek\_event\_t [ctk\\_signal\\_widget\\_select](#)  
*Emitted when a widget is selected.*
- ek\_event\_t [ctk\\_signal\\_button\\_hover](#)  
*Same as [ctk\\_signal\\_widget\\_select](#).*
- ek\_event\_t [ctk\\_signal\\_hyperlink\\_activate](#)  
*Emitted when a hyperlink is activated.*
- ek\_event\_t [ctk\\_signal\\_hyperlink\\_hover](#)  
*Same as [ctk\\_signal\\_widget\\_select](#).*
- ek\_event\_t [ctk\\_signal\\_menu\\_activate](#)  
*Emitted when a menu item is activated.*

- `ek_event_t ctk_signal_window_close`  
*Emitted when a window is closed.*
- `ek_event_t ctk_signal_pointer_move`  
*Emitted when the mouse pointer is moved.*
- `ek_event_t ctk_signal_pointer_button`  
*Emitted when a mouse button is pressed.*

## 7.6 ctk/ctk.h File Reference

### 7.6.1 Detailed Description

CTK header file.

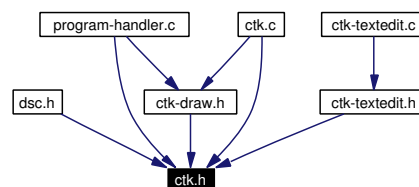
**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

The CTK header file contains function declarations and definitions of CTK structures and macros.

```
#include "ctk-conf.h"
#include "ctk-arch.h"
#include "ek.h"
#include "cc.h"
```

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



### 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\\_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) do {memset((e) → text, 0, (e) → len); (e) → xpos = 0;} while(0);  
*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, id) ctk\_icon\_add((struct [ctk\\_widget](#) \*)icon, id)  
*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.*

## Functions

- void [ctk\\_init](#) (void)  
*Initializes the Contiki Toolkit.*
- void [ctk\\_mode\\_set](#) (unsigned char mode)  
*Sets the current CTK mode.*
- unsigned char [ctk\\_mode\\_get](#) (void)  
*Retrieves the current CTK mode.*
- void [ctk\\_window\\_new](#) (struct [ctk\\_window](#) \*window, unsigned char w, unsigned char h, char \*title)  
*Create a new window.*
- void [ctk\\_window\\_clear](#) (struct [ctk\\_window](#) \*w)  
*Remove all widgets from a window.*
- void [ctk\\_window\\_close](#) (struct [ctk\\_window](#) \*w)  
*Close a window if it is open.*
- void [ctk\\_window\\_redraw](#) (struct [ctk\\_window](#) \*w)  
*Redraw a window.*
- 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\\_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\\_widget\\_redraw](#) (struct [ctk\\_widget](#) \*w)  
*Redraws a widget.*
- void [ctk\\_desktop\\_redraw](#) (struct [ctk\\_desktop](#) \*d)  
*Redraw the entire desktop.*
- 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

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



## 7.7 ek/arg.c File Reference

### 7.7.1 Detailed Description

Argument buffer for passing arguments when starting processes.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

```
#include "arg.h"
```

### Functions

- char \* [arg\\_alloc](#) (char size)  
*Allocates an argument buffer.*
- void [arg\\_free](#) (char \*arg)  
*Deallocates an argument buffer.*

## 7.8 ek/dsc.h File Reference

### 7.8.1 Detailed Description

Declaration of the DSC program description structure.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

```
#include "ctk.h"
```

### Data Structures

- struct [dsc](#)

*The DSC program description structure.*

### Defines

- #define [DSC](#)(dscname, description, prgname, initfunc, icon) const struct [dsc](#) dscname = { description, prgname, icon }

*Intantiating macro for the DSC structure.*

## 7.9 ek/ek.c File Reference

### 7.9.1 Detailed Description

Event kernel, event dispatcher and handler of uIP events.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

The dispatcher module is the event kernel in Contiki and handles processes, events and uIP events. All process execution is initiated by the dispatcher.

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

### Functions

- `ek_event_t ek_alloc_event` (void)  
*Allocates a event number.*
- `ek_id_t ek_start` (CC\_REGISTER\_ARG struct ek\_proc \*p)  
*Starts a new process.*
- `void ek_exit` (void)  
*Exit the currently running process.*
- `ek_proc * ek_process` (ek\_id\_t id)  
*Finds the process structure for a specific process ID.*
- `void ek_init` (void)  
*Initializes the dispatcher module.*
- `void ek_process_event` (void)  
*Process the next event in the event queue and deliver it to listening processes.*
- `void ek_process_poll` (void)  
*Call each process' poll handler.*
- `int ek_run` (void)  
*Run the system once - call poll handlers and process one event.*
- `ek_err_t ek_post` (ek\_id\_t id, ek\_event\_t s, ek\_data\_t data)  
*Post an asynchronous event.*
- `void ek_post_synch` (ek\_id\_t id, ek\_event\_t ev, ek\_data\_t data)  
*Post a synchronous event.*

## Variables

- ek\_event\_t [ek\\_event\\_quit](#)  
*The "quit" event.*
- ek\_event\_t [ek\\_event\\_msg](#)  
*A generic message event.*

## 7.10 ek/loader.h File Reference

### 7.10.1 Detailed Description

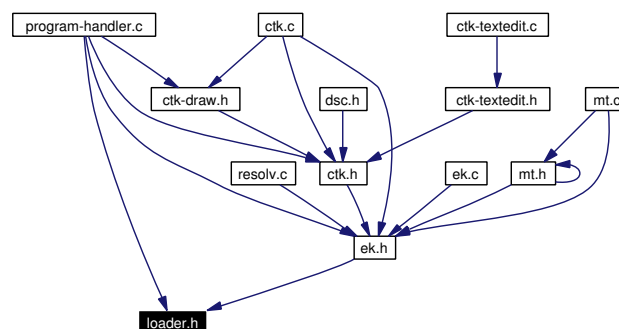
Default definitions and error values for the Contiki program loader.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

```
#include "loader-arch.h"
```

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



### Defines

- #define **LOADER\_OK** 0 /\*\*< No error. \*/  
*No error.*
- #define **LOADER\_ERR\_READ** 1 /\*\*< Read error. \*/  
*Read error.*
- #define **LOADER\_ERR\_HDR** 2 /\*\*< Header error. \*/  
*Header error.*
- #define **LOADER\_ERR\_OS** 3 /\*\*< Wrong OS. \*/  
*Wrong OS.*
- #define **LOADER\_ERR\_FMT** 4 /\*\*< Data format error. \*/  
*Data format error.*
- #define **LOADER\_ERR\_MEM** 5 /\*\*< Not enough memory. \*/  
*Not enough memory.*
- #define **LOADER\_ERR\_OPEN** 6 /\*\*< Could not open file. \*/  
*Could not open file.*
- #define **LOADER\_ERR\_ARCH** 7 /\*\*< Wrong architecture. \*/  
*Wrong architecture.*

- #define `LOADER_ERR_VERSION` 8 /\*\*< Wrong OS version. \*/  
*Wrong OS version.*
- #define `LOADER_ERR_NOLOADER` 9 /\*\*< Program loading not supported. \*/  
*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).*

## 7.11 ek/mt.c File Reference

### 7.11.1 Detailed Description

Implementation of the architecture agnostic parts of the preemptive multithreading library for Contiki.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

```
#include "ek.h"
#include "mt.h"
#include "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)  
*Start executing a thread.*
- void [mt\\_exit](#) (void)  
*Exit a thread.*
- void [mt\\_exec\\_event](#) (struct mt\_thread \*thread, ek\_event\_t ev, ek\_data\_t data)  
*Post an event to a thread.*
- void [mt\\_yield](#) (void)  
*Voluntarily give up the processor.*
- void [mt\\_post](#) (ek\_id\_t id, ek\_event\_t ev, ek\_data\_t data)  
*Emit a signal to another process.*
- void [mt\\_wait](#) (ek\_event\_t \*ev, ek\_data\_t \*data)  
*Block and wait for an event to occur.*
- void [mtp\\_start](#) (struct mtp\_thread \*t, void(\*function)(void \*), void \*data)  
*Start a thread.*

## 7.12 ek/mt.h File Reference

### 7.12.1 Detailed Description

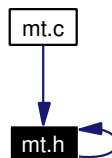
Header file for the preemptive multitasking library for Contiki.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

```
#include "ek.h"
#include "mtarch.h"
#include "mt.h"
```

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



### Defines

- #define [MT\\_OK](#)  
*No error.*
- #define [MTP](#)(thread, proc, name)  
*Declare a thread.*

### 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\\_yield](#) (void)  
*Yield the processor.*
- void [mtarch\\_exec](#) (struct mtarch\_thread \*thread)  
*Start executing 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)  
*Start executing a thread.*
- void [mt\\_exec\\_event](#) (struct mt\_thread \*thread, ek\_event\_t s, ek\_data\_t data)  
*Post an event to a thread.*
- void [mt\\_yield](#) (void)  
*Voluntarily give up the processor.*
- void [mt\\_post](#) (ek\_id\_t id, ek\_event\_t s, ek\_data\_t data)  
*Emit a signal to another process.*
- void [mt\\_wait](#) (ek\_event\_t \*s, ek\_data\_t \*data)  
*Block and wait for an event to occur.*
- void [mt\\_exit](#) (void)  
*Exit a thread.*
- void [mtp\\_start](#) (struct mtp\_thread \*t, void(\*function)(void \*), void \*data)  
*Start a thread.*

## 7.13 ek/pt-sem.h File Reference

### 7.13.1 Detailed Description

Couting semaphores implemented on protothreads.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

```
#include "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.*

## 7.14 ek/pt.h File Reference

### 7.14.1 Detailed Description

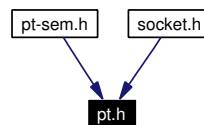
Protothreads implementation.

**Author:**

Adam Dunkels <adam@sics.se>

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

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



### Defines

- #define **PT\_THREAD**(name\_args)  
*Declaration of a protothread.*
- #define **PT\_INIT**(pt)  
*Initialize a protothread.*
- #define **PT\_BEGIN**(pt)  
*Start a protothread.*
- #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.*
- #define **PT\_WAIT\_THREAD**(pt, thread)  
*Block and wait until a child protothread completes.*
- #define **PT\_SPAWN**(pt, thread)  
*Spawn a child protothread and wait until it exits.*
- #define **PT\_RESTART**(pt)  
*Restart the protothread.*
- #define **PT\_EXIT**(pt)  
*Exit the protothread.*
- #define **PT\_END**(pt)  
*Declare the end of a protothread.*

## 7.15 lib/cc.h File Reference

### 7.15.1 Detailed Description

Default definitions of C compiler quirk work-arounds.

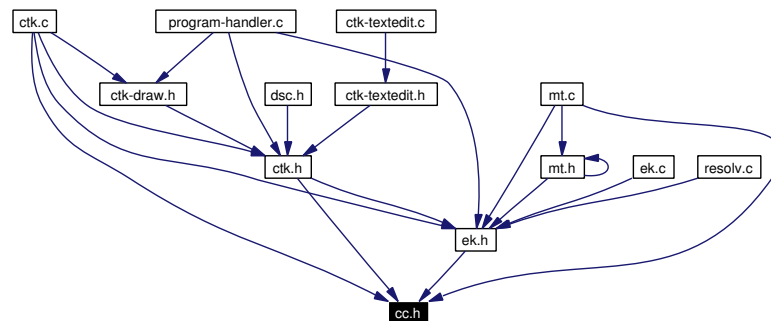
**Author:**

Adam Dunkels <[adam@dunkels.com](mailto: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.

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

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



### 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.

## 7.16 lib/ctk-textedit.c File Reference

### 7.16.1 Detailed Description

An experimental CTK text edit widget.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto: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.

```
#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, ek\_event\_t s, ek\_data\_t data)  
*The CTK textedit signal handler.*

### 7.16.2 Function Documentation

#### 7.16.2.1 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.

#### 7.16.2.2 void ctk\_textedit\_eventhandler (struct ctk\_textedit \* t, ek\_event\_t s, ek\_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.

## 7.17 lib/ctk-textedit.h File Reference

### 7.17.1 Detailed Description

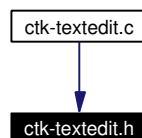
Header file for the experimental application level CTK textedit widget.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

```
#include "ctk.h"
```

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



### 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, [ek\\_event\\_t](#) s, [ek\\_data\\_t](#) data)  
*The CTK textedit signal handler.*

### 7.17.2 Define Documentation

#### 7.17.2.1 #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.

### 7.17.3 Function Documentation

#### 7.17.3.1 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.

#### 7.17.3.2 void ctk\_textedit\_eventhandler (struct ctk\_textedit \* *t*, ek\_event\_t *s*, ek\_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.

## 7.18 lib/memb.c File Reference

### 7.18.1 Detailed Description

Memory block allocation routines.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

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

```
#include "memb.h"
```



## 7.19 lib/memb.h File Reference

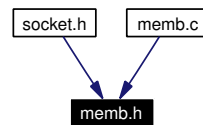
### 7.19.1 Detailed Description

Memory block allocation routines.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

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



### Defines

- #define [MEMB](#)(name, size, num)  
*Declare a memory block.*

### Functions

- void [memb\\_init](#) (struct memb\_blocks \*m)  
*Initialize a memory block that was declared with [MEMB](#)().*
- char \* [memb\\_alloc](#) (struct memb\_blocks \*m)  
*Allocate a memory block from a block of memory declared with [MEMB](#)().*
- char [memb\\_ref](#) (struct memb\_blocks \*m, char \*ptr)  
*Increase the reference count for a memory chunk.*
- char [memb\\_free](#) (struct memb\_blocks \*m, void \*ptr)  
*Deallocate a memory block from a memory block previously declared with [MEMB](#)().*

## 7.20 lib/petsciiconv.h File Reference

### 7.20.1 Detailed Description

PETSCII/ASCII conversion functions.

**Author:**

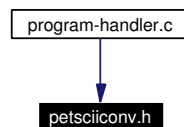
Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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.

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



### Functions

- void [petsciiconv\\_toascii](#) (char \*buf, unsigned int len)  
*Convert a text buffer from PETSCII to ASCII.*
- void [petsciiconv\\_topetscii](#) (char \*buf, unsigned int len)  
*Convert a text buffer from ASCII to PETSCII.*

### 7.20.2 Function Documentation

#### 7.20.2.1 void petsciiconv\_toascii (char \* buf, unsigned int len)

Convert a text buffer from PETSCII to ASCII.

**Parameters:**

*buf* A pointer to the buffer which is to be converted.

*len* The length of the buffer to be converted.

#### 7.20.2.2 void petsciiconv\_topetscii (char \* buf, unsigned int len)

Convert a text buffer from ASCII to PETSCII.

**Parameters:**

*buf* A pointer to the buffer which is to be converted.

*len* The length of the buffer to be converted.

## 7.21 uip/resolv.c File Reference

### 7.21.1 Detailed Description

DNS host name to IP address resolver.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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

```
#include "ek.h"
#include "tcpip.h"
#include "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.*
- u16\_t \* [resolv\\_getserver](#) (void)  
*Obtain the currently configured DNS server.*
- void [resolv\\_conf](#) (u16\_t \*dnsserver)  
*Configure a DNS server.*
- void [resolv\\_init](#) (char \*arg)  
*Initialize the resolver.*

### Variables

- ek\_event\_t [resolv\\_event\\_found](#)  
*Signal that is sent when a DNS name has been resolved.*

## 7.22 uip/resolv.h File Reference

### 7.22.1 Detailed Description

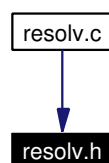
uIP DNS resolver code header file.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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

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



### Functions

- void [resolv\\_conf](#) (u16\_t \*dnsserver)  
*Configure a DNS server.*
- u16\_t \* [resolv\\_getserver](#) (void)  
*Obtain the currently configured DNS server.*
- void [resolv\\_init](#) (char \*arg)  
*Initialize the resolver.*
- u16\_t \* [resolv\\_lookup](#) (char \*name)  
*Look up a hostname in the array of known hostnames.*
- void [resolv\\_query](#) (char \*name)  
*Queues a name so that a question for the name will be sent out.*

### Variables

- ek\_event\_t [resolv\\_event\\_found](#)  
*Signal that is sent when a DNS name has been resolved.*

## 7.23 uip/socket.h File Reference

### 7.23.1 Detailed Description

Socket library header file.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

```
#include "pt.h"
#include "uipbuf.h"
#include "memb.h"
```

### Data Structures

- struct [socket](#)  
*The representation of a socket.*

### Defines

- #define [SOCKET\\_INIT](#)([socket](#), buffer, buffersize)  
*Initialize a socket.*
- #define [SOCKET\\_BEGIN](#)([socket](#))  
*Start the socket protothread in a function.*
- #define [SOCKET\\_SEND](#)([socket](#), data, datalen)  
*Send data.*
- #define [SOCKET\\_CLOSE](#)([socket](#))  
*Close a socket.*
- #define [SOCKET\\_READTO](#)([socket](#), c)  
*Read data up to a specified character.*
- #define [SOCKET\\_DATALEN](#)([socket](#))  
*The length of the data that was previously read.*
- #define [SOCKET\\_EXIT](#)([socket](#))  
*Exit the socket's protothread.*
- #define [SOCKET\\_CLOSE\\_EXIT](#)([socket](#))  
*Close a socket and exit the socket's protothread.*
- #define [SOCKET\\_NEWDATA](#)([socket](#))  
*Check if new data has arrived on a socket.*

- #define [SOCKET\\_WAIT\\_UNTIL](#)([socket](#), condition)

*Wait until data arrives or until a condition is true.*

## 7.24 uip/uip-split.h File Reference

### 7.24.1 Detailed Description

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

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

**Functions**

- void [uip\\_split\\_output](#) (void)  
*Handle outgoing packets.*

## 7.25 uip/uip.c File Reference

### 7.25.1 Detailed Description

The uIP TCP/IP stack code.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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

### Functions

- void [uip\\_init](#) (void)  
*uIP initialization function.*
- [uip\\_conn](#) \* [uip\\_connect](#) (u16\_t \*ripaddr, u16\_t rport)  
*Connect to a remote host using TCP.*
- [uip\\_udp\\_conn](#) \* [uip\\_udp\\_new](#) (u16\_t \*ripaddr, u16\_t rport)  
*Set up a new UDP connection.*
- void [uip\\_unlisten](#) (u16\_t port)  
*Stop listening to the specified port.*
- void [uip\\_listen](#) (u16\_t port)  
*Start listening to the specified port.*
- u16\_t [htons](#) (u16\_t val)  
*Convert 16-bit quantity from host byte order to network byte order.*

### Variables

- u8\_t [uip\\_buf](#) [UIP\_BUFSIZE+2]  
*The uIP packet buffer.*
- u8\_t \* [uip\\_appdata](#)  
*Pointer to the application data in the packet buffer.*
- u8\_t [uip\\_acc32](#) [4]  
*4-byte array used for the 32-bit sequence number calculations.*



## 7.26 uip/uip.h File Reference

### 7.26.1 Detailed Description

Header file for the uIP TCP/IP stack.

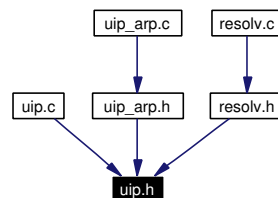
**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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

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

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



### Data Structures

- struct [uip\\_conn](#)  
*Representation of a uIP TCP connection.*
- struct [uip\\_udp\\_conn](#)  
*Representation of a uIP UDP connection.*
- struct [uip\\_stats](#)  
*The structure holding the TCP/IP statistics that are gathered if UIP\_STATISTICS is set to 1.*

### 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)`  
*Periodic processing for a connection identified by a pointer to its structure.*
- `#define uip_udp_periodic(conn)`  
*Periodic processing for a UDP connection identified by its number.*
- `#define uip_udp_periodic_conn(conn)`  
*Periodic processing for a UDP connection identified by a pointer to its structure.*
- `#define uip_send(data, len)`  
*Send data on the current connection.*
- `#define uip_datalen()`  
*The length of any incoming data that is currently available (if available) in the uip\_appdata buffer.*
- `#define uip_urgdatalen()`  
*The length of any out-of-band data (urgent data) that has arrived on the connection.*
- `#define uip_close()`  
*Close the current connection.*
- `#define uip_abort()`  
*Abort the current connection.*
- `#define uip_stop()`  
*Tell the sending host to stop sending data.*
- `#define uip_stopped(conn)`  
*Find out if the current connection has been previously stopped with `uip_stop()`.*
- `#define uip_restart()`  
*Restart the current connection, if it has previously been stopped with `uip_stop()`.*
- `#define uip_udpconnection()`  
*Is the current connection a UDP connection?*
- `#define uip_newdata()`  
*Is new incoming data available?*

- #define `uip_acked()`  
*Has previously sent data been acknowledged?*
- #define `uip_connected()`  
*Has the connection just been connected?*
- #define `uip_closed()`  
*Has the connection been closed by the other end?*
- #define `uip_aborted()`  
*Has the connection been aborted by the other end?*
- #define `uip_timedout()`  
*Has the connection timed out?*
- #define `uip_rexmit()`  
*Do we need to retransmit previously data?*
- #define `uip_poll()`  
*Is the connection being polled by uIP?*
- #define `uip_initialmss()`  
*Get the initial maximum segment size (MSS) of the current connection.*
- #define `uip_mss()`  
*Get the current maximum segment size that can be sent on the current connection.*
- #define `uip_udp_remove(conn)`  
*Removed a UDP connection.*
- #define `uip_udp_bind(conn, port)`  
*Bind a UDP connection to a local port.*
- #define `uip_udp_send(len)`  
*Send a UDP datagram of length len on the current connection.*
- #define `uip_ipaddr(addr, addr0, addr1, addr2, addr3)`  
*Pack an IP address into a 4-byte array which is used by uIP to represent IP addresses.*
- #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

- `void uip_init (void)`  
*uIP initialization function.*
- `void uip_listen (u16_t port)`  
*Start listening to the specified port.*
- `void uip_unlisten (u16_t port)`  
*Stop listening to the specified port.*
- `uip_conn * uip_connect (u16_t *ripaddr, u16_t port)`  
*Connect to a remote host using TCP.*
- `uip_udp_conn * uip_udp_new (u16_t *ripaddr, u16_t rport)`  
*Set up a new UDP connection.*
- `u16_t htons (u16_t val)`  
*Convert 16-bit quantity from host byte order to network byte order.*

## Variables

- `u8_t uip_buf [UIP_BUFSIZE+2]`  
*The uIP packet buffer.*
- `u8_t * uip_appdata`  
*Pointer to the application data in the packet buffer.*
- `u8_t uip_acc32 [4]`  
*4-byte array used for the 32-bit sequence number calculations.*

- [uip\\_stats uip\\_stat](#)

*The uIP TCP/IP statistics.*

## 7.27 uip/uip\_arp.c File Reference

### 7.27.1 Detailed Description

Implementation of the ARP Address Resolution Protocol.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

```
#include "uip_arp.h"
```

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

## 7.28 uip/uip\_arp.h File Reference

### 7.28.1 Detailed Description

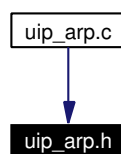
Macros and definitions for the ARP module.

**Author:**

Adam Dunkels <[adam@dunkels.com](mailto:adam@dunkels.com)>

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

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



### Data Structures

- struct [uip\\_eth\\_addr](#)  
*Representation of a 48-bit Ethernet address.*
- struct [uip\\_eth\\_hdr](#)  
*The Ethernet header.*

### Defines

- #define [uip\\_setethaddr](#)(eaddr)  
*Specify the Ethernet MAC address.*

### Functions

- void [uip\\_arp\\_init](#) (void)  
*Initialize the ARP module.*
- void [uip\\_arp\\_arpin](#) (void)  
*ARP processing for incoming ARP packets.*
- void [uip\\_arp\\_out](#) (void)  
*Prepend Ethernet header to an outbound IP packet and see if we need to send out an ARP request.*
- void [uip\\_arp\\_timer](#) (void)  
*Periodic ARP processing function.*

## 7.29 uip/uiplib.h File Reference

### 7.29.1 Detailed Description

Various uIP library functions.

**Author:**

Adam Dunkels <[adam@sics.se](mailto:adam@sics.se)>

### Functions

- unsigned char [uiplib\\_ipaddrconv](#) (char \*addrstr, unsigned char \*addr)  
*Convert a textual representation of an IP address to a numerical representation.*



# Index

- ackerr
  - uip\_stats, [120](#)
- active
  - ctk\_window, [112](#)
- apps/program-handler.c, [123](#)
- Architecture support for multi-threading, [98](#)
- arg\_alloc
  - kernel, [13](#)
- arg\_free
  - kernel, [13](#)
- chkerr
  - uip\_stats, [120](#)
- conf/uip-conf.h.example, [126](#)
- conf/www-conf.h.example, [128](#)
- ctk
  - ctk\_dialog\_open, [43](#)
  - ctk\_init, [43](#)
  - ctk\_menu\_add, [43](#)
  - ctk\_menu\_remove, [43](#)
  - ctk\_mode\_get, [44](#)
  - ctk\_mode\_set, [44](#)
  - ctk\_window\_clear, [44](#)
  - ctk\_window\_close, [44](#)
  - ctk\_window\_new, [44](#)
  - ctk\_window\_redraw, [45](#)
- CTK application functions, [28](#)
- CTK device driver functions, [46](#)
- ctk-textedit.c
  - ctk\_textedit\_add, [151](#)
  - ctk\_textedit\_eventhandler, [151](#)
- ctk-textedit.h
  - CTK\_TEXTEDIT, [152](#)
  - ctk\_textedit\_add, [153](#)
  - ctk\_textedit\_eventhandler, [153](#)
- ctk/ctk-draw.h, [130](#)
- ctk/ctk.c, [131](#)
- ctk/ctk.h, [134](#)
- CTK\_BUTTON
  - ctkappfunc, [31](#)
- ctk\_button\_set\_text
  - ctkappfunc, [31](#)
- ctk\_desktop\_height
  - ctkappfunc, [36](#)
- ctk\_desktop\_redraw
  - ctkappfunc, [36](#)
- ctk\_desktop\_width
  - ctkappfunc, [36](#)
- ctk\_dialog\_new
  - ctkappfunc, [37](#)
- ctk\_dialog\_open
  - ctk, [43](#)
  - ctkappfunc, [37](#)
- ctk\_draw\_clear
  - ctkdraw, [48](#)
- ctk\_draw\_clear\_window
  - ctkdraw, [48](#)
- ctk\_draw\_dialog
  - ctkdraw, [49](#)
- ctk\_draw\_init
  - ctkdraw, [49](#)
- ctk\_draw\_widget
  - ctkdraw, [49](#)
- ctk\_draw\_window
  - ctkdraw, [50](#)
- CTK\_HYPERLINK
  - ctkappfunc, [31](#)
- CTK\_ICON
  - ctkappfunc, [32](#)
- CTK\_ICON\_ADD
  - ctkappfunc, [32](#)
- ctk\_icon\_add
  - ctkappfunc, [37](#)
- ctk\_init
  - ctk, [43](#)
- CTK\_LABEL
  - ctkappfunc, [32](#)
- ctk\_label\_set\_height
  - ctkappfunc, [33](#)
- ctk\_label\_set\_text
  - ctkappfunc, [33](#)
- ctk\_menu, [105](#)
  - titlelen, [106](#)
- ctk\_menu\_add
  - ctk, [43](#)
  - ctkappfunc, [37](#)
- ctk\_menu\_new
  - ctkappfunc, [38](#)
- ctk\_menu\_remove
  - ctk, [43](#)

- ctkappfunc, 38
- ctk\_menuitem, 107
- ctk\_menuitem\_add
  - ctkappfunc, 38
- ctk\_menus, 108
  - open, 108
- ctk\_mode\_get
  - ctk, 44
  - ctkappfunc, 38
- ctk\_mode\_set
  - ctk, 44
  - ctkappfunc, 38
- CTK\_SEPARATOR
  - ctkappfunc, 33
- ctk\_signal\_hyperlink\_activate
  - ctkappfunc, 41
  - signals, 102
- ctk\_signal\_keypress
  - ctkappfunc, 41
  - signals, 102
- ctk\_signal\_menu\_activate
  - ctkappfunc, 41
  - signals, 103
- ctk\_signal\_pointer\_button
  - ctkappfunc, 41
  - signals, 103
- ctk\_signal\_pointer\_move
  - ctkappfunc, 41
  - signals, 103
- ctk\_signal\_widget\_activate
  - ctkappfunc, 41
  - signals, 103
- ctk\_signal\_widget\_select
  - ctkappfunc, 41
  - signals, 103
- ctk\_signal\_window\_close
  - ctkappfunc, 41
  - signals, 103
- CTK\_TEXTEDIT
  - ctk-textedit.h, 152
- ctk\_textedit\_add
  - ctk-textedit.c, 151
  - ctk-textedit.h, 153
- ctk\_textedit\_eventhandler
  - ctk-textedit.c, 151
  - ctk-textedit.h, 153
- CTK\_TEXTENTRY
  - ctkappfunc, 33
- CTK\_TEXTENTRY\_CLEAR
  - ctkappfunc, 34
- ctk\_widget, 109
- CTK\_WIDGET\_ADD
  - ctkappfunc, 34
- ctk\_widget\_add
  - ctkappfunc, 39
- CTK\_WIDGET\_FOCUS
  - ctkappfunc, 34
- CTK\_WIDGET\_REDRAW
  - ctkappfunc, 34
- ctk\_widget\_redraw
  - ctkappfunc, 39
- CTK\_WIDGET\_SET\_WIDTH
  - ctkappfunc, 35
- CTK\_WIDGET\_SET\_XPOS
  - ctkappfunc, 35
- CTK\_WIDGET\_SET\_YPOS
  - ctkappfunc, 35
- CTK\_WIDGET\_TYPE
  - ctkappfunc, 35
- CTK\_WIDGET\_XPOS
  - ctkappfunc, 35
- CTK\_WIDGET\_YPOS
  - ctkappfunc, 36
- ctk\_window, 111
  - active, 112
  - inactive, 112
  - owner, 112
  - title, 112
- ctk\_window\_clear
  - ctk, 44
  - ctkappfunc, 39
- ctk\_window\_close
  - ctk, 44
  - ctkappfunc, 39
- ctk\_window\_new
  - ctk, 44
  - ctkappfunc, 40
- ctk\_window\_open
  - ctkappfunc, 40
- ctk\_window\_redraw
  - ctk, 45
  - ctkappfunc, 40
- ctkappfunc
  - CTK\_BUTTON, 31
  - ctk\_button\_set\_text, 31
  - ctk\_desktop\_height, 36
  - ctk\_desktop\_redraw, 36
  - ctk\_desktop\_width, 36
  - ctk\_dialog\_new, 37
  - ctk\_dialog\_open, 37
  - CTK\_HYPERLINK, 31
  - CTK\_ICON, 32
  - CTK\_ICON\_ADD, 32
  - ctk\_icon\_add, 37
  - CTK\_LABEL, 32
  - ctk\_label\_set\_height, 33
  - ctk\_label\_set\_text, 33
  - ctk\_menu\_add, 37

- ctk\_menu\_new, 38
- ctk\_menu\_remove, 38
- ctk\_menuitem\_add, 38
- ctk\_mode\_get, 38
- ctk\_mode\_set, 38
- CTK\_SEPARATOR, 33
- ctk\_signal\_hyperlink\_activate, 41
- ctk\_signal\_keypress, 41
- ctk\_signal\_menu\_activate, 41
- ctk\_signal\_pointer\_button, 41
- ctk\_signal\_pointer\_move, 41
- ctk\_signal\_widget\_activate, 41
- ctk\_signal\_widget\_select, 41
- ctk\_signal\_window\_close, 41
- CTK\_TEXTENTRY, 33
- CTK\_TEXTENTRY\_CLEAR, 34
- CTK\_WIDGET\_ADD, 34
- ctk\_widget\_add, 39
- CTK\_WIDGET\_FOCUS, 34
- CTK\_WIDGET\_REDRAW, 34
- ctk\_widget\_redraw, 39
- CTK\_WIDGET\_SET\_WIDTH, 35
- CTK\_WIDGET\_SET\_XPOS, 35
- CTK\_WIDGET\_SET\_YPOS, 35
- CTK\_WIDGET\_TYPE, 35
- CTK\_WIDGET\_XPOS, 35
- CTK\_WIDGET\_YPOS, 36
- ctk\_window\_clear, 39
- ctk\_window\_close, 39
- ctk\_window\_new, 40
- ctk\_window\_open, 40
- ctk\_window\_redraw, 40
- ctkdraw
  - ctk\_draw\_clear, 48
  - ctk\_draw\_clear\_window, 48
  - ctk\_draw\_dialog, 49
  - ctk\_draw\_init, 49
  - ctk\_draw\_widget, 49
  - ctk\_draw\_window, 50
- drop
  - uip\_stats, 120
- DSC
  - loader, 17
- dsc, 113
  - loadaddr, 113
- ek/arg.c, 139
- ek/dsc.h, 140
- ek/ek.c, 141
- ek/loader.h, 143
- ek/mt.c, 145
- ek/mt.h, 146
- ek/pt-sem.h, 148
- ek/pt.h, 149
- ek\_alloc\_event
  - kernel, 13
- ek\_event\_msg
  - events, 9
- ek\_event\_quit
  - events, 9
- ek\_exit
  - kernel, 13
- ek\_init
  - kernel, 13
- ek\_post
  - kernel, 14
- ek\_post\_synch
  - kernel, 14
- ek\_process
  - kernel, 14
- ek\_run
  - kernel, 14
- ek\_start
  - kernel, 15
- events
  - ek\_event\_msg, 9
  - ek\_event\_quit, 9
- fragerr
  - uip\_stats, 120
- hblenerr
  - uip\_stats, 120
- HTONS
  - uipconvfunc, 76
- htons
  - uip, 57
  - uipconvfunc, 79
- inactive
  - ctk\_window, 112
- kernel
  - arg\_alloc, 13
  - arg\_free, 13
  - ek\_alloc\_event, 13
  - ek\_exit, 13
  - ek\_init, 13
  - ek\_post, 14
  - ek\_post\_synch, 14
  - ek\_process, 14
  - ek\_run, 14
  - ek\_start, 15
- lblenerr
  - uip\_stats, 120
- lib/cc.h, 150
- lib/ctk-textedit.c, 151

- lib/ctk-textedit.h, 152
- lib/memb.c, 154
- lib/memb.h, 155
- lib/petsciiconv.h, 156
- loadaddr
  - dsc, 113
- loader
  - DSC, 17
  - LOADER\_LOAD, 17
  - LOADER\_LOAD\_DSC, 17
  - LOADER\_UNLOAD, 18
  - LOADER\_UNLOAD\_DSC, 18
- LOADER\_LOAD
  - loader, 17
- LOADER\_LOAD\_DSC
  - loader, 17
- LOADER\_UNLOAD
  - loader, 18
- LOADER\_UNLOAD\_DSC
  - loader, 18
- Local continuations, 24
- MEMB
  - memb, 92
- memb
  - MEMB, 92
  - memb\_alloc, 92
  - memb\_free, 92
  - memb\_init, 93
  - memb\_ref, 93
- memb\_alloc
  - memb, 92
- memb\_free
  - memb, 92
- memb\_init
  - memb, 93
- memb\_ref
  - memb, 93
- Memory block management functions, 91
- mt
  - mt\_exec, 95
  - mt\_exec\_event, 95
  - mt\_exit, 95
  - mt\_post, 95
  - mt\_start, 96
  - mt\_wait, 96
  - mt\_yield, 96
- mt\_exec
  - mt, 95
- mt\_exec\_event
  - mt, 95
- mt\_exit
  - mt, 95
- mt\_post
  - mt, 95
- mt\_start
  - mt, 96
- mt\_wait
  - mt, 96
- mt\_yield
  - mt, 96
- mtarch
  - mtarch\_exec, 98
  - mtarch\_init, 98
  - mtarch\_start, 99
  - mtarch\_yield, 99
- mtarch\_exec
  - mtarch, 98
- mtarch\_init
  - mtarch, 98
- mtarch\_start
  - mtarch, 99
- mtarch\_yield
  - mtarch, 99
- MTP
  - mtp, 100
- mtp
  - MTP, 100
  - mtp\_start, 101
- mtp\_start
  - mtp, 101
- Multi-threading library convenience functions, 100
- open
  - ctk\_menus, 108
- owner
  - ctk\_window, 112
- Peemptive multi-threading, 94
- petsciiconv.h
  - petsciiconv\_toascii, 156
  - petsciiconv\_topetscii, 156
- petsciiconv\_toascii
  - petsciiconv.h, 156
- petsciiconv\_topetscii
  - petsciiconv.h, 156
- program-handler.c
  - program\_handler\_add, 124
  - program\_handler\_init, 124
  - program\_handler\_load, 124
  - program\_handler\_screensaver, 124
- program\_handler\_add
  - program-handler.c, 124
- program\_handler\_init
  - program-handler.c, 124
- program\_handler\_load
  - program-handler.c, 124

- program\_handler\_screensaver
  - program-handler.c, [124](#)
- protoerr
  - uip\_stats, [120](#)
- Protothread semaphores, [25](#)
- Protothreads, [19](#)
- pt
  - PT\_BEGIN, [20](#)
  - PT\_END, [20](#)
  - PT\_EXIT, [21](#)
  - PT\_INIT, [21](#)
  - PT\_RESTART, [21](#)
  - PT\_SPAWN, [21](#)
  - PT\_THREAD, [21](#)
  - PT\_WAIT\_THREAD, [22](#)
  - PT\_WAIT\_UNTIL, [22](#)
  - PT\_WAIT\_WHILE, [23](#)
- PT\_BEGIN
  - pt, [20](#)
- PT\_END
  - pt, [20](#)
- PT\_EXIT
  - pt, [21](#)
- PT\_INIT
  - pt, [21](#)
- PT\_RESTART
  - pt, [21](#)
- PT\_SEM\_INIT
  - ptsem, [26](#)
- PT\_SEM\_SIGNAL
  - ptsem, [27](#)
- PT\_SEM\_WAIT
  - ptsem, [27](#)
- PT\_SPAWN
  - pt, [21](#)
- PT\_THREAD
  - pt, [21](#)
- PT\_WAIT\_THREAD
  - pt, [22](#)
- PT\_WAIT\_UNTIL
  - pt, [22](#)
- PT\_WAIT\_WHILE
  - pt, [23](#)
- ptsem
  - PT\_SEM\_INIT, [26](#)
  - PT\_SEM\_SIGNAL, [27](#)
  - PT\_SEM\_WAIT, [27](#)
- recv
  - uip\_stats, [120](#)
- resolv\_conf
  - uipdns, [83](#)
- resolv\_getserver
  - uipdns, [84](#)
- resolv\_lookup
  - uipdns, [84](#)
- resolv\_query
  - uipdns, [84](#)
- rexmit
  - uip\_stats, [120](#)
- rst
  - uip\_stats, [120](#)
- sent
  - uip\_stats, [120](#)
- signals
  - gtk\_signal\_hyperlink\_activate, [102](#)
  - gtk\_signal\_keypress, [102](#)
  - gtk\_signal\_menu\_activate, [103](#)
  - gtk\_signal\_pointer\_button, [103](#)
  - gtk\_signal\_pointer\_move, [103](#)
  - gtk\_signal\_widget\_activate, [103](#)
  - gtk\_signal\_widget\_select, [103](#)
  - gtk\_signal\_window\_close, [103](#)
- socket, [114](#)
  - SOCKET\_BEGIN, [88](#)
  - SOCKET\_CLOSE, [88](#)
  - SOCKET\_CLOSE\_EXIT, [88](#)
  - SOCKET\_DATALEN, [88](#)
  - SOCKET\_EXIT, [88](#)
  - SOCKET\_INIT, [88](#)
  - SOCKET\_NEWDATA, [89](#)
  - SOCKET\_READTO, [89](#)
  - SOCKET\_SEND, [89](#)
  - SOCKET\_WAIT\_UNTIL, [89](#)
- Socket library, [85](#)
- SOCKET\_BEGIN
  - socket, [88](#)
- SOCKET\_CLOSE
  - socket, [88](#)
- SOCKET\_CLOSE\_EXIT
  - socket, [88](#)
- SOCKET\_DATALEN
  - socket, [88](#)
- SOCKET\_EXIT
  - socket, [88](#)
- SOCKET\_INIT
  - socket, [88](#)
- SOCKET\_NEWDATA
  - socket, [89](#)
- SOCKET\_READTO
  - socket, [89](#)
- SOCKET\_SEND
  - socket, [89](#)
- SOCKET\_WAIT\_UNTIL
  - socket, [89](#)
- syndrop
  - uip\_stats, [120](#)

- synrst
  - uip\_stats, 121
- System events, 9
- System signals, 102
- The Contiki event kernel, 11
- The Contiki program loader, 16
- The CTK graphical user interface., 42
- The uIP TCP/IP stack, 51
- title
  - ctk\_window, 112
- titlelen
  - ctk\_menu, 106
- typeerr
  - uip\_stats, 121
- uip
  - htons, 57
  - uip\_appdata, 59
  - uip\_buf, 59
  - uip\_connect, 57
  - uip\_init, 58
  - uip\_listen, 58
  - uip\_stat, 59
  - uip\_udp\_new, 58
  - uip\_unlisten, 59
- uIP Address Resolution Protocol, 80
- uIP application functions, 68
- uIP configuration functions, 61
- uIP conversion functions, 75
- uIP device driver functions, 64
- uIP hostname resolver functions, 83
- uIP initialization functions, 63
- uIP TCP throughput booster hack, 82
- uip-conf.h.example
  - UIP\_CONF\_BUFFER\_SIZE, 126
  - UIP\_CONF\_BYTE\_ORDER, 126
  - UIP\_CONF\_MAX\_CONNECTIONS, 126
  - UIP\_CONF\_MAX\_LISTENPORTS, 127
  - UIP\_CONF\_PINGADDRCONF, 127
- uip/resolv.c, 157
- uip/resolv.h, 158
- uip/socket.h, 159
- uip/uip-split.h, 161
- uip/uip.c, 162
- uip/uip.h, 163
- uip/uip\_arp.c, 168
- uip/uip\_arp.h, 169
- uip/uilib.h, 170
- uip\_abort
  - uipappfunc, 69
- uip\_aborted
  - uipappfunc, 69
- uip\_acked
  - uipappfunc, 70
- uip\_appdata
  - uip, 59
- uip\_arp\_arpin
  - uiparp, 81
- uip\_arp\_out
  - uiparp, 81
- uip\_arp\_timer
  - uiparp, 81
- uip\_buf
  - uip, 59
  - uipdevfunc, 66
- uip\_close
  - uipappfunc, 70
- uip\_closed
  - uipappfunc, 70
- UIP\_CONF\_BUFFER\_SIZE
  - uip-conf.h.example, 126
- UIP\_CONF\_BYTE\_ORDER
  - uip-conf.h.example, 126
- UIP\_CONF\_MAX\_CONNECTIONS
  - uip-conf.h.example, 126
- UIP\_CONF\_MAX\_LISTENPORTS
  - uip-conf.h.example, 127
- UIP\_CONF\_PINGADDRCONF
  - uip-conf.h.example, 127
- uip\_conn, 115
- uip\_connect
  - uip, 57
  - uipappfunc, 73
- uip\_connected
  - uipappfunc, 70
- uip\_dataalen
  - uipappfunc, 70
- uip\_eth\_addr, 117
- uip\_eth\_hdr, 118
- uip\_getdraddr
  - uipconffunc, 61
- uip\_ghostaddr
  - uipconffunc, 61
- uip\_getnetmask
  - uipconffunc, 61
- uip\_init
  - uip, 58
  - uipinit, 63
- uip\_input
  - uipdevfunc, 64
- uip\_ipaddr
  - uipconvfunc, 76
- uip\_ipaddr1
  - uipconvfunc, 76
- uip\_ipaddr2
  - uipconvfunc, 76
- uip\_ipaddr3

- uipconvfunc, 76
- uip\_ipaddr4
  - uipconvfunc, 77
- uip\_ipaddr\_cmp
  - uipconvfunc, 77
- uip\_ipaddr\_copy
  - uipconvfunc, 77
- uip\_ipaddr\_mask
  - uipconvfunc, 78
- uip\_ipaddr\_maskcmp
  - uipconvfunc, 78
- uip\_listen
  - uip, 58
  - uipappfunc, 73
- uip\_mss
  - uipappfunc, 70
- uip\_newdata
  - uipappfunc, 70
- uip\_periodic
  - uipdevfunc, 65
- uip\_periodic\_conn
  - uipdevfunc, 65
- uip\_poll
  - uipappfunc, 70
- uip\_restart
  - uipappfunc, 71
- uip\_rexmit
  - uipappfunc, 71
- uip\_send
  - uipappfunc, 71
- uip\_setdraddr
  - uipconffunc, 62
- uip\_setethaddr
  - uipconffunc, 62
- uip\_sethostaddr
  - uipconffunc, 62
- uip\_setnetmask
  - uipconffunc, 62
- uip\_split\_output
  - uipsplit, 82
- uip\_stat
  - uip, 59
- uip\_stats, 119
  - ackerr, 120
  - chkerr, 120
  - drop, 120
  - fragerr, 120
  - hblenerr, 120
  - lblenerr, 120
  - protoerr, 120
  - recv, 120
  - rexmit, 120
  - rst, 120
  - sent, 120
  - syndrop, 120
  - synrst, 121
  - typeerr, 121
  - vherr, 121
- uip\_stop
  - uipappfunc, 71
- uip\_timeout
  - uipappfunc, 71
- uip\_udp\_bind
  - uipappfunc, 72
- uip\_udp\_conn, 122
- uip\_udp\_new
  - uip, 58
  - uipappfunc, 73
- uip\_udp\_periodic
  - uipdevfunc, 66
- uip\_udp\_periodic\_conn
  - uipdevfunc, 66
- uip\_udp\_remove
  - uipappfunc, 72
- uip\_udp\_send
  - uipappfunc, 72
- uip\_udpconnection
  - uipappfunc, 72
- uip\_unlisten
  - uip, 59
  - uipappfunc, 74
- uip\_urgdatalen
  - uipappfunc, 72
- uipappfunc
  - uip\_abort, 69
  - uip\_aborted, 69
  - uip\_acked, 70
  - uip\_close, 70
  - uip\_closed, 70
  - uip\_connect, 73
  - uip\_connected, 70
  - uip\_datalen, 70
  - uip\_listen, 73
  - uip\_mss, 70
  - uip\_newdata, 70
  - uip\_poll, 70
  - uip\_restart, 71
  - uip\_rexmit, 71
  - uip\_send, 71
  - uip\_stop, 71
  - uip\_timeout, 71
  - uip\_udp\_bind, 72
  - uip\_udp\_new, 73
  - uip\_udp\_remove, 72
  - uip\_udp\_send, 72
  - uip\_udpconnection, 72
  - uip\_unlisten, 74
  - uip\_urgdatalen, 72

Uiparch, [104](#)

uiparp

- [uip\\_arp\\_arpin](#), [81](#)
- [uip\\_arp\\_out](#), [81](#)
- [uip\\_arp\\_timer](#), [81](#)

uipconffunc

- [uip\\_getdraddr](#), [61](#)
- [uip\\_gethostaddr](#), [61](#)
- [uip\\_getnetmask](#), [61](#)
- [uip\\_setdraddr](#), [62](#)
- [uip\\_setethaddr](#), [62](#)
- [uip\\_sethostaddr](#), [62](#)
- [uip\\_setnetmask](#), [62](#)

uipconvfunc

- [HTONS](#), [76](#)
- [htons](#), [79](#)
- [uip\\_ipaddr](#), [76](#)
- [uip\\_ipaddr1](#), [76](#)
- [uip\\_ipaddr2](#), [76](#)
- [uip\\_ipaddr3](#), [76](#)
- [uip\\_ipaddr4](#), [77](#)
- [uip\\_ipaddr\\_cmp](#), [77](#)
- [uip\\_ipaddr\\_copy](#), [77](#)
- [uip\\_ipaddr\\_mask](#), [78](#)
- [uip\\_ipaddr\\_maskcmp](#), [78](#)
- [uiplib\\_ipaddrconv](#), [79](#)

uipdevfunc

- [uip\\_buf](#), [66](#)
- [uip\\_input](#), [64](#)
- [uip\\_periodic](#), [65](#)
- [uip\\_periodic\\_conn](#), [65](#)
- [uip\\_udp\\_periodic](#), [66](#)
- [uip\\_udp\\_periodic\\_conn](#), [66](#)

uipdns

- [resolv\\_conf](#), [83](#)
- [resolv\\_getserver](#), [84](#)
- [resolv\\_lookup](#), [84](#)
- [resolv\\_query](#), [84](#)

uipinit

- [uip\\_init](#), [63](#)

uiplib\_ipaddrconv

- [uipconvfunc](#), [79](#)

uipsplit

- [uip\\_split\\_output](#), [82](#)

vherr

- [uip\\_stats](#), [121](#)

www-conf.h.example

- [WWW\\_CONF\\_MAX\\_NUMPAGEWIDGETS](#), [129](#)

WWW\_CONF\_MAX\_NUMPAGEWIDGETS

- [www-conf.h.example](#), [129](#)