1.Introduction

Contiki is an open-source multitasking event-driven operating system designed for networked embedded devices. Its lightweight footprint makes it suitable for memory constraint microcontrollers.

Contiki gathers several independent modules such as an event-driven thread-like multitasking environment with the **protothread** library, the **uIP** TCP/IP (v4 and v6) stack, the wireless sensor network set of protocols: the **Rime** stack.
Contiki is primarily designed for networking applications, but can be used for any other application using only its event driven kernel.

You can visit the official website for more information.
This page describes the main functionality of Contiki, and what works on the WSN430 platform. The Contiki source files ported for the WSN430 platform are found in the download section. For setup howto and examples, see the Contiki example page.

Events

The Contiki kernel is event-driven. The idea of such a system is that every execution of a part of the application is a reaction to an event. The entire application (kernel + libraries + user code) may contain several processes that will execute concurrently.

The different processes usually execute for some time, then wait for events to happen. While waiting, a process is said to be blocked. When a event happen, the kernel execute the process passing it information about the event. The kernel is responsible for activating the processes when the events they're waiting for happen.

Events can be classified in three kinds:

- **timer events**: a process may set a timer to generate an event after a given time, it will block until the timer expires and then continue its execution. This is useful for periodic actions, or for networking protocols e.g. involving synchronization;
- **external events**: peripheral devices connected to IO pins of the microcontroller with interrupt capabilities may generate events when triggering interruptions. A push-button, a radio chip or a shock detector accelerometer are a few examples of devices that could generate interruptions, thus events. Processes may wait for such events to react accordingly.
- **internal events**: any process has the possibility to address events to any other process, or itself. This is useful for interprocess communication as informing a process that data is ready for computation.

Events are said posted. An interrupt service routine will post an event to a process when it is executed. Events have the following information:

- **process**: the process addressed by the event. It can be either one specific process or all the registered processes;
- **event type**: the type of event. The user can define some event types for the processes to differentiate them, such as one when a packet is received, one when a packet is sent;
- **data**: additionally, some data may be provided along with the event for the process.

This is the main principle of the Contiki Operating System: events are posted to processes, these execute when they receive them until they block waiting for another event.

Processes

Processes are the task-equivalent of Contiki. The process mechanism uses the underlying protothread library (website) which in turn uses the local continuation library (website). Refer to the given links for more information.
A process is a C function most likely containing an infinite loop and some blocking macro calls. Since the Contiki event-driven kernel is not preemptive, each process when executed will run until it blocks for an event. Several macros are defined for the different blocking possibilities. This allows programming state-machines as a sequential flow of control. Here is the skeleton of a Contiki process, as provided by the Contiki website:

```
#include "contiki.h"

/* The PROCESS() statement defines the process' name. */
PROCESS(my_example_process, "My example process");

/* The AUTOSTART_PROCESS() statement selects what process(es)
   to start when the module is loaded. */
AUTOSTART_PROCESSES(&my_example_process);

/* The PROCESS_THREAD() contains the code of the process. */
PROCESS_THREAD(my_example_process, ev, data)
{
  /* Do not put code before the PROCESS_BEGIN() statement -
     such code is executed every time the process is invoked. */
  PROCESS_BEGIN();
  /* Initialize stuff here. */
  while(1) {
    PROCESS_WAIT_EVENT();
    /* Do the rest of the stuff here. */
  }
  /* The PROCESS_END() statement must come at the end of the
     PROCESS_THREAD(). */
  PROCESS_END();
}
```

This code does obviously nothing, since it's a skeleton. It waits repeatedly for an event to happen, and again…

There are some special consideration to take care of when programming with protothreads (or Contiki processes):

- **local variables are not preserved**: when a process calls a blocking macro, what happens in fact is that the process function returns, letting the kernel call others. When an event is posted, the kernel will call the same process function which will jump right after it returned before. Thus if local variables of the process (but not declared `static`) were given some values before blocking, these values are not guaranteed to hold the same value when continuing after the block! A good way to work it around is to use **`static`** variables in the process function.
- **don't use switches**: protothreads use local continuations to find their state back after returning, which is done using a `switch` statement. If the `case` statements can be put almost anywhere (in an `if` or a `while` section), it can't be mixed up with another `switch` statement. Therefore it is better not to use **`switch`** statements inside a process function.

Please look at the examples section for some simple application showing the use of processes

uIP TCP/IP stack

Contiki contains a lightweight TCP/IP stack called uIP (uIP website). It implements RFC-compliant IPv4, IPv6, TCP and UDP (the latter two compatible with IPv4 and IPv6). uIP is very optimized, only the required features are implemented. For instance there is a single buffer for the whole stack, used for received packets as well as for those to send.

Application API

There are two ways to program an application on top of the uIP stack:

- **raw API**: the uIP raw API is well suited for implementing one simple application, e.g. a simple 'echo' server that would listen on some TCP port and send back every data it receives. However it becomes more complex to program when a more feature-full application is desired, or when two of these should run together. Even the TCP connection state machine is already a little pain.
- **protosocket API**: the protosocket library makes use of the protothread library for having a more flexible way to program TCP/IP applications. This library provides an interface similar to the standard BSD sockets, and allows programming the application in a process.

Refer to the examples section for some explained networking examples.

Lower Layers

Having a functional TCP/IP stack and some applications running on top of it is good, but not enough. The uIP stack requires a lower layer (according to the OSI model) in order to communicate with peers. We'll distinguish two different types of peers:

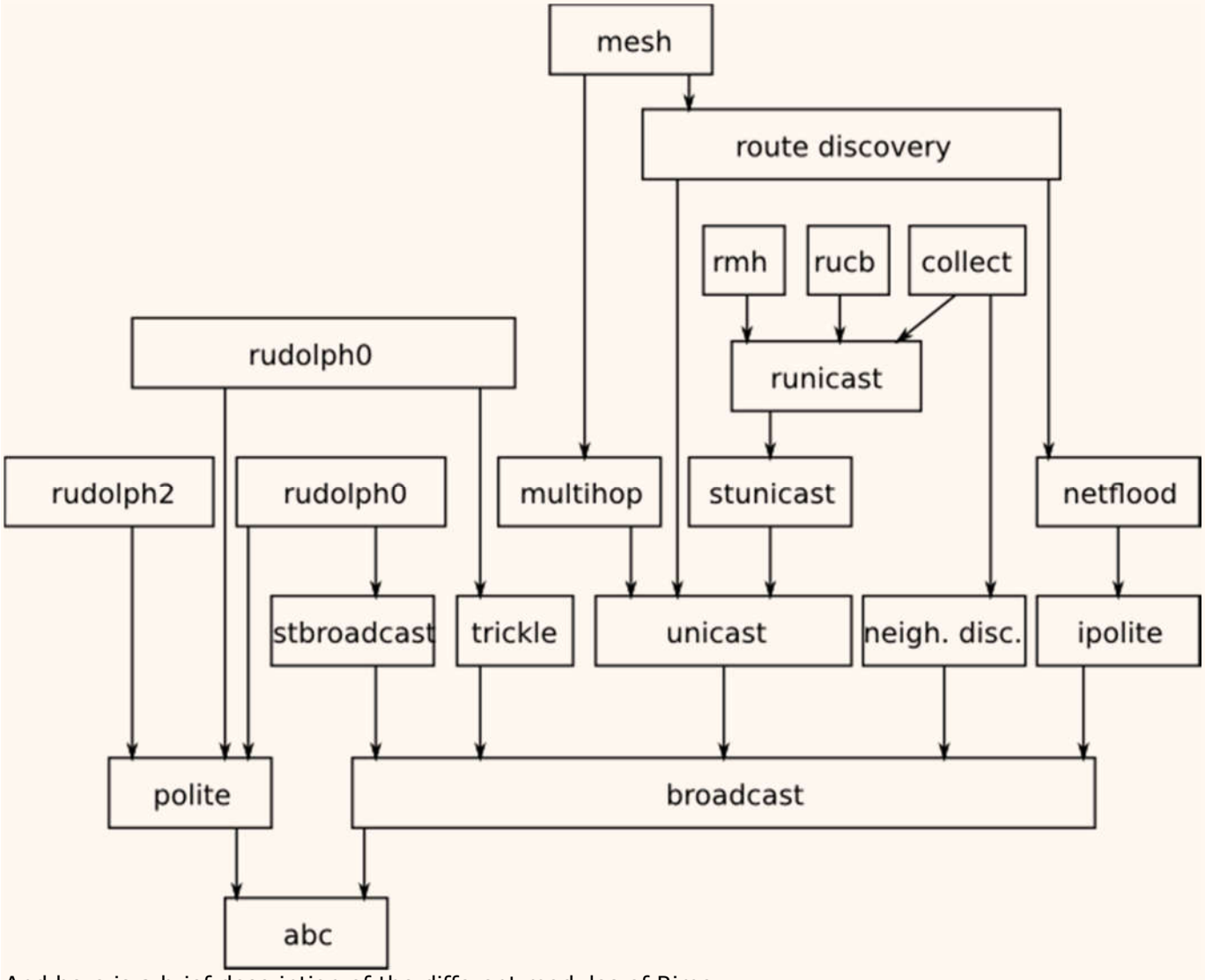
- **nodes**: communication between nodes is achieved with a wireless link. The uIP stack needs to be able to send and receive packets. Depending on the uIP version, Contiki follows different directions.
  - When it comes to IPv6, Contiki chose to follow a route-over configuration. Therefore, uIP6 uses a simple MAC layer called sicslowmac. Beside header compression provided by the 6loWPAN module, it just forwards the packet to/from the radio.
  - However, for IPv4, Contiki chose a mesh-under configuration. This is done with the Rime communication stack. Rime provide mesh routing and route discovery, therefore uIP uses it to forward packets on the network. From the IP point of view, all the nodes of the sensor network form a local subnetwork, even though multiple radio hops may be required.
- **gateways**: to reach a network entity outside the wireless sensor network, a gateway is required. It's a system that will make the link between the wireless sensors network and another network. It will typically be a PC in most experiments, although it could be many embedded system. The connection between a PC and a mote is a serial link. IP packets are sent between these two using **SLIP**, which stands for Serial Line IP. On the computer side, a program must run to do the interface between the serial line and a network interface. Depending on the uIP stack version, the functionality is not the same.
  - With uIPv6, a node will be loaded with a very simple program that forwards every packet from the radio to the serial link and vice versa. It doesn't do any address comparison, there is no IP stack on it, besides the header compression/decompression mechanism (6loWPAN). This

node will just be seen from the PC point of view as an ethernet network interface, thus that's the PC that does all the work.

- With uIPv4 it works differently. The node connected to the PC will act as a gateway, with all the IP stack in it. Every time it has a packet to send, it will check its IP address: if it belongs to the wireless sensor network subnet range, then it will send it using its radio, otherwise it will send it to the PC using the serial link. The PC runs a program that create a IP network interface.

Rime stack

The Rime stack provides a hierchical set of wireless network protocols, from a simple anonymous broadcaster to a mesh network routing. Implementing a complex protocol (say the multihop mesh routing) is split into several parts, where the more complex modules make use of the simpler ones.
Here is the overall organization of the Rime protocols:



And here is a brief description of the different modules of Rime:

- **abc:** the anonymous broadcast, it justs sends a packet via the radio driver, receives all packet from the radio driver and passes them to the upper layer;

- **broadcast:** the identified broadcast, it adds the sender address to the outgoing packet and passes it to the abc module;
- **unicast:** this module adds a destination address to the packets passed to the broadcast block. On the receive side, if the packet's destination address doesn't match the node's address, the packet is discarded;
- **stunicast:** the stubborn unicast, when asked to send a packet to a node, it sends it repeatedly with a given time period until asked to stop. This module is usually not used as is, but is used by the next one;
- **runicast:** the reliable unicast, it sends a packet using the stunicast module waiting for an acknowledgment packet. When it is received it stops the continuous transmission of the packet. A maximum retransmission number must be specified, in order to avoid infinite sending;
- **polite** and **ipolite:** these two modules are almost identical, when a packet has to be sent in a given time frame, the module waits for half of the time, checking if it has received the same packet it's about to send. If it has, the packet is not sent, otherwise it sends the packet. This is useful for flooding techniques to avoid unnecessary retransmissions;
- **multihop:** this module requires a route table function, and when about to send a packet it asks the route table for the next hop and sends the packet to it using unicast. When it receives a packet, if the node is the destination then the packet is passed to the upper layer, otherwise it asks again the route table for the next hop and relays the packet to it;

MAC layer

As we've seen, the uIPv6 stack use a very simple MAC layer called 'sicslomac', and there is no real option here. However the uIPv4 stack uses Rime as its immediately lower layer, which in turn uses a selectable MAC layer.

There are a few MAC layers implemented in Contiki:

- the nullmac protocol, as its name may say does nothing but forwarding packets from the upper layer to the radio driver, and vice versa;
- the xmac protocol, which is a preamble sampling protocol, where nodes listen on the radio channel for a small amount of time, periodically.
- the lpp protocol, which is a probing protocol, where the nodes periodically send a small message saying they're listening, they listen for a short time afterward and go back to sleep.

Working so far

What has been ported and tested so far on the WSN430 platform follows:

- **kernel**: the basic Contiki kernel works without problem, including the processes handling, the different timers management, etc… Furthermore, all the drivers required by Contiki should be

provided by the WSN430 drivers package. In the Contiki original packaging, those were part of the system, but for maintainability reasons they have been separated.

- **uIP**: the uIP stack works with no problem for both IPv4 and IPv6 versions. Both TCP and UDP are available, but care should be taken with UDP because of the Maximum Segment Size, which is usually set to 128 octets. The communications work directly between nodes using the radio link, or with a PC using a node gateway.
- **Rime**: the wireless sensor network set of protocols has been partly tested, and the simpler modules are working, such as abc, broadcast, unicast, ipolite, stunicast, runicast, multihop. The global communication stack, that is using the *mesh* module has been proven to work with a few number of nodes in the same radio neighborhood. Multihop and routing has not been tested.
- **MAC**: the four previously mentioned MAC protocols have been configured for the WSN430 platform: nullmac, xmac, lpp and sicslomac.

2. Contiki Application Examples

This page describes different application examples for Contiki, detailing its main features, whether it's about processes or communication stacks.

In order to compile any Contiki application, the environment variable `WSN430_DRIVERS_PATH` must point to the wsn430 drivers location.

Hello World

The 01_hello-world example implements two concurrent processes. The first is called "hello_world" and will periodically print "Hello World" and an increasing counter on the serial port. The second is called "blink" and will blink the LEDs at a higher rate than the other process.

The source files for this example is found in the `example/01_hello-world` folder. The `hello-world.c` file is the application main file, it contains all the application specific code and the `Makefile` indicates how to build the application.
Here is the source code of the application:

```
1:  #include "contiki.h"
2:  #include "dev/leds.h"
3:
4:  #include <stdio.h> /* For printf() */
5:  /*---------------------------------------------------------------------------*/
6:  /* We declare the two processes */
7:  PROCESS(hello_world_process, "Hello world process");
8:  PROCESS(blink_process, "LED blink process");
9:
10: /* We require the processes to be started automatically */
11: AUTOSTART_PROCESSES(&hello_world_process, &blink_process);
12: /*---------------------------------------------------------------------------*/
13: /* Implementation of the first process */
14: PROCESS_THREAD(hello_world_process, ev, data)
15: {
16:     // variables are declared static to ensure their values are kept
17:     // between kernel calls.
18:     static struct etimer timer;
19:     static int count = 0;
20:
21:     // any process mustt start with this.
```

```
22:     PROCESS_BEGIN();
23:
24:     // set the etimer module to generate an event in one second.
25:     etimer_set(&timer, CLOCK_CONF_SECOND);
26:     while (1)
27:     {
28:         // wait here for an event to happen
29:         PROCESS_WAIT_EVENT();
30:
31:         // if the event is the timer event as expected...
32:         if(ev == PROCESS_EVENT_TIMER)
33:         {
34:             // do the process work
35:             printf("Hello, world #%i\n", count);
36:             count ++;
37:
38:             // reset the timer so it will generate an other event
39:             // the exact same time after it expired (periodicity guaranteed)
40:             etimer_reset(&timer);
41:         }
42:
43:         // and loop
44:     }
45:     // any process must end with this, even if it is never reached.
46:     PROCESS_END();
47: }
48: /*---------------------------------------------------------------------------*/
49: /* Implementation of the second process */
50: PROCESS_THREAD(blink_process, ev, data)
51: {
52:     static struct etimer timer;
53:     static uint8_t leds_state = 0;
54:     PROCESS_BEGIN();
55:
56:     while (1)
57:     {
58:         // we set the timer from here every time
59:         etimer_set(&timer, CLOCK_CONF_SECOND / 4);
60:
61:         // and wait until the vent we receive is the one we're waiting for
62:         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
63:
64:         // update the LEDs
65:         leds_off(0xFF);
66:         leds_on(leds_state);
67:         leds_state += 1;
68:     }
69:     PROCESS_END();
70: }
71: /*---------------------------------------------------------------------------*/
```

hello-world.c

The code is rather fully documented, but let's see once more how processes are made:

- First the process must be declared using the macro: `PROCESS(name, strname)`. The arguments `name` and `strname` are the variable name and the process described name respectively. You will refer to the process using the process `name`.
- The process definition is a function whose definition is the macro: `PROCESS_THREAD(name, ev, data)`. The `name` argument should be replaced by the process `name` specified in the `PROCESS` macro. `ev` and `data` should be let as they are, those will be variables that will indicate

what event generated the process execution (`ev`) and a pointer to optional data passed as the event was generated (`data`).

- The process function may declare some variables, then must start with the macro `PROCESS_BEGIN()` and end with `PROCESS_END()`. These two macro work together, as they in-fact implement a switch-statement. The code placed before `PROCESS_BEGIN()` will be executed every time the process is executed (or scheduled), which is most likely not wanted since the process is supposed to continue from where it blocked. Between these two macros is the application user code.
- The process code once executed will run until a specific macro is reached indicating a wait for an event. Here are some of the blocking macro calls you can make:
  - `PROCESS_WAIT_EVENT()`: yield and wait for an event to be posted to this process;
  - `PROCESS_WAIT_EVENT_UNTIL(cond)`: yield and wait for an event, and for the `cond` condition to be true;
  - `PROCESS_WAIT_UNTIL(cond)`: wait until the `cond` condition is true. If it is true at the instant of call, does not yield;
  - `PROCESS_WAIT_WHILE(cond)`: wait while the `cond` condition is true. If it is false at the instant of call, does not yield;
  - `PROCESS_PAUSE()`: post an event to itself and yield. This allows the kernel to execute other processes if there are events pending, and ensures continuation of this process afterward;
  - `PROCESS_EXIT()`: terminate the process.

Event Post

This second example called 02_event-post implements two processes: the first reads a temperature sensor periodically, and averages the measures over a given number of samples. When a new measure is ready, this process posts an event to another process in charge of printing the result over a serial link. Not only the event is posted (which is a notification), but a pointer to the measured data is passed to the other process as well.

The source files for this example are located in the `examples/02_event-post` folder. The main source file is `event-post.c`. Here is the code for this example:

```
1:  #include "contiki.h"
2:  #include "dev/leds.h"
3:
4:  #include <stdio.h> /* For printf() */
5:
6:  /* Driver Include */
7:  #include "ds1722.h"
8:
9:  /* Variables: the application specific event value */
10: static process_event_t event_data_ready;
11:
12: /*---------------------------------------------------------------------------*/
13: /* We declare the two processes */
14: PROCESS(temp_process, "Temperature process");
15: PROCESS(print_process, "Print process");
16:
17: /* We require the processes to be started automatically */
18: AUTOSTART_PROCESSES(&temp_process, &print_process);
19: /*---------------------------------------------------------------------------*/
20: /* Implementation of the first process */
21: PROCESS_THREAD(temp_process, ev, data)
```

```
22: {
23:     // variables are declared static to ensure their values are kept
24:     // between kernel calls.
25:     static struct etimer timer;
26:     static int count = 0;
27:     static int average, valid_measure;
28:
29:     // those 3 variables are recomputed at every run, therefore it is not
30:     // necessary to declare them static.
31:     int measure;
32:     uint8_t msb, lsb;
33:
34:     // any process mustt start with this.
35:     PROCESS_BEGIN();
36:
37:     /* allocate the required event */
38:     event_data_ready = process_alloc_event();
39:
40:     /* Initialize the temperature sensor */
41:     ds1722_init();
42:     ds1722_set_res(10);
43:     ds1722_sample_cont();
44:
45:     average = 0;
46:
47:     // set the etimer module to generate an event in one second.
48:     etimer_set(&timer, CLOCK_CONF_SECOND/4);
49:
50:     while (1)
51:     {
52:         // wait here for the timer to expire
53:         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
54:
55:         leds_toggle(LEDS_BLUE);
56:
57:         // do the process work
58:         msb = ds1722_read_MSB();
59:         lsb = ds1722_read_LSB();
60:
61:         measure = ((uint16_t)msb) << 2;
62:         measure += (lsb >> 6) & 0x03;
63:
64:         average += measure;
65:         count ++;
66:
67:         if (count == 4)
68:         {
69:             // average the sum and store
70:             valid_measure = average >> 2;
71:
72:             // reset variables
73:             average = 0;
74:             count = 0;
75:
76:             // post an event to the print process
77:             // and pass a pointer to the last measure as data
78:             process_post(&print_process, event_data_ready, &valid_measure);
79:         }
80:
81:         // reset the timer so it will generate another event
82:         etimer_reset(&timer);
83:     }
84:     // any process must end with this, even if it is never reached.
85:     PROCESS_END();
86: }
```

```
87: /*---------------------------------------------------------------------------*/
88: /* Implementation of the second process */
89: PROCESS_THREAD(print_process, ev, data)
90: {
91:     PROCESS_BEGIN();
92:
93:     while (1)
94:     {
95:         // wait until we get a data_ready event
96:         PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready);
97:
98:         // display it
99:         printf("temperature = %u.%u\n", (*(int*)data)>>2, ((*(int*)data)&0x3)*25);
100:
101:    }
102:    PROCESS_END();
103: }
104: /*---------------------------------------------------------------------------*/
```

event-post.c

The main difference with the previous example is that the two processes interact with each other. This is achieved thanks to events, which allow waking up processes while indicating them why they have been woken and optional data is passed to them.

There is a certain number of system-defined events, here is the list:

- PROCESS_EVENT_NONE
- PROCESS_EVENT_INIT
- PROCESS_EVENT_POLL
- PROCESS_EVENT_EXIT
- PROCESS_EVENT_SERVICE_REMOVED
- PROCESS_EVENT_CONTINUE
- PROCESS_EVENT_MSG
- PROCESS_EVENT_EXITED
- PROCESS_EVENT_TIMER
- PROCESS_EVENT_COM
- PROCESS_EVENT_MAX

These events are used by the kernel internally, but may be used in an application with care. Otherwise new events can be defined using the following function:

- `process_event_t process_alloc_event (void)`

Hence declaring a global variable that will receive the new event value can then be used amongst processes, as it is done in the example.

Process interaction is achieved with the following functions:

- `void process_start (struct process *p, const char *arg)`: this adds a process to the kernel active process list. Processes may be started automatically using the `AUTOSTART_PROCESS` macro (see the examples for how to use it), or started manually using this function. The argument `p` points

to the process variable (declared with the `PROCESS` macro), and `arg` is the data the process will receive for its initialization call.

- `void process_exit (struct process *p)`: this stops a process and removes it from the kernel active process list. What is really done is: all the other processes will receive an event informing them the process is about to exit, then the process itself is run a last time with the event 'PROCESS_EVENT_EXIT' to let it end correctly, and finally the process is removed from the kernel list.
- `int process_post (struct process *p, process_event_t ev, void *data)`: this function is the most generic, it posts an event to a given process, specifying the event type and a pointer to the data passed. The value returned is `PROCESS_ERR_OK` if the event is posted, or `PROCESS_ERR_FULL` if the event queue was full and the event could not be posted. If the event is posted, the kernel will execute the process when it will have done so with all the other processes with an event pending.
- `void process_post_synch (struct process *p, process_event_t ev, void *data)`: this function is similar to the previous one, except that if the event can be posted the process will execute now with the event. In other word, it will skip the event waiting queue, and when it returns, the calling process will continue its execution until the next blocking macro call.
- `void process_poll (struct process *p)`: this last function sets the process to be requiring a poll. A poll request is a little different from a event post, because before processing an event, the kernel first processes all the poll request for all the running processes. When it does, the polled process is executed with a `PROCESS_EVENT_POLL` event and no data.

Echo Server (TCP)

This third example brings several new features at once. We'll see here how to set up the uIP stack (IPv4 only for now), how to set up SLIP (Serial Line IP) for enabling IP connectivity between a PC running Linux and a WSN430 with a serial link, and how to implement a simple TCP echo server using the protosocket library.

This example is found in the `example/03_echo-server` folder, and contains the following files: `echo-server.c` for the echo server code, and the `Makefile` to build the project. Let's get started…

uIP setup

In order to enable uIP, adding `CFLAGS += -DWITH_UIP=1` in the `Makefile` file of the project folder is all what's required. But for your information, let's see what is done to make uIP functional. The networking setup is executed in the `platform/wsn430/contiki-wsn430-init-net.c` file.

- **initialize the modules**: the `uip` and `uip_fw` modules must be initialized by starting their processes:

```
process_start(&tcpip_process, NULL);
process_start(&uip_fw_process, NULL);
```

- **set the node IP address**: the node IP address is set according to its Rime address. The Rime module is initialized before, and its address is obtained from the DS2411 unique identifier value. The subnetwork address is hardcoded to 172.16.0.0/16, and the two lower bytes only are picked from the Rime address:

```
uip_ipaddr_t hostaddr, netmask;
```

```
uip_init();

uip_ipaddr(&hostaddr, 172,16,
     rimeaddr_node_addr.u8[0],rimeaddr_node_addr.u8[1]);
uip_ipaddr(&netmask, 255,255,0,0);
uip_ipaddr_copy(&meshif.ipaddr, &hostaddr);

uip_sethostaddr(&hostaddr);
uip_setnetmask(&netmask);
uip_over_mesh_set_net(&hostaddr, &netmask);
```

- **define some network interfaces**: the network interfaces are bound to subnetworks. A uIP network interface contains a subnetwork IP address, a netmask an a function used to send the IP packet. When using the radio interface to send IP packets, the Rime module is used. In order to send IP packet to a gateway, the SLIP is used to send them on a serial link. ATherefore the network interfaces are defined as follows:

```
static struct uip_fw_netif meshif =
    {UIP_FW_NETIF(172,16,1,0, 255,255,0,0, uip_over_mesh_send)};
static struct uip_fw_netif slipif =
    {UIP_FW_NETIF(0,0,0,0, 0,0,0,0, slip_send)};
```

- **register the interfaces**: once the network interfaces are defined as described above, they must be registered to the `uip` module:

```
uip_over_mesh_set_gateway_netif(&slipif);
uip_fw_default(&meshif);
```

This setup allows the nodes to have 2 network interfaces one for radio communications and one other for serial line communications.

SLIP setup

SLIP (Serial Line IP) is build with uIP in every node, but is not activated by default. Only nodes that will receive SLIP packets on their serial line will activate the module, and start using this new interface.

If you use the `tunslip` program found in the `tools/` directory, you will be able to enable SLIP on the connected node, and send IP packets to this one and the others with the forwarding capability of uIP.
The `tunslip.sh` script lauches `tunslip` with the default parameters.

Echo server implementation

The uIP stack is up and running on the gateway node, we have the tunslip network interface ready on the computer, we just have to write the application! The code is in the `echo-server.c` file.
The echo server will use a TCP connection. It will listen on port 12345, waiting for connection. Once one is established, it will send a welcome message, then wait for incoming data. Once it has received some it will send it back to the sender and closes the connection.

Here is the code:

```
 1: #include "contiki.h"
 2: #include "contiki-net.h"
 3:
 4: #include "dev/leds.h"
 5:
 6: #include <stdio.h>
 7: #include <string.h>
 8:
 9: /*
10:  * We define one protosocket since we've decided to only handle one
```

```
11:  * connection at a time. If we want to be able to handle more than one
12:  * connection at a time, each parallell connection needs its own
13:  * protosocket.
14:  */
15:
16: static struct psock ps;
17: /*
18:  * We must have somewhere to put incoming data, and we use a 50 byte
19:  * buffer for this purpose.
20:  */
21: static char buffer[50];
22:
23: /*---------------------------------------------------------------------------*/
24: /*
25:  * A protosocket always requires a protothread. The protothread
26:  * contains the code that uses the protosocket. We define the
27:  * protothread here.
28:  */
29: static PT_THREAD(handle_connection(struct psock *p))
30: {
31:   /*
32:    * A protosocket's protothread must start with a PSOCK_BEGIN(), with
33:    * the protosocket as argument.
34:    *
35:    * Remember that the same rules as for protothreads apply: do NOT
36:    * use local variables unless you are very sure what you are doing!
37:    * Local (stack) variables are not preserved when the protothread
38:    * blocks.
39:    */
40:   PSOCK_BEGIN(p);
41:
42:   /*
43:    * We start by sending out a welcoming message. The message is sent
44:    * using the PSOCK_SEND_STR() function that sends a null-terminated
45:    * string.
46:    */
47:   PSOCK_SEND_STR(p, "Welcome, please type something and press return.\n");
48:
49:   /*
50:    * Next, we use the PSOCK_READTO() function to read incoming data
51:    * from the TCP connection until we get a newline character. The
52:    * number of bytes that we actually keep is dependant of the length
53:    * of the input buffer that we use. Since we only have a 10 byte
54:    * buffer here (the buffer[] array), we can only remember the first
55:    * 10 bytes received. The rest of the line up to the newline simply
56:    * is discarded.
57:    */
58:   PSOCK_READTO(p, '\n');
59:
60:   /*
61:    * And we send back the contents of the buffer. The PSOCK_DATALEN()
62:    * function provides us with the length of the data that we've
63:    * received. Note that this length will not be longer than the input
64:    * buffer we're using.
65:    */
66:   PSOCK_SEND_STR(p, "Got the following data: ");
67:   PSOCK_SEND(p, buffer, PSOCK_DATALEN(p));
68:   PSOCK_SEND_STR(p, "Good bye!\r\n");
69:
70:   /*
71:    * We close the protosocket.
72:    */
73:   PSOCK_CLOSE(p);
```

```
74:
75:   /*
76:    * And end the protosocket's protothread.
77:    */
78:   PSOCK_END(p);
79: }
80: /*---------------------------------------------------------------------------*/
81: /*
82:  * We declare the process.
83:  */
84: PROCESS(example_psock_server_process, "Example protosocket server");
85: AUTOSTART_PROCESSES(&example_psock_server_process);
86: /*---------------------------------------------------------------------------*/
87: /*
88:  * The definition of the process.
89:  */
90: PROCESS_THREAD(example_psock_server_process, ev, data)
91: {
92:   /*
93:    * The process begins here.
94:    */
95:   PROCESS_BEGIN();
96:
97:   /*
98:    * We start with setting up a listening TCP port. Note how we're
99:    * using the HTONS() macro to convert the port number (1234) to
100:   * network byte order as required by the tcp_listen() function.
101:   */
102:  tcp_listen(HTONS(12345));
103:
104:  /*
105:   * We loop for ever, accepting new connections.
106:   */
107:  while(1) {
108:
109:    /*
110:     * We wait until we get the first TCP/IP event, which probably
111:     * comes because someone connected to us.
112:     */
113:    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
114:
115:    /*
116:     * If a peer connected with us, we'll initialize the protosocket
117:     * with PSOCK_INIT().
118:     */
119:    if(uip_connected()) {
120:
121:      /*
122:       * The PSOCK_INIT() function initializes the protosocket and
123:       * binds the input buffer to the protosocket.
124:       */
125:      PSOCK_INIT(&ps, buffer, sizeof(buffer));
126:
127:      /*
128:       * We loop until the connection is aborted, closed, or times out.
129:       */
130:      while(!(uip_aborted() || uip_closed() || uip_timedout())) {
131:
132:        /*
133:         * We wait until we get a TCP/IP event. Remember that we
134:         * always need to wait for events inside a process, to let
135:         * other processes run while we are waiting.
136:         */
137:        PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
```

```
138:
139:           /*
140:            * Here is where the real work is taking place: we call the
141:            * handle_connection() protothread that we defined above. This
142:            * protothread uses the protosocket to receive the data that
143:            * we want it to.
144:            */
145:           handle_connection(&ps);
146:         }
147:       }
148:     }
149:
150:     /*
151:      * We must always declare the end of a process.
152:      */
153:     PROCESS_END();
154: }
155: /*---------------------------------------------------------------------------*/
```

echo-server.c

You can now compile the project, and program a device with it.

For SLIP, assuming you're using the `/dev/ttyS0` serial port of your computer, type `make` then run `./tunslip.sh` as root in the `tools`directory. You should observe the following:

```
[root@brigand tools]# ./tunslip.sh
./tunslip -s /dev/ttyS0 -B 115200 172.16.0.0 255.255.0.0
slip started on ``/dev/ttyS0''
opened device ``/dev/tun0''
ifconfig tun0 inet `hostname` up
route add -net 172.16.0.0 netmask 255.255.0.0 dev tun0
ifconfig tun0

tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:194.199.21.252  P-t-P:194.199.21.252  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

This is waiting for a connection from the WSN430. Power on the WSN430 you've just programmed, already plugged on the serial link, you should see:

```
Contiki 2.3 started. Rime address is 1.109
MAC ff:00:00:12:91:bf:6d:01
 X-MAC channel 26
uIP started with IP address 172.16.1.109
Starting 'Example protosocket server'
1.109: making myself the IP network gateway.
IPv4 address of the gateway: 172.16.1.109
```

```
route add -net 172.16.1.109 netmask 255.255.255.255 dev tun0
```

You have the IP of the node (172.16.1.109 here) printed. You can now either ping the device (`ping 172.16.1.109`) or connect to the server we've created using netcat (`nc 172.16.1.109 12345`)

Echo Server (UDP)

The example found in the `examples/04_echo-udp` folder implements a simple UDP server application. The source code is found in the `echo-serve.c` file.
UDP is connectionless, therefore programming a UDP application is rather simple. Only two things can be done with UDP: sending and receiving datagrams. Contiki uses the word 'connection' both for UDP and TCP, but in the sense of a BSD socket. It is an endpoint for Internet communication. Here are the different functions/macros used for programming using UDP:

- `struct uip_udp_conn * **udp_new** (const uip_ipaddr_t *ripaddr, u16_t port, void *appstate)`: this function creates a new UDP 'connection'. If the connection is used to send data to a specific host, the arguments `ripaddr` (remote IP address) and `port` must be filled adequately, otherwise `ripaddr` may be set to `NULL` and port to 0 in order to accept any incoming datagram. `appstate` is a pointer to some data that will be passed to the process when UDP events will occur. The function returns a pointer to the connection, that will be used by all the other functions afterward. The function chooses a local port number arbitrarily, which can be overwritten as explained below;
- **`udp_bind`**`(conn, port)`: this macro binds an open UDP connection to a local port number. The specified port will be the source port used to send datagrams as well as the listening port for incoming datagrams;
- **`uip_newdata`**`()`: this macro indicates if there is data available on the connection. If so it can be accessed from the pointer `uip_appdata` and the length obtained by `uip_datalen()`.

The Echo Server UDP uses most of these functions. It listens on UDP port 50000 and replies back incoming datagrams. The complete source code of the application is here:

```
1:  /**
2:   * \file
3:   *         A network application that listens on UDP port 50000 and echoes.
4:   * \author
5:   *         Clément Burin des Roziers <clement.burin-des-roziers@inrialpes.fr>
6:   */
7:
8:  #include "contiki.h"
9:  #include "contiki-net.h"
10:
11: #include "dev/leds.h"
12:
13: #include <stdio.h>
14: #include <string.h>
15:
16: #define PRINTF printf
17:
18: #define UDP_DATA_LEN 120
19: #define UDP_HDR ((struct uip_udpip_hdr *)&uip_buf[UIP_LLH_LEN])
20:
21:
22: static struct uip_udp_conn *udpconn;
23: static uint8_t udpdata[UDP_DATA_LEN] = "rx=";
24:
25: static void udphandler(process_event_t ev, process_data_t data)
```

```
26: {
27:     if (uip_newdata())
28:     {
29:         /* Set the last byte of the received data as 0 in order to print it. */
30:         int len = uip_datalen();
31:         ((char *)uip_appdata)[len] = 0;
32:           printf("Received from %u.%u.%u.%u:%u: '%s'\n", uip_ipaddr_to_quad(&UDP_HDR-
>srcipaddr), HTONS(UDP_HDR->srcport), (char*)uip_appdata);
33: 
34:         /* Prepare the response datagram in a local buffer */
35:         memcpy(udpdata, "rx=", 3);
36:         memcpy(udpdata+3, uip_appdata, len);
37: 
38:         /* Copy the information about the sender to the udpconn in order to reply */
39:         uip_ipaddr_copy(&udpconn->ripaddr , &UDP_HDR->srcipaddr); // ip address
40:         udpconn->rport = UDP_HDR->srcport; // UDP port
41: 
42:         /* Send the reply datagram */
43:         printf("sending back\n");
44:         uip_udp_packet_send(udpconn, udpdata, uip_datalen()+3);
45: 
46:         /* Restore the udpconn to previous setting in order to receive other packets */
47:         uip_ipaddr_copy(&udpconn->ripaddr , &uip_all_zeroes_addr);
48:         udpconn->rport = 0;
49:     }
50: }
51: 
52: /*---------------------------------------------------------------------------*/
53: /*
54:  * We declare the process.
55:  */
56: PROCESS(example_udp_server_process, "Example UDP server");
57: AUTOSTART_PROCESSES(&example_udp_server_process);
58: /*---------------------------------------------------------------------------*/
59: /*
60:  * The definition of the process.
61:  */
62: PROCESS_THREAD(example_udp_server_process, ev, data)
63: {
64: 
65:     uip_ipaddr_t ipaddr;
66: 
67:     PROCESS_BEGIN();
68:     printf("UDP Echo Server test\n");
69: 
70:     /* Create a UDP 'connection' with IP 0.0.0.0 and port 0 as remote host.
71:      * This means the stack will accepts UDP datagrams from any node. */
72:     udpconn = udp_new(NULL, HTONS(0), NULL);
73: 
74:     /* Bind the UDP 'connection' to the port 50000. That's the port we're listening on.
*/
75:     udp_bind(udpconn, HTONS(50000));
76: 
77:     printf("listening on UDP port %u\n", HTONS(udpconn->lport));
78: 
79:     while(1) {
80:         /* Wait until we have an event caused by tcpip interaction */
81:         PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
82:         /* Handle it */
83:         udphandler(ev, data);
84:     }
85: 
86:     PROCESS_END();
87: }
```

```
88: /*---------------------------------------------------------------------------*/
```

echo-server.c

The application starts in the process definition. First a UDP connection is created, specifying remote IP address and port as 0, indicating it's a listening connection. Then the local UDP port is bound to 50000, which means the connection listens on that port. Finally, an infinite loop is entered waiting for TCPIP events, and calling a handler function when these occur.

The handler function checks if new data is available from the uIP buffer. If so, it prints the received data for debug as well as the sender IP address and port, then prepare a response datagram by prepending "rx=" to the received message. The connection information must be updated in order to send the response to the emitter of the message. Therefore the `ripaddr` and `rport` fields of the UDP connection structure are filled with the sender information extracted from the received IP packet. Then the function for sending the packet is invoked. Finally the UDP connection structure is updated back to its original state (with remote IP address and port set to 0) in order to be able to receive datagrams from any host.
Run tunslip as in the previous example and execute `nc -u 176.12.xxx.xxx 50000` in a terminal to test the echo server.

IPv6 UDP Communication

The examples found in the `05_ipv6` folder implement a UDP-server and a UDP-client applications, using UDP and IPv6. The first example will run on a node, listening on a given UDP port and waiting fir datagrams to arrive. The second example will send datagrams to the server periodically.
To enable IPv6, the following line must be added to the project's Makefile:

```
UIP_CONF_IPV6=1
```

Here is the code for the server example:

```
 1: /*
 2:  * Redistribution and use in source and binary forms, with or without
 3:  * modification, are permitted provided that the following conditions
 4:  * are met:
 5:  * 1. Redistributions of source code must retain the above copyright
 6:  *    notice, this list of conditions and the following disclaimer.
 7:  * 2. Redistributions in binary form must reproduce the above copyright
 8:  *    notice, this list of conditions and the following disclaimer in the
 9:  *    documentation and/or other materials provided with the distribution.
10:  * 3. Neither the name of the Institute nor the names of its contributors
11:  *    may be used to endorse or promote products derived from this software
12:  *    without specific prior written permission.
13:  *
14:  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
15:  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16:  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17:  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
18:  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
19:  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
20:  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
21:  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
22:  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
23:  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
24:  * SUCH DAMAGE.
25:  *
26:  * This file is part of the Contiki operating system.
27:  *
28:  */
29:
30: #include "contiki.h"
31: #include "contiki-lib.h"
32: #include "contiki-net.h"
33:
```

```
34: #include <string.h>
35:
36: #define DEBUG 1
37: #if DEBUG
38: #include <stdio.h>
39: #define PRINTF(...) printf(__VA_ARGS__)
40:                     #define                    PRINT6ADDR(addr)                    PRINTF("
%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x      ",      ((u8_t
*)addr)[0], ((u8_t *)addr)[1], ((u8_t *)addr)[2], ((u8_t *)addr)[3], ((u8_t *)addr)[4],
((u8_t  *)addr)[5],  ((u8_t  *)addr)[6],  ((u8_t  *)addr)[7],  ((u8_t  *)addr)[8],  ((u8_t
*)addr)[9], ((u8_t *)addr)[10], ((u8_t *)addr)[11], ((u8_t *)addr)[12], ((u8_t *)addr)[13],
((u8_t *)addr)[14], ((u8_t *)addr)[15])
41: #define PRINTLLADDR(lladdr) PRINTF(" %02x:%02x:%02x:%02x:%02x:%02x ",(lladdr)->addr[0],
(lladdr)->addr[1],   (lladdr)->addr[2],   (lladdr)->addr[3],(lladdr)->addr[4],   (lladdr)-
>addr[5])
42: #else
43: #define PRINTF(...)
44: #define PRINT6ADDR(addr)
45: #define PRINTLLADDR(addr)
46: #endif
47:
48: #define UDP_IP_BUF   ((struct uip_udpip_hdr *)&uip_buf[UIP_LLH_LEN])
49:
50: #define MAX_PAYLOAD_LEN 120
51:
52: static struct uip_udp_conn *server_conn;
53:
54: PROCESS(udp_server_process, "UDP server process");
55: AUTOSTART_PROCESSES(&udp_server_process);
56: /*---------------------------------------------------------------------------*/
57: static void
58: tcpip_handler(void)
59: {
60:   static int seq_id;
61:   char buf[MAX_PAYLOAD_LEN];
62:
63:   if(uip_newdata()) {
64:     ((char *)uip_appdata)[uip_datalen()] = 0;
65:     PRINTF("Server received: '%s' from ", (char *)uip_appdata);
66:     PRINT6ADDR(&UDP_IP_BUF->srcipaddr);
67:     PRINTF("\n");
68:
69:     uip_ipaddr_copy(&server_conn->ripaddr, &UDP_IP_BUF->srcipaddr);
70:     server_conn->rport = UDP_IP_BUF->srcport;
71:     PRINTF("Responding with message: ");
72:     sprintf(buf, "Hello from the server! (%d)", ++seq_id);
73:     PRINTF("%s\n", buf);
74:
75:     uip_udp_packet_send(server_conn, buf, strlen(buf));
76:     /* Restore server connection to allow data from any node */
77:     memset(&server_conn->ripaddr, 0, sizeof(server_conn->ripaddr));
78:     server_conn->rport = 0;
79:   }
80: }
81: /*---------------------------------------------------------------------------*/
82: static void
83: print_local_addresses(void)
84: {
85:   int i;
86:   uip_netif_state state;
87:
88:   PRINTF("Server IPv6 addresses: \n");
89:   for(i = 0; i < UIP_CONF_NETIF_MAX_ADDRESSES; i++) {
90:     state = uip_netif_physical_if.addresses[i].state;
```

```
91:     if(state == TENTATIVE || state == PREFERRED) {
92:       PRINT6ADDR(&uip_netif_physical_if.addresses[i].ipaddr);
93:       PRINTF("\n");
94:     }
95:   }
96: }
97: /*---------------------------------------------------------------------------*/
98: PROCESS_THREAD(udp_server_process, ev, data)
99: {
100:   static struct etimer timer;
101:
102:   PROCESS_BEGIN();
103:   PRINTF("UDP server started\n");
104:
105:   // wait 3 second, in order to have the IP addresses well configured
106:   etimer_set(&timer, CLOCK_CONF_SECOND*3);
107:
108:   // wait until the timer has expired
109:   PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
110:
111:   print_local_addresses();
112:
113:    // set NULL and 0 as IP address and port to accept packet from any node and any
srcport.
114:   server_conn = udp_new(NULL, HTONS(0), NULL);
115:   udp_bind(server_conn, HTONS(3000));
116:
117:   PRINTF("Server listening on UDP port %u\n", HTONS(server_conn->lport));
118:
119:   while(1) {
120:     PROCESS_YIELD();
121:     if(ev == tcpip_event) {
122:       tcpip_handler();
123:     }
124:   }
125:
126:   PROCESS_END();
127: }
128: /*---------------------------------------------------------------------------*/
```

udp-server.c

And here's the code for the client example:

```
1: /*
2:  * Redistribution and use in source and binary forms, with or without
3:  * modification, are permitted provided that the following conditions
4:  * are met:
5:  * 1. Redistributions of source code must retain the above copyright
6:  *    notice, this list of conditions and the following disclaimer.
7:  * 2. Redistributions in binary form must reproduce the above copyright
8:  *    notice, this list of conditions and the following disclaimer in the
9:  *    documentation and/or other materials provided with the distribution.
10:  * 3. Neither the name of the Institute nor the names of its contributors
11:  *    may be used to endorse or promote products derived from this software
12:  *    without specific prior written permission.
13:  *
14:  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
15:  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16:  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17:  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
18:  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
19:  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
20:  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
21:  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
```

```
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 */

#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"

#include <string.h>

#define DEBUG 1
#if DEBUG
#include <stdio.h>
#define PRINTF(...) printf(__VA_ARGS__)
#define PRINT6ADDR(addr) PRINTF(" %02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x ", ((u8_t *)addr)[0], ((u8_t *)addr)[1], ((u8_t *)addr)[2], ((u8_t *)addr)[3], ((u8_t *)addr)[4], ((u8_t *)addr)[5], ((u8_t *)addr)[6], ((u8_t *)addr)[7], ((u8_t *)addr)[8], ((u8_t *)addr)[9], ((u8_t *)addr)[10], ((u8_t *)addr)[11], ((u8_t *)addr)[12], ((u8_t *)addr)[13], ((u8_t *)addr)[14], ((u8_t *)addr)[15])
#define PRINTLLADDR(lladdr) PRINTF(" %02x:%02x:%02x:%02x:%02x:%02x ",(lladdr)->addr[0], (lladdr)->addr[1], (lladdr)->addr[2], (lladdr)->addr[3],(lladdr)->addr[4], (lladdr)->addr[5])
#else
#define PRINTF(...)
#define PRINT6ADDR(addr)
#define PRINTLLADDR(addr)
#endif

#define SEND_INTERVAL        15 * CLOCK_SECOND
#define MAX_PAYLOAD_LEN        40

static struct uip_udp_conn *client_conn;
/*---------------------------------------------------------------------------*/
PROCESS(udp_client_process, "UDP client process");
AUTOSTART_PROCESSES(&udp_client_process);
/*---------------------------------------------------------------------------*/
static void
tcpip_handler(void)
{
  char *str;

  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("Response from the server: '%s'\n", str);
  }
}
/*---------------------------------------------------------------------------*/
static void
timeout_handler(void)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN];

  printf("Client sending to: ");
  PRINT6ADDR(&client_conn->ripaddr);
  sprintf(buf, "Hello %d from the client", ++seq_id);
  printf(" (msg: %s)\n", buf);
```

```
78:   uip_udp_packet_send(client_conn, buf, strlen(buf));
79: }
80: /*---------------------------------------------------------------------------*/
81: static void
82: print_local_addresses(void)
83: {
84:   int i;
85:   uip_netif_state state;
86:
87:   PRINTF("Client IPv6 addresses: ");
88:   for(i = 0; i < UIP_CONF_NETIF_MAX_ADDRESSES; i++) {
89:     state = uip_netif_physical_if.addresses[i].state;
90:     if(state == TENTATIVE || state == PREFERRED) {
91:       PRINT6ADDR(&uip_netif_physical_if.addresses[i].ipaddr);
92:       PRINTF("\n");
93:     }
94:   }
95: }
96: /*---------------------------------------------------------------------------*/
97: static void
98: set_connection_address(uip_ipaddr_t *ipaddr)
99: {
100:   // change this IP address depending on the node that runs the server!
101:   uip_ip6addr(ipaddr,0xfe80,0,0,0,0x8400,0x0012,0x91c7,0x1b01);
102: }
103: /*---------------------------------------------------------------------------*/
104: PROCESS_THREAD(udp_client_process, ev, data)
105: {
106:   static struct etimer et;
107:   uip_ipaddr_t ipaddr;
108:
109:   PROCESS_BEGIN();
110:   PRINTF("UDP client process started\n");
111:
112:   // wait 3 second, in order to have the IP addresses well configured
113:   etimer_set(&et, CLOCK_CONF_SECOND*3);
114:
115:   // wait until the timer has expired
116:   PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
117:
118:   print_local_addresses();
119:   set_connection_address(&ipaddr);
120:
121:   /* new connection with remote host */
122:   client_conn = udp_new(&ipaddr, HTONS(3000), NULL);
123:
124:   PRINTF("Created a connection with the server ");
125:   PRINT6ADDR(&client_conn->ripaddr);
126:   PRINTF("local/remote port %u/%u\n",
127:     HTONS(client_conn->lport), HTONS(client_conn->rport));
128:
129:   etimer_set(&et, SEND_INTERVAL);
130:   while(1) {
131:     PROCESS_YIELD();
132:     if(etimer_expired(&et)) {
133:       timeout_handler();
134:       etimer_restart(&et);
135:     } else if(ev == tcpip_event) {
136:       tcpip_handler();
137:     }
138:   }
139:
140:   PROCESS_END();
141: }
```

```
142: /*---------------------------------------------------------------------------*/
```

udp-client.c

The IPv6 local-link addresses of the nodes are determined automatically using Stateless Address Autoconfiguration. Each node appends its link-layer 6byte address to the local-link network prefix (fe80::0/64).

When the `tcpip_process` is started, the address configuration is done, then some IPv6 ND messages are exchanged (such as Duplicate Address Detection and Router Sollicitation). If a router is present, it may give some extra addresses to the node.The stack is now ready to operate at least on a local link scope.
The procedure to send and receive UDP datagrams is now similar to when IPv4 was used. The UDP connection must be filled with the other node's IP address and the destination port used, and the local port number may be specified. Then the functions called and events are the same.

To try the communication between 2 nodes, make sure that you update the `udp-client.c` with the server IP address.
If you want to interact with your node UDP server from a PC, you can use the `tools/uip6-bridge` application on a node linked to the PC with a serial link. The PC should run tapslip6 and be configure to act as an IPv6 router. The procedure is described in the `tools/uip6-bridge/README` file.

An Introduction to Cooja

This page contains information about the COOJA network simulator for Contiki. To get most out of the information contained here, the reader should have basic Contiki knowledge: how to use the Contiki build system and how to create simple Contiki processes. COOJA is included in Contiki since version 2.0. In the Contiki source tree, COOJA resides in

- `tools/cooja` Simulator code
- `platform/cooja` Native platform with glue drivers for Cooja

The COOJA Simulator

The section gives a high-level overview of sensor network simulations in COOJA. This information is not strictly needed for using COOJA, but may help users to more easily understand problems, limitations, and features in the simulator.

Contiki level: the Contiki Mote Type

A simulated Contiki Mote in COOJA is an actual compiled and executing Contiki system. The system is controlled and analyzed by COOJA. This is performed by compiling Contiki for the native platform as a shared library, and loading the library into Java using Java Native Interfaces (JNI). Several different Contiki libraries can be compiled and loaded in the same COOJA simulation, representing different kinds of sensor nodes (heterogeneous networks). COOJA controls and analyzes a Contiki system via a few functions. For instance, the simulator informs the Contiki system to handle an event, or fetches the entire Contiki system memory for analysis. This approach gives the simulator full control of simulated systems. Unfortunately, using JNI also has some annoying side-effects. The most significant is the dependency on external tools such as compilers and linkers and their run-time arguments. COOJA was originally developed for Cygwin/Windows and Linux platform, but has later been ported to MacOS.

Getting started

Java version 1.6 or later is required to run COOJA. We recommend using the latest version from Sun. In addition, the build tool ant is also required for building COOJA. For Windows users, we recommend using Cygwin. Add the Cygwin binaries path (for example c:\cygwin\bin) to your PATH environmental variable. To compile and start COOJA:

```
cd tools/cooja

ant run
```

COOJA builds and the simulator is started. However, before you can simulate a Contiki system, you need to configure COOJA to correctly interface your toolchain.

- Open menu *Settings* > *External tools paths*

This dialog displays your current COOJA configuration: compiler paths and arguments, a bunch of regular expressions used to parse information from these tools, etc. **These settings are stored in your home directory: .cooja.user.properties**, such as in *home/myuser* or *C:\Documents And Settings\myuser*.

Configuration Wizard

(This wizard replaces the old JNI tests in /tools/cooja/examples/jni_tests.) The configuration wizard helps configuring COOJA for using JNI, a challenging task on many systems. The wizard consists of 4 steps and tests compiling, linking, loading libraries, analyzing library memory, and controlling library execution.

- Open menu *Settings* > *Cooja mote configuration wizard*

Complete all 4 steps. Note that you may have to change the recommended settings. Any changes can later be reviewed in the *External tools paths* dialog. If errors related to jni.h not being found, review the $JAVA_HOME path points to the right files.

Create a Hello World simulation

After completing the configuration wizard, we are now ready to create a simulation.

- Open menu *File* > *New simulation*, and click *Create*

A simulation without motes and using the default parameters is created. Before adding motes to the simulation, we first need to create a mote type. The mote type determines the type of sensor hardware and which Contiki applications are to be simulated.

- Open menu *Motes* > *Add Motes* > *Create new mote type* > *Cooja Mote Type*

A dialog allowing you to configure the new mote type appears.

- Enter a suitable description: "My first hello world mote type"
- Click Browse, and select the Contiki Hello World application: **hello-world.c**
- Compile the Contiki shared library by clicking **Compile**
- When the compilation finishes, load the library and create the mote type by clicking **Create**.

We have now added a mote type, however, the simulation does not yet contain any simulated motes. In the Add Motes dialog:

- Enter **10** and click **Add motes**

You may later add further motes of this type by menu *Motes* > *Add motes* > *My first hello world mote type*. A few plugins are started: a control panel for starting and pausing the simulation, a visualizer that shows the node positions, and a log listener showing printouts of all simulated nodes.

- Press **Start** (or CTRL+S) in the **Control Panel** plugin to start simulating.

Save and load simulations

Cooja allows for saving and loading simulation configurations. A simulation configuration contains the simulated modes and mote types, radio medium configuration, active plugins, etc.

Note that only the configuration, not the state of the simulation is saved. Hence, when a simulation is loaded, the simulation will start over from time 0.

To save the current simulation:

- Open menu *File* > *Save simulation as*
- Enter suitable name. Configuration files are stored with the file extension **.csc**

To load a simulation:

- Open menu *File* > *Open simulation* > *Browse...*
- Select configuration file to load.

The simulation is loaded, and the plugins will appear after a while. Loading a simulation with several mote types may take some time, since Contiki is recompiled in the background.
A similar functionality as saving and loading simulation, is **reloading** a simulation. To reload the current simulation:

- Open menu *Simulation* > *Reload simulation* > *Reload with same random seed*

COOJA configs for Contiki development: sharing simulation configs

When developing Contiki applications, you should normally keep all your code in a project directory. Your project directory may be a subfolder of Contiki (e.g. *examples/myproject*), or external to Contiki (e.g. */home/user/mycontikiproject*).

COOJA leverages the Contiki project directories by storing any simulation's external file references (such as to Contiki applications) relative to the directory where you save the simulation configuration. If you save a COOJA configuration in your project directory, any references to Contiki code residing in that directory will hence be stored as portable relative paths. This feature enables moving and sharing project directories, for example committing Contiki projects to version control.

Example: Contiki path: */home/user/contiki-2.x* Contiki project path: */home/user/mycontikiproject*Contiki application: */home/user/mycontikiproject/myapp.c*

- Create a COOJA simulation, compile *myapp.c* and finally save the simulation configuration **in the project directory**.

COOJA config: */home/user/mycontikiproject/mysimconfig.csc*
The Contiki application path will be stored as "[CONFIG]/myapp.c" i.e. relative to where the configuration file is stored.

COOJA configs for Contiki development: simulation quickstart

It is also possible to quickstart COOJA, instead of starting COOJA from the *tools/cooja/* directory.

To quickstart COOJA from a Contiki project directory, */home/user/mycontikiproject/*:

```
make myapp.cooja TARGET=cooja
```

or

```
make mysimconfig.csc TARGET=cooja
```

Commandline options

- **-quickstart=Simulation** Load Simulation at startup. This must be the first parameter.
- **-nogui=Simulation** Run simulation without gui. This must be the first parameter.
- **-applet** Run as applet. Must be the first parameter.
- **-log4j=File** Load Log4j-Config from file.
- **-contiki=Path** Set Contiki path
- **-external_tools_config=File** Set Cooja config file.

Using Cooja Test Scripts to Automate Simulations

Overview

(The following tutorial is based on an email from Fredrik Österlind to the developer mailing list, 2009-05-18)

The easiest way to automate simulations in Cooja so that you can run multiple test is by using COOJAs **Contiki test scripts**. Contiki tests can be run both with and without GUI, and could, combined with a shell script, automate several test runs (for example changing parameters in the .csc simulation file).

Example: running a simple Hello world test repeatedly with different random seeds.

Create new simulation

When in the "Create new simulation" dialog, be sure to configure for automatically generated random seeds (the checkbox right of the "Main random seed"). Every time this simulation is loaded, a new random seed will be used.

Configure simulation

Create a mote type (use hello world or your favorite Contiki app), and add a few nodes. At this step you may typically also want to configure the radio medium, node positions...

Create a test script

Start the "Contiki Test Editor" plugin. There are several test scripts available (Javascripts, ends with .js), for now use this very simple script:

```
TIMEOUT(10000);
log.log("first simulation message at time : " + time + "\n");
while (true) {
  YIELD(); /* wait for another mote output */
}
```

Copy and paste this script into the upper part of the test editor - this is where the javascript code goes. The lower part shows the currently active test output.

The above script is very simple: it declares a test timeout of 10 seconds. It also prints the first printf() output from any of the simulation nodes. Depending on what you want to do, perhaps

such a simple script is sufficient if you only want to repeat tests with a given duration (10 seconds in this case).

Activate the test

Click "Activate" in the plugin, and then start the simulation. The simulation stops after ten seconds. In the lower part of the test editor plugin, you should messages indicating that the test timed out (failed).

Run test (repeatedly) without GUI

(This section is outdated and doesn't work, there is no "Export as Contiki test" option, see mailing list discussion at http://sourceforge.net/p/contiki/mailman/contiki-developers/thread/51961D79.5080003@informatik.uni-erlangen.de/)

To run this test without GUI, you should first export it: "Export as Contiki test". Follow the instructions. The test files will be saved to /tools/cooja/contiki_tests. Open a terminal and enter this directory.

To run the test once:

```
bash RUN_TEST mytest
```

mytest.log contains the test output (the lower part of the plugin when in graphical mode).

To run the test 10 times:

```
bash RUN_REPEATED 10 mytest
```

mytest6.log contains the test output of test run number 6/10.

Background

Test scripts are written in JavaScript using Rhino The following objects are exported from the Cooja Java environment to the JavaScript environment. Using these you should be able to access any part of the Cooja.

- `se.sics.cooja.Mote mote` The Mote Mote.java
- `int id` The id of the mote
- `long time` The current time
- `String msg` The message
- `se.sics.cooja.Simulation sim` The simulation Simulation.java
- `se.sics.cooja.Gui gui` The GUI Gui.java
- `interface ScriptLog log`

    - `log(String log)`
    - `public void testOK()`
    - `public void testFailed()`
    - `public void generateMessage(long delay, String msg)` This generates a message after delay. This can be used to wait without having a mote to output anything.
- `boolean TIMEOUT` Set to true when the script times out
- `boolean SHUTDOWN` Set to true when the script is deactivated
- `java.util.concurrent.Semaphore.Semaphore SEMAPHORE_SCRIPT`
- `java.util.concurrent.Semaphore.Semaphore SEMAPHORE_SIM`

Keywords

These may not be split into multiple lines!

- `write(Mote mote, String msg)` Write to the mote's serial interface
- `TIMEOUT(long time[, action])`
    - `time` Time until timeout
    - `action` Script to execute when timing out (e.g. `log.log("last message: " + msg + "\n")`)
- `YIELD()` Wait for the next message
- `WAIT_UNTIL(expr)` Yield if `expr` is not true
- `YIELD_THEN_WAIT_UNTIL(expr)` This is equivalent to `YIELD(); WAIT_UNTIL(expr)`
- `GENERATE_MSG(long time, String msg)` Send a message in the future. This is handy if you want to sleep a while, but continue the execution afterwards

```
GENERATE_MSG(2000, "sleep"); //Wait for two sec
YIELD_THEN_WAIT_UNTIL(msg.equals("sleep"));
```

Documentation might be buggy or the commands might have unexpected side effects:

- `SCRIPT_KILL()` Terminate the script
- `SCRIPT_TIMEOUT()` Cause a timeout

Tips and Tricks

Get access to a certain mote

This allows to write to a mote without having to get a reference by waiting for the ''mote'' variable to point to the right mote.

```
write(sim.getMoteWithID(1), "some string");
```

Writing to Flash (Sky,??) c

The interfaces must be initialized by opening the interface viewer and selecting the flash. The window can be minimized, though.

```
file = "Myfile.bin";
fs = mote.getFilesystem();
success = fs.insertFile(file);
```

Get the current simulation Time

```
sim.getSimulationTime() // us
sim.getSimulationTimeMillis() // ms
```

Press a button

```
mote.getInterfaces().getButton().clickButton()
```

Set simulation to real time

```
sim.setSpeedLimit(1.0);
```

Send message to all motes

```
allm = sim.getMotes();
for(var i = 0; i < allm.length; i++){
    write(allm[i], "msg");
}
```

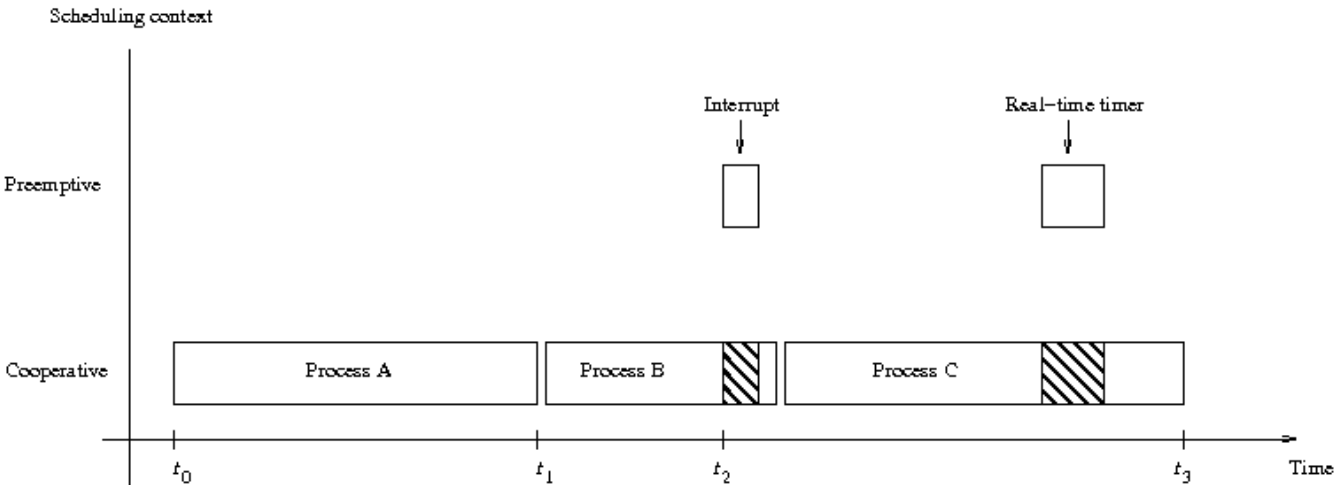
Write Output to separate Files

```
//import Java Package to JavaScript
importPackage(java.io);

// Use JavaScript object as an associative array
outputs = new Object();

while (true) {
   //Has the output file been created.
   if(! outputs[id.toString()]){
       // Open log_<id>.txt for writing.
       // BTW: FileWriter seems to be buffered.
       outputs[id.toString()]= new FileWriter("log_" + id +".txt");
   }
   //Write to file.
   outputs[id.toString()].write(time + " " + msg + "\n");

   try{
```

```
        //This is the tricky part. The Script is terminated using
        // an exception. This needs to be caught.
        YIELD();
    } catch (e) {
        //Close files.
        for (var ids in outputs){
            outputs[ids].close();
        }
        //Rethrow exception again, to end the script.
        throw('test script killed');
    }
}
```

Processes

Thanos Tsakiris edited this page 13 days ago · 10 revisions



Code in Contiki runs in either of two execution contexts: cooperative or preemptive. Cooperative code runs sequentially with respect to other cooperative code. Preemptive code temporarily stops the cooperative code. Contiki processes run in the cooperative context, whereas interrupts and real-time timers run in the preemptive context.

All Contiki programs are processes. A process is a piece of code that is executed regularly by the Contiki system. Processes in Contiki are typically started when the system boots, or when a module that contains a process is loaded into the system. Processes run when something happens, such as a timer firing or an external event occurring.

Code in Contiki can run in one of two execution contexts: cooperative or preemptive. Code running in the cooperative execution context is run sequentially with respect to other code in the cooperative context. Cooperative code must run to completion before other cooperatively scheduled code can run. Preemptive code may stop the cooperative code at any time. When preemptive code stops the cooperative code, the cooperative code will not be resumed until the preemptive code has completed. The concept of Contiki's two scheduling contexts is illustrated above.

Processes always run in the cooperative context. The preemptive context is used by interrupt handlers in device drivers and by real-time tasks that have been scheduled for a specific deadline. We return to the real-time tasks when we discuss timers.

Table of Contents

The Structure of a Process

A Contiki process consists of two parts: a process control block and a process thread. The process control block, which is stored in RAM, contains run-time information about the process such as the name of the process, the state of the process, and a pointer to the process thread of the process. The process thread is the code of the process and is stored in ROM.

The Process Control Block

The process control block contains information about each process, such as the state of the process, a pointer to the process' thread, and the textual name of the process. The process control block is only used internally by the kernel and is never directly accessed by processes.

The internal structure of the process control block. None of the fields in the process control block should be directly accessed by user code:

```
struct process {
  struct process *next;
  const char *name;
  int (* thread)(struct pt *,
                 process_event_t,
                 process_data_t);
  struct pt pt;
  unsigned char state, needspoll;
};
```

A process control block is lightweight in that it requires only a couple of bytes of memory. The structure of the process control block is shown above. None of the fields in this structure should be accessed directly. Only the process management functions should access them. The structure is shown here only to demonstrate how lightweight the process control block is. The first field in the process control block, `next`, points to the next process control block in the linked list of active processes. The `name` field points to the textual name of the process. The `thread` field, which is a function pointer, points to the process thread of the processes. The `pt` field holds the state of the protothread in the process thread. The `state` and `needspoll` fields are internal flags that keep the state of the process. The `needspoll` flag is set by the `process_poll()` function when the process is polled.
A process control block is not declared or defined directly, but through the `PROCESS()` macro. This macro takes two parameters: the variable name of the process control block, which is used when accessing the process, and a textual name of the process, which is used in debugging and when printing out lists of active processes to users. The definition of the process control block corresponding to the Hello World example is shown below.
An example process control block:

```
PROCESS(hello_world_process, "Hello world process");
```

The Process Thread

A process thread contains the code of the process. The process thread is a single protothread that is invoked from the process scheduler. An example process thread is shown below.

An example process thread:

```
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();

  printf("Hello, world\n");

  PROCESS_END();
}
```

Protothreads

A protothread is a way to structure code in a way that allows the system to run other activities when the code is waiting for something to happen. The concept of protothreads was developed within the context of Contiki, but the concept is not Contiki-specific and protothreads are used in many other systems as well.

Protothread provides a way for C functions to work in a way that is similar to threads, without the memory overhead of threads. Reducing the memory overhead is important on the memory-constrained systems on which Contiki runs.

A protothread is a regular C function. The function starts and ends with two special macros, `PT_BEGIN()` and `PT_END()`. Between these macros, a set of protothread functions can be used.
C preprocessor implementation of the main protothread operations:

```
struct pt { lc_t lc };
#define PT_WAITING 0
#define PT_EXITED  1
#define PT_ENDED   2
#define PT_INIT(pt)           LC_INIT(pt->lc)
#define PT_BEGIN(pt)          LC_RESUME(pt->lc)
#define PT_END(pt)            LC_END(pt->lc);    \
                              return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc);    \
                              if(!(c))           \
                                return PT_WAITING
#define PT_EXIT(pt)           return PT_EXITED
```

Local continuations implemented with the C switch statement:

```
typedef unsigned short lc_t;
#define LC_INIT(c)   c = 0
#define LC_RESUME(c) switch(c) { case 0:
#define LC_SET(c)    c = __LINE__; case __LINE__:
#define LC_END(c)    }
```

Protothreads in Processes

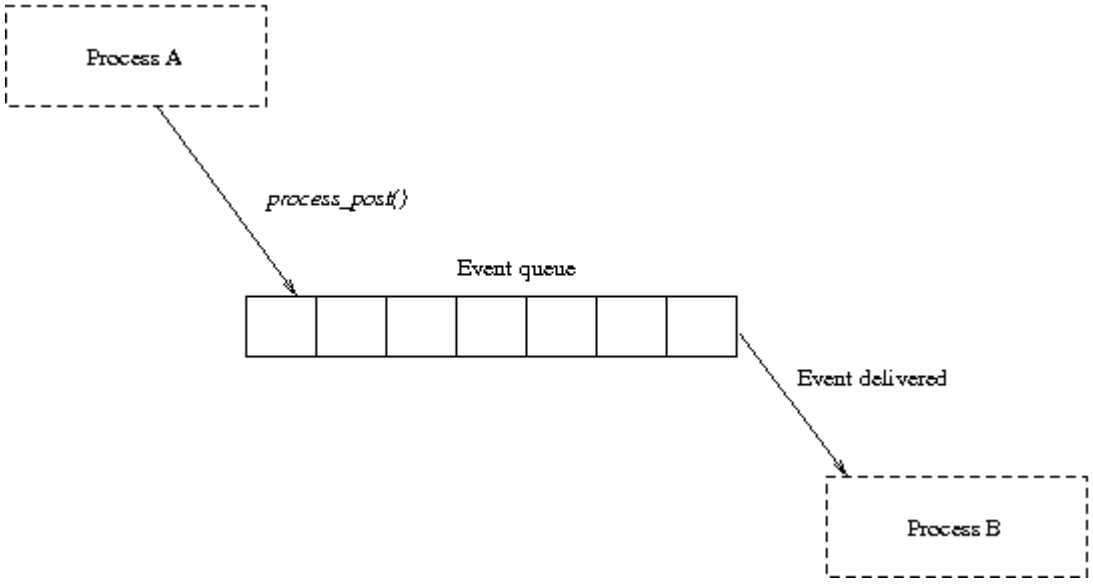
Contiki process implement their own version of protothreads, that allow processes to wait for incoming events. The protothread statements used in Contiki processes are therefore slightly different than the pure protothread statements as presented in the previous section.

Process-specific protothread macros that are used in Contiki processes:

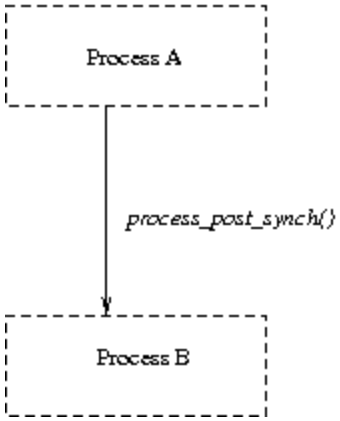
```
PROCESS_BEGIN(); // Declares the beginning of a process' protothread.
PROCESS_END(); // Declares the end of a process' protothread.
PROCESS_EXIT(); // Exit the process.
PROCESS_WAIT_EVENT(); // Wait for any event.
PROCESS_WAIT_EVENT_UNTIL(); // Wait for an event, but with a condition.
PROCESS_YIELD(); // Wait for any event, equivalent to PROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL(); // Wait for a given condition; may not yield the process.
PROCESS_PAUSE(); // Temporarily yield the process.
```

Events

In Contiki, a process is run when it receives an event. There are two types of events: asynchronous events and synchronous events.



When an asynchronous events is posted, the event is put on the kernel's event queue and delivered to the receiving process at some later time.



When a synchronous event is posted, the event is immediately delivered to the receiving process.

Asynchronous Events

Asynchronous events are delivered to the receiving process some time after they have been posted. Between their posting and their delivery, the asynchronous events are held on an event queue inside the Contiki kernel.

The events on the event queue are delivered to the receiving process by the kernel. The kernel loops through the event queue and delivers the events to the processes on the queue by invoking the processes.

The receiver of an asynchronous event can either be a specific process, or all running processes. When the receiver is a specific process, the kernel invokes this process to deliver the event. When the receiver of an event is set to be all processes in the system, the kernel sequentially delivers the same event to all processes, one after another.

Asynchronous events are posted with the `process_post()` function. The internals of the `process_post()` function is simple. It first checks the size of the current event queue to determine if there is room for the event on the queue. If not, the function returns an error. If there is room for the event on the queue, the function inserts the event at the end of the event queue and returns.

Synchronous Events

Unlike asynchronous events, synchronous events are delivered directly when they are posted. Synchronous events can only be posted to a specific processes.

Because synchronous events are delivered immediately, posting a synchronous event is functionally equivalent to a function call: the process to which the event is delivered is directly invoked, and the process that posted the event is blocked until the receiving process has finished processing the event. The receiving process is, however, not informed whether the event was posted synchronously or asynchronously.

Polling

A poll request is a special type of event. A process is polled by calling the function `process_poll()`. Calling this function on a process causes the process to be scheduled as quickly as possible. The process is passed a special event that informs the process that it has been polled.
Polling is the way to make a process run from an interrupt. The `process_poll()` function is the only function in the process module that is safe to call from preemptive mode.

Event Identifiers

Events are identified by an event identifier. The event identifier is an 8-bit number that is passed to the process that receives an event. The event identifier allows the receiving process to perform different actions depending on what type of event that was received.

Event identifiers below 127 can be freely used within a user process, whereas event identifiers above 128 are intended to be used between different processes. Identifiers above 128 are managed by the kernel.

The first numbers over 128 are statically allocated by the kernel, to be used for a range of different purposes. The definition for these event identifiers are shown below.

Event identifiers reserved by the Contiki kernel:

```
#define PROCESS_EVENT_NONE            128
#define PROCESS_EVENT_INIT            129
#define PROCESS_EVENT_POLL            130
#define PROCESS_EVENT_EXIT            131
#define PROCESS_EVENT_CONTINUE        133
#define PROCESS_EVENT_MSG             134
#define PROCESS_EVENT_EXITED          135
#define PROCESS_EVENT_TIMER           136
```

These event identifiers are used as follows.

PROCESS_EVENT_NONE : This event identifier is not used.

PROCESS_EVENT_INIT : This event is sent to new processes when they are initiated.

PROCESS_EVENT_POLL : This event is sent to a process that is being polled.

PROCESS_EVENT_EXIT : This event is sent to a process that is being killed by the kernel. The process may choose to clean up any allocated resources, as the process will not be invoked again after receiving this event.

PROCESS_EVENT_CONTINUE : This event is sent by the kernel to a process that is waiting in a `PROCESS_YIELD()` statement.

PROCESS_EVENT_MSG : This event is sent to a process that has received a communication message. It is typically used by the IP stack to inform a process that a message has arrived, but can also be used between processes as a generic event indicating that a message has arrived.

PROCESS_EVENT_EXITED : This event is sent to all processes when another process is about to exit. A pointer to the process control block of the process that is existing is sent along the event. When receiving this event, the receiving processes may clean up state that was allocated by the process that is about to exit.

PROCESS_EVENT_TIMER : This event is sent to a process when an event timer (etimer) has expired.

In addition to the statically allocated event numbers, processes can allocate event identifiers above 128 to be used between processes. The allocated event identifiers are stored in a variable, which the receiving process can use to match the event identifier with.

The Process Scheduler

The purpose of the process scheduler is to invoke processes when it is their time to run. The process scheduler invokes process by calling the function that implements the process' thread.

All process invocation in Contiki is done in response to an event being posted to a process, or a poll has being requested for the process, the process scheduler passes the event identifier to the process that is being invoked. With the event identifier, an opaque pointer is passed. The pointer is provided by the caller and may be set to NULL to indicate that no data is to be passed with the event. When a poll is requested for a process, no data can be passed.

Starting Processes

Processes are started with the `process_start()`. The purpose of this function is to set up the process control structure, place the process on the kernel's list of active processes, and to call the initialization code in the process thread. After `process_start()` has been called, the process is started.

The `process_start()` function first does a sanity check if the process that is to be started already exists on the list of active processes. If so, the process has already been started and the`process_start()` function returns.

After making sure that the process is not already started, the process is put on the list of active processes and the process control block is set up. The state of the process is set to`PROCESS_STATE_RUNNING` and the process' protothread is initialized with `PT_INIT()`.

Finally, the process is sent a synchronous event, `PROCESS_EVENT_INIT`. An opaque pointer is provided together with the event. This pointer is passed from the process that invoked`process_start()` and can be used as a way to transport information to the process that is to be started. Normally, however, this pointer is set to NULL.

As the process receives its first event, the `PROCESS_EVENT_INIT` event, the process executes the first part of its process thread. Normally, this part of the process thread contains initialization code that is to be run once when the process starts.

Exiting and Killing Processes

Processes exit in either of two ways. Either the process itself exits, or it is killed by another process. Processes exit either by calling the `PROCESS_EXIT()` function or when the execution of the process thread reaches a `PROCESS_END()` statement. A process can be killed by another process by calling the `process_exit()` function.

When a process exits, regardless of it exits itself or if it is being killed by another process, the Contiki kernel send an event to all other processes to inform them of the process exiting. This can be used by other processes to free up any resource allocations made by the process that is exiting. For example, the uIP TCP/IP stack will close and remove any active network connections that the exiting process has. The event, `PROCESS_EVENT_EXITED`, is sent as a synchronous event to all active processes except for the one that is exiting.

If a process is killed by another process, the process that is killed is also sent a synchronous event, `PROCESS_EVENT_EXIT`. This event informs the process that it is about to be killed and the process can take the opportunity to free up any resource allocations it has made, or inform other processes that it is about to exit.

After the Contiki kernel has sent the events informing the processes about the process that is about to exit, the process is removed from the list of active processes.

Autostarting Processes

Contiki provides a mechanism whereby processes can be automatically started either when the system is booted, or when a module that contains the processes is loaded. This mechanism is implemented by the autostart module. The purpose of the autostart mechanism is to provide a way that the developer of a module can inform the system about what active processes the module contains. The list is also used when killing processes as a module is to be unloaded from memory.

Autostarted processes are kept on a list, that is used by the autostart module to start the processes. The processes are started in the order they appear on the list.

There are two ways in which processes are autostarted: during system boot-up or when a module is loaded. All processes that are to be started at system boot-up must be contained in a single, system-wide list. This list is supplied by the user, typically in one of the user modules that are compiled with the Contiki kernel. When the module is used as a loadable module, the same list is used to know what processes to start when the module is loaded.

When a module is loaded, the module loader finds the list of autostart processes and starts these processes after the module has been loaded into executable memory. When the module is to be unloaded, the module loader uses the same process list to kill the processes that were started as part of the module loading process.

In Contiki, the autostart mechanism is the most common way through which user processes are started.

An Example Process

To make the discussion about how processes work concrete, we now turn to an example of two processes. This example not only shows how the code that implements a process looks like, but also the different steps in the lifetime of a process. Below is a brief example of a "Hello, world"-style program, but this example is a little more elaborate.

An example process that receives events and prints out their number:

```
#include "contiki.h"

PROCESS(example_process, "Example process");
AUTOSTART_PROCESSES(&example_process);

PROCESS_THREAD(example_process, ev, data)
```

```
{
  PROCESS_BEGIN();

  while(1) {
    PROCESS_WAIT_EVENT();
    printf("Got event number %d\n", ev);
  }

  PROCESS_END();
}
```

The code shown above is a complete example of a Contiki process. The process is declared, defined, and is automatically started with the autostart feature. We go through the example line by line.

At line 1, we include the Contiki header files. The contiki.h file will include all the headers necessary for the basic Contiki functions to work. At line 3, we define the process control block. The process control definition defines both the variable name of the process control block, which in this case is `example_process`, and the textual, human-readable name of the process, which in this case is*Example process*.
After defining the process control block, we can use the variable name in other expressions. At line 4, the `AUTOSTART_PROCESSES()` declaration tells Contiki that the `example_process` is to be automatically started when Contiki boots, or, if this module is compiled as a loadable module, when the module is loaded. The autostart list consists of pointers to process control blocks, so we take the address of `example_process` by prefixing it with an ampersand.
At line 6, we begin the definition of the process' thread. This includes the variable name of the process, `example_process`, and the names of the event passing variables `ev` and `data`. The event passing variables are used when the process receives events, as we describe below.
At line 8, we begin the process with the `PROCESS_BEGIN()` declaration. This declaration marks start of code belonging to the process thread. Code that is placed above this declaration will be (re)executed each time the process is scheduled to run. Code placed below the declaration will be executed based on the actual process thread control flow. In most cases, you don't need to place any code above `PROCESS_BEGIN()`.
At line 10, we start the main loop of the process. As we have previously said, Contiki processes cannot start loops that never end, but in this case we are safe to do this because we wait for events below. When a Contiki process waits for events, it returns control to the Contiki kernel. The Contiki kernel will service other processes while this process is waiting.

The process waits for events at line 11. The `PROCESS_WAIT_EVENT()` statement will return control to the Contiki kernel, and wait for the kernel to deliver an event to this process. When the Contiki kernel delivers an event to the process, the code that follows the `PROCESS_WAIT_EVENT()` is executed. After the process wakes up, the `printf()` statement at line 12 is executed. This line simply prints out the event number of the event that the process received. The event number is contained in the `ev` variable. If a pointer was passed along with the event, this pointer is available in the `data` variable. In this example, however, we disregard any such pointer.

The `PROCESS_END()` statement at line 15 marks the end of the process. Every Contiki process must have a `PROCESS_BEGIN()` and a `PROCESS_END()` statement. When the execution reaches the`PROCESS_END()` statement, the process will end and will be removed from the kernel's active list. In this case, however, the `PROCESS_END()` statement will never be reached, because of the never-ending loop between lines 10 and 13. Instead, this process will continue to run either until the system is switched off, or until another process kills it via `process_exit()`.
A function that starts a process and sends it events:

```
static char msg[] = "Data";

static void
example_function(void)
{
  /* Start "Example process", and send it a NULL
     pointer. */

  process_start(&example_process, NULL);

  /* Send the PROCESS_EVENT_MSG event synchronously to
     "Example process", with a pointer to the message in the
     array 'msg'. */
  process_post_synch(&example_process,
                     PROCESS_EVENT_CONTINUE, msg);

  /* Send the PROCESS_EVENT_MSG event asynchronously to
     "Example process", with a pointer to the message in the
     array 'msg'. */
  process_post(&example_process,
               PROCESS_EVENT_CONTINUE, msg);

  /* Poll "Example process". */
  process_poll(&example_process);
}
```

Interaction between two processes is done with events. The example above shows an example of a function that starts a process and sends it a synchronous event, an asynchronous event, and a poll.

Conclusions

Processes are the primary way applications are run in Contiki. Processes consist of a process control block and a process thread. The process control block contains run-time information about the process and the process thread contains the code of the process. The process thread is implemented as a protothread, which is a lightweight thread specifically designed for memory-constrained systems.

In Contiki, code run in either of two execution contexts: cooperative, in which code never preempts other code, and preemptive, which preempts the execution of the cooperative code and returns control when the preemptive code is finished. Processes always run in cooperative

mode, where as interrupts run in preemptive mode. The only process control function that can be called from preemptive mode is `process_poll()`.
Processes communicate with each other by posting events to each other. Events are also posted when a process starts and exits.

Timers

The Contiki system provides a set of timer libraries that are used both by application programs and by the Contiki system itself. The timer libraries contain functionality for checking if a time period has passed, waking up the system from low power mode at scheduled times, and real-time task scheduling. The timers are also used by applications to let the system work with other things or enter low power mode for a time period before resuming execution.

Table of Contents

The Contiki Timer Modules

Contiki has one clock module and a set of timer modules: timer, stimer, ctimer, etimer, and rtimer. The different timer modules have different uses: some provide long-running timers with low granularity, some provide a high granularity but with a short range, some can be used in interrupt contexts (rtimer) others cannot.

The clock module provides functionality to handle the system time and also to block the CPU for short time periods. The timer libraries are implemented with the functionality of the clock module as base.

The timer and stimer libraries provides the simplest form of timers and are used to check if a time period has passed. The applications need to ask the timers if they have expired. The difference between these is the resolution of time: timers use system clock ticks while stimers

use seconds to allow much longer time periods. Unlike the other timers, the timer and stimer libraries can be safely used from interrupts which makes them especially useful in low level drivers.

The etimer library provides event timers and are used to schedule events to Contiki processes after a period of time. They are used in Contiki processes to wait for a time period while the rest of the system can work or enter low power mode.

The ctimer library provides callback timers and are used to schedule calls to callback functions after a period of time. Like event timers, they are used to wait for some time while the rest of the system can work or enter low power mode. Since the callback timers call a function when a timer expires, they are especially useful in any code that do not have an explicit Contiki process such as protocol implementations. The callback timers are, among other things, used throughout the Rime protocol stack to handle communication timeouts.

The rtimer library provides scheduling of real-time tasks. The rtimer library pre-empt any running Contiki process in order to let the real-time tasks execute at the scheduled time. The real-time tasks are used in time critical code such as the X-MAC implementation where the radio needs to be turned on or off at scheduled times without delay.

The Clock Module

The clock module provides functions for handling system time.

The API for the Contiki clock module is shown below. The function `clock_time()` returns the current system time in clock ticks. The number of clock ticks per second is platform dependent and is specified with the constant `CLOCK_SECOND`. The system time is specified as the platform dependent type `clock_time_t` and in most platforms this is a limited unsigned value which wraps around when getting too large. The clock module also provides a function `clock_seconds()` for getting the system time in seconds as an unsigned long and this time value can become much larger before it wraps around (136 years on MSP430 based platforms). The system time starts from zero when the Contiki system starts.
The clock module provides two functions for blocking the CPU: `clock_delay()`, which blocks the CPU for a specified delay, and `clock_wait()`, which blocks the CPU for a specified number of clock ticks. These functions are normally only used in low-level drivers where it sometimes is necessary to wait a short time without giving up the control over the CPU.
The function `clock_init()` is called by the system during the boot-up procedure to initialize the clock module.
The Clock Module API:

```
clock_time_t clock_time(); // Get the system time.
unsigned long clock_seconds(); // Get the system time in seconds.
void clock_delay(unsigned int delay); // Delay the CPU.
```

```
void clock_wait(int delay); // Delay the CPU for a number of clock ticks.
void clock_init(void); // Initialize the clock module.
CLOCK_SECOND; // The number of ticks per second.
```

Porting the Clock Module

The clock module is platform dependent and is implemented in the file `clock.c`. Since the clock module handles the system time, the clock module implementation usually also handles the notifications to the etimer library when it is time to check for expired event timers.

The Timer Library

The Contiki timer library provides functions for setting, resetting and restarting timers, and for checking if a timer has expired. An application must "manually" check if its timers have expired; this is not done automatically. The timer library use `clock_time()` in the clock module to get the current system time.
A timer is declared as a `struct timer` and all access to the timer is made by a pointer to the declared timer.
The API for the Contiki timer library is shown below. A timer is always initialized by a call to `timer_set()` which sets the timer to expire the specified delay from current time and also stores the time interval in the timer. `timer_reset()` can then be used to restart the timer from previous expire time and `timer_restart()` to restart the timer from current time.
Both `timer_reset()` and`timer_restart()` uses the time interval set in the timer by the call to `timer_set()`. The difference between these functions is that `timer_reset()` set the timer to expire at exactly the same time interval while `timer_restart()` set the timer to expire some time interval from current time, thus allowing time drift.
The function `timer_expired()` is used to determine if the timer has expired and `timer_remaining()` to get an estimate of the remaining time until the timer expires. The return value of the latter function is undefined if the timer already has expired.
The timer library can safely be used from interrupts. The code example below shows a simple example how a timer can be used to detect timeouts in an interrupt.

The Timer Library API:

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart the timer from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time until the timer expires.
```

An example showing how a timer can be used to detect timeouts:

```
#include "sys/timer.h"

static struct timer rxtimer;
```

```
void init(void) {
  timer_set(&rxtimer, CLOCK_SECOND / 2);
}

interrupt(UART1RX_VECTOR)
uart1_rx_interrupt(void)
{
  if(timer_expired(&rxtimer)) {
    /* Timeout */
    /* ... */
  }
  timer_restart(&rxtimer);
  /* ... */
}
```

The Stimer Library

The Contiki stimer library provides a timer mechanism similar to the timer library but uses time values in seconds, allowing much longer expiration times. The stimer library use `clock_seconds()`in the clock module to get the current system time in seconds.
The API for the Contiki stimer library is shown below and it is similar to the timer library. The difference is that times are specified as seconds instead of clock ticks.

The stimer library can safely be used from interrupts.

The stimer Library API:

```
void stimer_set(struct stimer *t, unsigned long interval); // Start the timer.
void stimer_reset(struct stimer *t); // Restart the stimer from the previous expiration time.
void stimer_restart(struct stimer *t); // Restart the stimer from current time.
int stimer_expired(struct stimer *t); // Check if the stimer has expired.
unsigned long stimer_remaining(struct stimer *t); // Get the time until the timer expires.
```

The Etimer Library

The Contiki etimer library provides a timer mechanism that generate timed events. An event timer will post the event `PROCESS_EVENT_TIMER` to the process that set the timer when the event timer expires. The etimer library use `clock_time()` in the clock module to get the current system time.
An event timer is declared as a `struct etimer` and all access to the event timer is made by a pointer to the declared event timer.
The API for the Contiki etimer library is shown below. Like the previous timers, an event timer is always initialized by a call to `etimer_set()` which sets the timer to expire the specified delay from current time. `etimer_reset()` can then be used to restart the timer from previous expire time and `etimer_restart()` to restart the timer from current time, both using the same time interval that was originally set by `etimer_set()`. The difference

between `etimer_reset()` and `etimer_restart()` is that the former schedules the timer from previous expiration time while the latter schedules the timer from current time thus allowing time drift. An event timer can be stopped by a call to `etimer_stop()` which means it will be immediately expired without posting a timer event. `etimer_expired()` is used to determine if the event timer has expired.
Note that the timer event is sent to the Contiki process used to schedule the event timer. If an event timer should be scheduled from a callback function or another Contiki process,`PROCESS_CONTEXT_BEGIN()` and `PROCESS_CONTEXT_END()` can be used to temporary change the process context. See Processes for more information about process management.
Below is a simple example how an etimer can be used to schedule a process to run once per second.

The etimer library cannot safely be used from interrupts.

The Etimer Library API:

```
void etimer_set(struct etimer *t, clock_time_t interval); // Start the timer.
void etimer_reset(struct etimer *t); // Restart the timer from the previous expiration time.
void etimer_restart(struct etimer *t); // Restart the timer from current time.
void etimer_stop(struct etimer *t); // Stop the timer.
int etimer_expired(struct etimer *t); // Check if the timer has expired.
int etimer_pending(); // Check if there are any non-expired event timers.
clock_time_t etimer_next_expiration_time(); // Get the next event timer expiration time.
void etimer_request_poll(); // Inform the etimer library that the system clock has changed.
```

Setup an event timer to let a process execute once per second.

```
#include "sys/etimer.h"

PROCESS_THREAD(example_process, ev, data)
{
  static struct etimer et;
  PROCESS_BEGIN();

  /* Delay 1 second */
  etimer_set(&et, CLOCK_SECOND);

  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    /* Reset the etimer to trig again in 1 second */
    etimer_reset(&et);
    /* ... */
  }
  PROCESS_END();
}
```

Porting the Etimer Library

The etimer library implementation in `core/sys/etimer.c` is platform independent but requires callback to `etimer_request_poll()` for handling the event timers. This allows the etimer library to

wakeup the system from low power mode when an event timer expires. The etimer library provides three functions for this:
`etimer_pending()` Check if there are any non-expired event timers.
`etimer_next_expiration_time()` Get the next event timer expiration time.
`etimer_request_poll()` Inform the etimer library that the system clock has changed and that an etimer might have expired. This function is safe to call from interrupts.
The implementation of the clock module usually also handles the callbacks to the etimer library since the module already handles the system time. This can be implemented simply by calling`etimer_request_poll()` periodically, or by taking advantage of `etimer_next_expiration_time()`and only notify the etimer library when needed.

The Ctimer Library

The Contiki ctimer library provides a timer mechanism that calls a specified function when a callback timer expires. The ctimer library use `clock_time()` in the clock module to get the current system time.
The API for the Contiki ctimer library is shown below and is similar to the etimer library. The difference is that `ctimer_set()` takes a callback function pointer and a data pointer as arguments. When a ctimer expires, it will call the callback function with the data pointer as argument. The code example below shows how a ctimer can be used to schedule a callback to a function once per second.
Note that although the callback timers are calling a specified callback function, the process context for the callback is set to the process used to schedule the ctimer. Do not assume any specific process context in the callback unless you are sure about how the callback timers are scheduled.

The ctimer library can not safely be used from interrupts.

The Ctimer Library API:

```
void ctimer_set(struct ctimer *c, clock_time_t t, void(*f)(void *), void *ptr); // Start the timer.
void ctimer_reset(struct ctimer *t); // Restart the timer from the previous expiration time.
void ctimer_restart(struct ctimer *t); // Restart the timer from current time.
void ctimer_stop(struct ctimer *t); // Stop the timer.
int ctimer_expired(struct ctimer *t); // Check if the timer has expired.
```

Setup a ctimer to call a function once per second:

```
#include "sys/ctimer.h"
static struct ctimer timer;

static void
callback(void *ptr)
{
  ctimer_reset(&timer);
  /* ... */
```

```
}

void
init(void)
{
  ctimer_set(&timer, CLOCK_SECOND, callback, NULL);
}
```

Porting the Ctimer Library

The ctimer library is implemented using the etimer library and no further porting is needed.

The Rtimer Library

The Contiki rtimer library provides scheduling and execution of real-time tasks (with predictable execution times). The rtimer library uses its own clock module for scheduling to allow higher clock resolution. The function `RTIMER_NOW()` is used to get the current system time in ticks and `RTIMER_SECOND` specifies the number of ticks per second.
Unlike the other timer libraries in Contiki, the real-time tasks pre-empt normal execution for the task to execute immediately. This sets some constraints for what can be done in real-time tasks because most functions do not handle pre-emption. Interrupt-safe functions such as `process_poll()` are always safe to use in real-time tasks but anything that might conflict with normal execution must be synchronized.
A real-time task can use the function `RTIMER_TIME(struct rtimer *t)` to retrieve the execution time required when the task was executed last time.
No examples here yet. The documentation that was here tried to explain the API from pre-2007, and was completely misleading.

Porting the Rtimer Library

The rtimer library implementation in `core/sys/rtimer.c` is platform independent and depends on the `rtimer-arch.c` to handle the platform dependent functionality such as scheduling. The following three functions need to be implemented when porting the rtimer library.
`rtimer_arch_init()` is called by the rtimer library to initialize the rtimer architecture code.
`rtimer_arch_now()` is used to get the current rtimer system time.
`rtimer_arch_schedule()` is used to schedule a call to `rtimer_run_next()` at a specific time. `rtimer_arch_schedule()` takes one time argument, `wakeup_time`, the requested time for the wakeup callback.
Besides these three functions, the rtimer architecture code needs to define `RTIMER_ARCH_SECOND` as the number of ticks per second and the data type `rtimer_clock_t` that is used for rtimer times. These are declared in the file `rtimer-arch.h`.
Platform dependent functions for the rtimer library:

```
RTIMER_CLOCK_LT(a, b); // This should give TRUE if 'a' is less than 'b', otherwise false.
RTIMER_ARCH_SECOND; // The number of ticks per second.
void rtimer_arch_init(void); // Initialize the rtimer architecture code.
rtimer_clock_t rtimer_arch_now(); // Get the current time.
int rtimer_arch_schedule(rtimer_clock_t wakeup_time); //  Schedule a call to
<tt>rtimer_run_next()</tt>.
```

Conclusions

Contiki contains a set of timer libraries that are used both by core Contiki modules and applications. The timer libraries are used to detect timeouts as well as schedule process events and function callbacks to allow the system to work with other things, or enter low power mode, for a time period before resuming execution.

Memory allocation

Contiki provides three ways to allocate and deallocate memory: the memb memory block allocator, the mmem managed memory allocator, and the standard C library malloc heap memory allocator. The memb memory block allocator is the most frequently used. The mmem managed memory allocator is used very infrequently and use of the malloc heap memory allocator is discouraged.

Table of Contents

The memb Memory Block Allocator

The memb library provides a set of memory block management functions. Memory blocks are allocated as an array of objects of constant size and are placed in static memory.

The memb API

```
MEMB(name, structure, num); // Declare a memory block.
void memb_init(struct memb *m); // Initialize a memory block.
void *memb_alloc(struct memb *m); // Allocate a memory block.
int memb_free(struct memb *m, void *ptr); // Free a memory block.
int memb_inmemb(struct memb *m, void *ptr); // Check if an address is in a memory block.
```

The `MEMB()` macro declares a memory block, which has the type `struct memb`. The definition of this data type is shown below. Since the block is put into static memory, it is typically placed at the top of a C source file that uses memory blocks. `name` identifies the memory block, and is later used as an argument to the other memory block functions. The `structure` parameter specifies the C type of the memory block, `num` represent the amount objects that the block accommodates. The expansion of the `MEMB()` macro yields three statements that define static memory. One statement stores the amount of object that the memory block can hold. Since the amount is stored in a variable of type `char`, the memory block can hold at most 127 objects. The second

statement allocates an array of `num` structures of the type referenced to by the `structure` parameter.

Once the memory block has been declared by using `MEMB()`, it has to be initialized by calling `memb_init()`. This function takes a parameter of `struct memb`, identifying the memory block. After initializing a `struct memb`, we are ready to start allocating objects from it by using `memb_alloc()`. All objects allocated through the same `struct memb` have the same size, which is determined by the size of the `structure` argument to `MEMB()`. `memb_alloc()` returns a pointer to the allocated object if the operation was successful, or NULL if the memory block has no free object.

`memb_free()` deallocates an object that has previously been allocated by using `memb_alloc()`. Two arguments are needed to free the object: `m` points to the memory block, whereas `ptr` points to the object within the memory block.

Any pointer can be checked to determine whether it is within the data area of a memory block. `memb_inmemb()` returns 1 if `ptr` is inside the memory block `m`, and 0 if it points to unknown memory.

The memb memory block structure.

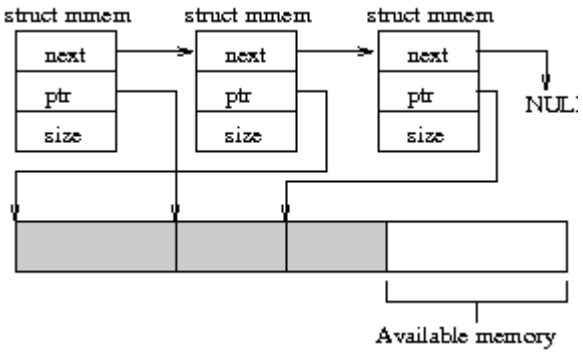
```
struct memb {
  unsigned short size;
  unsigned short num;
  char *count;
  void *mem;
};
```

The `open_connection()` function allocates a new `struct connection` variable for each new connection identified by `socket`. When a connection is closed, we free the memory block for the `struct connection` variable.

```
#include "contiki.h"
#include "lib/memb.h"

struct connection {
  int socket;
};
MEMB(connections, struct connection, 16);

struct connection *
open_connection(int socket)
{
  struct connection *conn;

  conn = memb_alloc(&connections);
  if(conn == NULL) {
    return NULL;
  }
  conn->socket = socket;
  return conn;
}

void
close_connection(struct connection *conn)
{
```

```
  memb_free(&connections, conn);
}
```

The mmem Managed Memory Allocator



The managed memory allocator (mmem) provides a dynamic memory allocation service similar to malloc. Its main distinction, however, is that it uses a level of indirection to enable automatic deframentation of the managed memory area.

It must be noted that the memory allocated with mmem_alloc() is 1-byte aligned. This is different from what malloc() does. The memory allocated with malloc() is suitably aligned for every data type and the returned void pointer can be safely casted to any other pointer type.

Instead, the pointer to the memory allocated by mmem_alloc() cannot be safely converted to any pointer type other than char*, signed char* or unsigned char*.

This means that if the allocated memory chunk is used to store the contents of a struct type, either the struct must be declared packed or memcpy() must be used. With GCC a packed struct can be specified using the following syntax:

```
struct __attribute__ ((__packed__)) my_packed_struct {
  ...
}
```

Many other compilers allow to specify a packed struct, often through an implementation-specific #pragma directive.

Every managed memory block is represented by an object of type `struct mmem`, as shown below. The mmem library organizes the `struct mmem` objects in a list named `mmemlist`. In the `struct mmem` object, `ptr` refers to the size of the allocated chunk in the contiguous memory pool reserved for the mmem library. `size` denotes the amount of bytes that the memory block can store.
The managed memory structure:

```
struct mmem {
  struct mmem *next;
  unsigned int size;
  void *ptr;
```

```
};
```

Programming Interface

The mmem programming interface is declared in `core/lib/mmem.h`. The available functions and macros are shown below.
`mmem_init()` initializes the managed memory library. MMEM_SIZE determines the size of the managed memory pool. The default size is 4~kB, but can be modified by defining MMEM_CONF_SIZE to another value in the configuration file used by the platform. Note that several platforms do not initialize the mmem library, which makes it necessary to include ensure that `mmem_init()` is called when using those platforms. The call to `mmem_init()` can be placed either in the module using mmem or in the platform startup code, which is typically found in`platforms/name/contiki-name-main.c()`.
`mmem_alloc()` allocates a block of memory. The first argument, `m`, is the reference to the `struct mmem` that will be initializes in `mmem_alloc()` and later used to refer to the memory block.
The `size`argument specifies how large the block should be. The function returns a non-zero value if the memory block was allocated, and zero if there was not enough memory available.
Allocated memory must be deallocated by using `mmem_free()` when it is no longer needed. `MMEM_PTR()` is a macro that returns a pointer to the memory block pointed to by `m`.
`mmem_free()` deallocates a managed memory block pointed to by the `m` argument. The mmem library will try to defragmentate the managed memory poll inside this function. All memory following the deallocated block is moved downwards to start at the newly freed position.
The `struct mmem` object of each moved block gets it `MMEM_PTR()` pointer updated to reflect the new position of the memory block.
The mmem API

```
MMEM_PTR(m); // Provide a pointer to managed memory.
int mmem_alloc(struct mmem *m, unsigned int size); // Allocated managed memory.
void mmem_free(struct mmem *); // Free managed memory.
void mmem_init(void); // Initialize the managed memory library.
```

`test_mmem()` allocates a managed memory object, copies a data structure into it, and retrieves the value of an element in the structure. The managed memory is deallocated when we have finished using it.

```
#include "contiki.h"
#include "lib/mmem.h"

static struct mmem mmem;

static void
test_mmem(void)
{
  struct my_struct {
    int a;
  } my_data, *my_data_ptr;

  if(mmem_alloc(&mmem, sizeof(my_data)) == 0) {
    printf("memory allocation failed\n");
```

```
  } else {
    printf("memory allocation succeeded\n");
    my_data.a = 0xaa;
    memcpy(MMEM_PTR(&mmem), &my_data, sizeof(my_data));
    /* The cast below is safe only if the struct is packed */
    my_data_ptr = (struct my_struct *)MMEM_PTR(&mmem);
    printf("Value a equals 0x%x\n", my_data_ptr->a);
    mmem_free(&mmem);
  }
}
```

The example above shows a basic example of how the managed memory library can be used. On line 4, we allocate a variable, `mmem`, that identifies the managed memory object that we are about to allocate. On line 13, we use the `mmem` variable as an argument for `mmem_alloc()` to allocate space for a structure of `sizeof(my_data)` bytes. If the allocation succeeded, we copy the values from an existing structure into the allocated structure, pointed to by `MMEM_PTR(&mmem)`. Individual members of allocated structure can then be accessed by a type cast of `MMEM_PTR(&mmem)` to `struct my_struct *`, as shown on line 20. Note that the cast is only safe if the struct is packed. The managed memory is finally deallocated on line 21 by calling `mmem_free()`.

Thread Safety

The mmem library is unsafe to use in preemptible code if it is also used in interrupt handlers or in MT threads. A module that uses a memory block could find its memory block relocated if the preemptive code called free.

In the ordinary case, the mmem library is used only within Contiki processes. Although such processes are cooperatively scheduled, it is necessary to ensure that each pointer to an address within in the memory block is updated when the process resumes control after waiting for an event. Any pointer to the block should then be reassigned using `MMEM_PTR()`.

The malloc Heap Memory Allocator

The standard C library provides a set of functions for allocating and freeing memory in the heap memory space. Contiki platforms may specify a small area of their memory spaces for the heap.

Static allocations are typically preferable in memory-constrained systems because dynamic allocations may incur fragmentation. Allocation and deallocation patterns on objects of varying sizes may be problematic in some malloc implementations.

The Malloc API

```
void *malloc(size_t size); // Allocate uninitialized memory.
void *calloc(size_t number, size_t size); // Allocate zero-initialized memory.
void *realloc(void *ptr, size_t size); // Change the size of an allocated object.
void free(void *ptr); // Free memory.
```

All functions listed in this section are declared in the standard C header `stdlib.h`.
The `malloc()`function allocates `size` bytes of memory on the heap. If the memory was successfully allocated, `malloc()` returns a pointer to it. If there was not enough contiguous free memory, `malloc()`returns NULL. Interested readers should get a book on the C programming language for a thorough description of the C malloc API.
`calloc()` works like malloc, but also ensures that allocated memory is zero-initialized. For unknown reasons, `calloc()` takes two arguments instead of one. The argument `number` denotes the amount of objects of a particular, specified by the `size` argument, that `calloc()` should allocate. Hence, the product of these two arguments is the size of the block of memory requested to be allocated.
The `realloc()` function reallocates a previously allocated block, `ptr`, with a new `size`. If the new block is smaller, `size` bytes of the data in the old block is copied into the new block. If the new block is larger, the complete old block is copied, and the rest of the new block contains unspecified data. Once the new block has been allocated, and its contents has been filled in, the old block is deallocated. `realloc()` returns NULL if the block could not be allocated. If the reallocation succeeded, `realloc()` returns a pointer to the new block.
`free()` deallocates a block that was previously allocated through `malloc()`, `calloc()`, or `realloc()`. The argument `ptr` must point to the start of an allocated block.

Conclusions

The memb library is a block allocator that use a statically declared memory area to store objects of a fixed size. The mmem library enables dynamic allocations with automatic defragmentation by using pointer indirection. Contiki also supports the malloc family of functions available in the standard C library. Although static memory is the most common type in Contiki, the methods presented here give programmers good options to dynamically adjusting the size the memory in use based on changing requirements during runtime

Input and output

The type of I/O devices available to the programmer is highly dependent on the platform in use. Contiki provides two basic interfaces that most platforms share: serial I/O and LEDs. Serial output is supported by the standard C library API for printing; whereas, serial input depends on a Contiki-specific mechanism. The LEDs API, on the other hand, is an abstraction for using LEDs portably. Platforms may implement that API as a stub if no LEDs are available.

Table of Contents

Serial Communication

The serial I/O API

```
int printf(const char *format, ...); /* Print a string according to a format. */
int putchar(int c); /* Print a single character. */
```

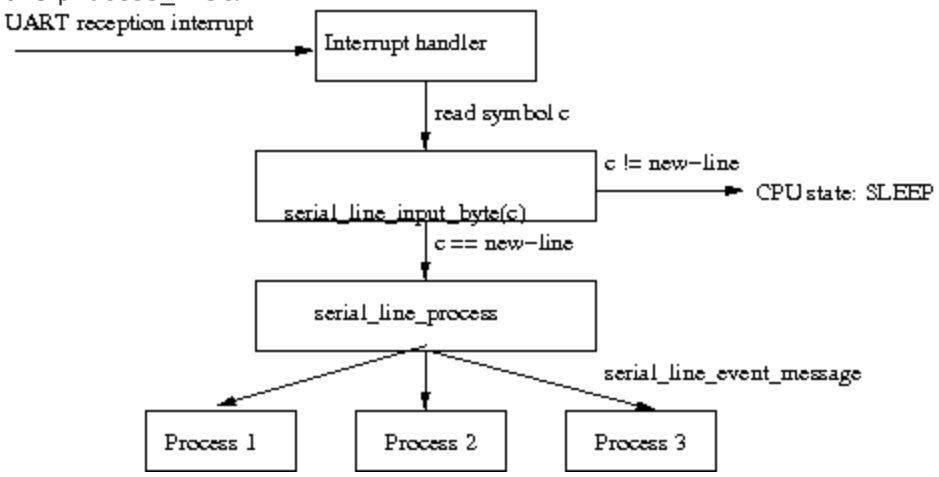
Printing

`printf()` is supported typically by linking in the function from the standard C library provided by the used compiler suite. We refer the reader to the ISO C standard for a complete description of how that function can be used.
Note that some embedded C libraries implement printf without floating point support. `printf()`simply calls `putchar()` after formatting an output string. `putchar()` has a hardware-dependent implementation that directs one byte at-a-time to the serial port.

Receiving

Contiki has a generic interface for line-based serial communication, specified in `core/dev/serial-line.h`. `serial_line_init()` needs to be called before using this unit, so that the `serial_line_process` and ring buffer required can be initialized. **Note**: `process_init()` must be

called before calling `serial_line_init()` so that the `serial_line_process` is added to the `process_list`.



An interrupt is generated when a character is ready to be read from the serial port. The interrupt handler calls `serial_line_input_byte()`, which buffers data until a new-line symbol is received. If we have a complete line, we broadcast an event to all processes.

`serial_line_input_byte()` is a callback function that receives input bytes from the serial drivers in several Contiki platforms. The function fills a local buffer until a line break is received; after which, it polls the `serial_line_process`. The local buffer size for serial lines is configurable through the parameter SERIAL_LINE_CONF_BUFSIZE; and, defaults to 128 bytes. If longer lines are received, only (SERIAL_LINE_CONF_BUFSIZE - 1) bytes and a nil byte will be passed on to `serial_line_process`. The `serial_line_process` broadcasts a `serial_line_event_message` to all processes in the system, along with the received string (pointed to by the event data).

The serial line driver has two macros defined, to control the input. `IGNORE_CHAR(c)` provides a simple filter that returns true if character `c` should not be put into the local serial buffer. It is defined currently to filter out carriage return (ASCII code 0x0D) symbols. `END` specifies the symbol that should cause the `serial_line_process` to be polled; and, currently is set to the line break (ASCII code 0x0A) symbol.

In platforms that have processors capable of going into low-power mode,`serial_line_input_byte()` can indicate whether the CPU should stay awake for further processing, or not. Since the function typically is called from an interrupt handler in a serial driver, it might have been called from low-power mode. Thus, if further processing is necessary, the hardware-dependent part of Contiki must ensure that the CPU moves into active mode, in order to be able to schedule the execution for other processes that might be interested in a serial line. For that purpose, all single-byte serial input functions in Contiki, including `serial_line_input_byte()`, must indicate whether the system should continue sleeping or not. A non-zero return value specifies that the CPU should move into active mode; whereas, a zero return value tells the driver that the system can continue sleeping in low-power mode.

We show how to receive serial lines below.

Serial line events are broadcast to all processes when the underlying device driver has received a line break. The event data contains a string with the line of data.

```
#include "contiki.h"
#include "dev/serial-line.h"
#include <stdio.h>

PROCESS(test_serial, "Serial line test process");
AUTOSTART_PROCESSES(&test_serial);

PROCESS_THREAD(test_serial, ev, data)
{
  PROCESS_BEGIN();

  for(;;) {
    PROCESS_YIELD();
    if(ev == serial_line_event_message) {
      printf("received line: %s\n", (char *)data);
    }
  }
  PROCESS_END();
}
```

LEDs

LEDs are a simple but important tool to communicate with users, or to debug programs. The LEDs API is shown below. The platform start-up code initializes the LEDs library by calling `leds_init()`. Any of the following functions can be used thereafter by either the system or applications.

The LEDs API

```
void leds_init(void); /* Initialize the LEDs driver. */
unsigned char leds_get(void); /* Get the status of a LED. */
void leds_on(unsigned char ledv); /* Turn on a set of LEDs. */
void leds_off(unsigned char ledv); /* Turn off a set of LEDs. */
void leds_toggle(unsigned char ledv); /* Toggle a set of LEDs. */
void leds_invert(unsigned char ledv); /* Toggle a set of LEDs. */
```

The set of LEDs that are on can be retrieved by calling `leds_get()`. That function returns a LEDs vector of type `unsigned char`, in which one bit represents one LED. The mapping between LEDs and the bits in a LED vector is platform-independent because of the set of macros specified below. If a platform has a superset of the defined colors, the LEDs vector must include a portion which is platform-defined.

LED identifiers for use in LED vectors.

```
#define LEDS_GREEN                  1
#define LEDS_YELLOW                 2
#define LEDS_RED                    4
#define LEDS_ALL                    7
```

`leds_on()` takes a LEDs vector argument, `ledv`, and switches on the LEDs that are set in the vector. The `leds_off()` function switches off the LEDs marked in the argument `ledv`. `leds_invert()` inverts

the current status of the LEDs that are set in the argument `ledv`. The function `leds_toggle()` is an alias for `leds_invert()`; and, is kept only for backward compatibility.

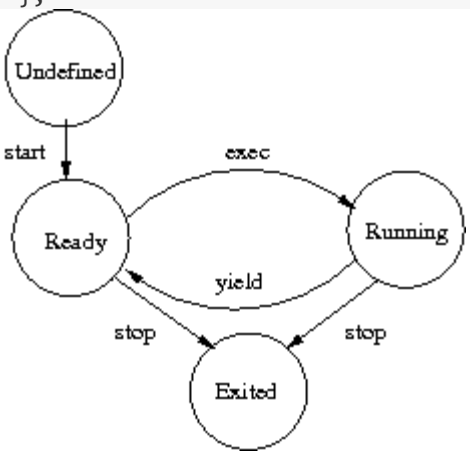
Conclusions

Contiki provides two generic programming interfaces for input and output. The serial communication uses standard C functions for output, and Contiki-specific functions for input. The LEDs library is entirely Contiki-specific; and, is supported on most platforms, either as an implementation using real hardware or as a set of stub functions.

Multithreading

Contiki supports preemptive threads through its mt library. mt threads have private stack segments and program counters that are saved when switching thread context. The library is divided into an architecture-independent part and an architecture-dependent part. Application programmers only need to regard the architecture-independent part of the multithreading library, which is declared in`sys/mt.h`.
Each thread is represented by an object of type `struct mt_thread`, which is defined as follows.

```
struct mt_thread {
  int state;
  process_event_t *evptr;
  process_data_t *dataptr;
  struct mtarch_thread thread;
};
```



The figure to the right shows the state chart of threads. A thread can have three different states throughout its lifetime. Once a thread is first started by applying `mt_start()` on it, the mt library sets its state to MT_STATE_READY. This state specifies that the thread is ready to execute, which is done by applying the `mt_exec()` function on the thread. The ready state can be reset once control is yielded by using `mt_yield()`. MT_STATE_RUNNING is the state of a thread that is currently being executed. MT_STATE_WAITING and MT_STATE_PEEK are currently unused. MT_STATE_EXITED is the final state of a thread, specifying that it can not be executed anymore.
The `thread` variable of type `struct mtarch_thread` is designated for architectural state, which typically includes the execution context of the thread. For example, in the msp430 port of libmt, the architectural context stores a private stack, and a stack pointer that delineates the top of the private stack.
The variables `evptr` and `dataptr` are currently unused by all implementations of the multithreading library.

Table of Contents

Programming Threads

The Multithreading API

void mt_init(void) : Initializes the library.

void mt_remove(void) : Uninstalls the library.

void mt_start(struct mt_thread *thread, void (* function)(void *), void *data) : Starts a thread.

void mt_exit(void) : Exits a thread.

void mt_exec(struct mt_thread *thread) : Execute as thread.

void mt_yield(void) : Release control voluntarily.

void mt_stop(struct mt_thread *thread) : Stops a thread.

The multithreading API is shown above. The `mt_init()` function initialized the thread library. If the thread library is no longer needed, one can call `mt_remove()` to uninstall the library after allowing it to deallocate internal state.

Starting and Stopping

Threads are started by calling the `mt_start()` function. The first argument, `thread`, is a pointer to allocated, but uninitialized space in which `mt_start()` will put internal state used for the threading functions. The second argument is a function pointer of type `void (*)(void *)`, which will be called when starting the execution of the thread. The third and final argument is a pointer to data that will be passed to the function when the thread is started. If no data is needed when calling the thread, the `data` argument can point to NULL, and the called function can just ignore the argument.
To stop a thread, a Contiki process can call `mt_stop()` with a pointer to the thread in question as the argument. This ensures that the thread is stopped and cleaned up properly. After this, the contents of the object of `struct mt_thread` are no longer used, and may be discarded by the calling process.

Scheduling

After starting a thread with `mt_start()`, the thread will continue executing until it has either been preempted by the system or yielded itself voluntarily. Preemption is controlled by the system, and therefore out of scope in this programming API. Yielding, on the other hand, is an action incurred by the threads themselves. By yielding, the thread releases control of the thread and allows the system to execute the main system thread. In order to do so, the thread calls `mt_yield()` without any argument.
When `mt_exec()` is called for the first time on a thread, it calls the function which was pointed to by the second argument of `mt_start()`. Subsequent calls to `mt_exec()` will restore the thread context and continue execution from the place where the thread was last yielded or preempted. The example below shows how to set up a process that in turn creates a thread.
Setup a thread from a Contiki process and execute it. We release control to the Contiki event scheduler by yielding from the thread, and then calling PROCESS_PAUSE in the process.

```
void
thread_entry(void *data)
{
  for(;;) {
    printf("Looping in thread_entry\n");
    mt_yield();
  }
}

PROCESS_THREAD(mt_process, event, data)
{
  static struct mt_thread thread;
  static int counter;

  PROCESS_BEGIN();
  mt_start(&thread, thread_entry, NULL);
  for(;;) {
    PROCESS_PAUSE();
    mt_exec(&thread);
    if(++counter == 10) {
      printf("Stopping the thread after %d calls\n",
             counter);
      mt_stop(&thread);
      break;
    }
  }
  PROCESS_END();
}
```

Architecture Support

The multithreading module requires processor-dependent code for switching context between threads. Each architecture-agnostic function in the multithreading API has a corresponding

architecture-dependent function. These implementations are located in the different subdirectories under `cpu/()` and named `mtarch.c`.
`mtarch_init()` is available to initialize the run-time support for multithreading, but this function is currently unused in most Contiki platforms. `mtarch_remove()` cleans up resources in an architecture-dependent way. `mtarch_start()`, `mtarch_yield()`, and `mtarch_exec()` implement the platform-dependent part of their corresponding mt library functions.
Contiki permits preemptive thread implementations in the functions `mtarch_pstart()` and `mtarch_pstop()`. Unlike the other functions in the underlying mt architecture API, these functions do not have corresponding platform-independent functions. The reason is that the place where it is suitable to place the preemption and the scheduling depends on the hardware architecture. If the platform supports only voluntary thread yielding, these functions may be left empty.

Conclusions

In this chapter we described an alternative method for concurrent programming in Contiki. Unlike protothreads, the mt library allows preemptive scheduling along with the possibility to yield from nested functions in a thread. mt threads cannot receive Contiki events, however, and are most suitable for freestanding computational units.

RPL modes

RPL has two ways to send data, either up the tree or down the tree. To send data up the tree, nodes need to keep track of one parent each. To send data down the tree, all nodes need to keep track of all nodes below them. Parent routes are built with DIO messages. Child routes are built with DAO messages. Upward routing has great scaling properties since the number of upward routes is constant with network size. Downward routing does not scale as well because the number of routes each node needs to have room for increases linearly with network size. Contiki supports both upward and downward routing.

By default, Contiki nodes are always downward-routable because they send DAO messages to announce their routes. And by default, Contiki nodes are always potential parents for upward routing because they send DIO messages to announce their routing availability. There is a way to, at compile time, turn on leaf mode, which makes nodes unavailable as parents in the mesh.

RPL modes was introduced with this pull request: https://github.com/contiki-os/contiki/pull/454

The patch adds a mechanism for setting RPL mode at run-time and adds a new mode in addition to the mesh and leaf modes: feather mode. A node in feather mode routes packets on behalf of other nodes but does not advertise their own route via DAO messages. Feather nodes can send data to other nodes inside or outside of the network and they can receive multicast messages and unicast messages from immediate neighbors, but they cannot be reached through the mesh.

Feather nodes are very lightweight as they require no routing table space in the network. A network can contain any number of feather nodes without using precious routing table space.

The name feather was chosen because of their light weight. Think of a network of feather nodes as a gathering of bird feathers, freely flying around in the breeze.

Radio duty cycling

Contiki has three duty cycling mechanisms: ContikiMAC, X-MAC and LPP. ContikiMAC is a protocol based on the principles behind low-power listening but with better power efficiency. Contiki's X-MAC is based on the original X-MAC protocol, but has been enhanced to reduce power consumption and maintain good network conditions. Contiki's LPP is based on the Low-Power Probing (LPP) protocol but with enhancements that improve power consumption as well as provide mechanisms for sending broadcast data.

Table of Contents

Background on Radio Duty Cycling Protocols

Power consumption is important for wireless sensor nodes to achieve a long network lifetime. To achieve this, low-power radio hardware is not enough. Existing low-power radio transceivers use too much power to provide long node lifetimes on batteries. For example, the CC2420 radio transceiver, used in the Tmote Sky boards, draws approximately 60 milliwatt of power when it is listening for radio traffic. Its power consumption when transmitting radio data is slightly higher. With a power draw of 60 milliwatt, a sensor node depletes its batteries in a matter of days.

To achieve a long lifetime, the radio transceiver must be switched off as much as possible. But when the radio is switched off, the node is not able to send or receive any messages. Thus the radio must managed in a way that allows nodes to receive messages but keep the radio turned off in between the reception and transmission of messages.

The purpose of a power-saving duty cycling protocol is to keep the radio turned off while providing enough rendezvous points for two nodes to be able to communicate with each other. There are multiple ways of providing rendezvous points in a duty cycling protocol. If the nodes are time-synchronized, the duty cycling protocol can set up and maintain a schedule of when nodes can communicate. The protocol would turn the radio on at a known time, and neighboring nodes could then transmit packets at this know time. Without scheduled time synchronization, nodes must use other means to provide rendezvous points.

In the sensor network research community, radio duty cycling protocols are often called MAC protocols because radio duty cycling was often implemented at the MAC layer. This is the reason for ContikiMAC's name. In Contiki, we have decided to factor out duty cycling from the MAC layer and moved it into its own layer, called the RDC layer.

Low-Power Listening

Low-power listening is a duty cycling technique where the receivers periodically turn on their radios to poll the radio medium for activity. If there is activity, the radio is kept on for a longer time in case a packet would be sent to the node.

Before sending a packet, the sender sends a number of strobe packets with the intent to let the receiver know that a neighbor wants to send a packet. Because all neighbors are periodically sampling the radio medium, they will see the strobe packets and keep their radios on in anticipation of the data packet. The strobe period is defined to be as long as the sleep period of the neighbors so that the neighbors will be woken up by the strobe packets.

There are several variants of low-power listening. Some do not send strobe packets to wake up their neighbors, but send a long wake-up tone instead. This tone serves the same purpose as the strobe packets but cannot contain any information.

By placing the receiver address in the strobe packets, the neighbors can quickly decide if they should be awake for the data packet. Only if the receiver address matches the address of the node will the node keep its radio on.

B-MAC is an early example of a low-power listening protocol. B-MAC, which was initially developed for radio transceivers that did not automatically packetize the bytes it sent, did not use strobe packets but a tone to wake up every neighbor.

WiseMAC is similar to B-MAC, but further optimizes transmission by allowing all nodes to record their neighbors' radio sample phase. When sending a packet to a given neighbor, the sender waits until the neighbor is known to sample the radio medium before it sends its wake-up tone. Thus the wake-up tone will be sent just before the receiver wakes up, further saving energy.

X-MAC was the first low-power listening protocol that was developed for a packetizing radio, and the first protocol to use strobes instead of a wake-up tone. In X-MAC, strobes contain the address of the receiver and other neighbors can refrain from keeping their radios on when hearing a strobe for another node. Additionally, X-MAC optimizes the mechanism further by adding a strobe acknowledgment packet. This packet is sent by a receiver in response to a strobe that contains its address. The sender will, upon hearing the strobe acknowledgment packet, immediately sent its data packet.

Low-Power Probing

Low-power probing is a rendezvous method that reverses the roles from low-power listening. Instead of having the receivers periodically sample the radio medium, the receivers periodically transmit a probe into the radio medium. To send a packet, the sender turns on its radio and listens for a probe packet from the neighbor that should receive the data packet. When the probe packet arrives, the sender immediately sends its data packet. The receiver, which keeps its radio on for a short while after sending its probe, will thus receive the data packet.

Two low-power probing protocols have been published. The first, simply called Low-Power Probing (LPP), used the mechanism to wake up a section of the network. The RI-MAC protocol developed a fully functional MAC protocol based on the same idea. Although the two protocols were published one after the other, they were developed independently of each other.

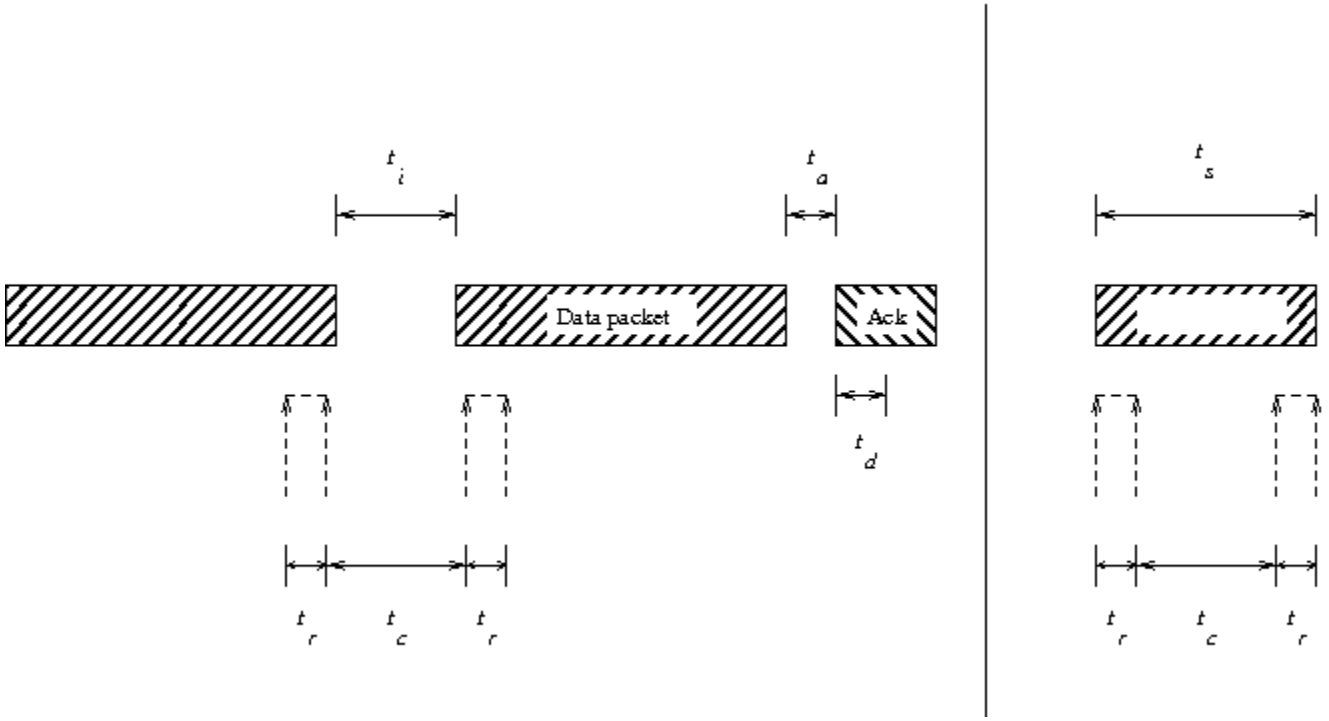
The Contiki Radio Duty Cycling Protocols

By default, Contiki provides three duty cycling MAC protocols: the ContikiMAC protocol, Contiki X-MAC and Contiki LPP. ContikiMAC is a low-power listening-based protocol with a very high power efficiency. The Contiki X-MAC is based on the original X-MAC protocol but with a significant set of improvements and extensions. The Contiki LPP is based on the same ideas as the original LPP and RI-MAC but with a number of changes.

In both the X-MAC and LPP protocols, nodes periodically wake up in anticipation of a potential incoming packet. The wake-up period is fixed, but the nodes are not synchronized. Therefore, there is a phase shift between all neighboring nodes. By utilizing information about a neighbor's phase, it is possible to save power when sending packets. Both the Contiki X-MAC and the Contiki LPP make use of this phase awareness.

Both the Contiki X-MAC and the Contiki LPP provides a streaming functionality where packets can be tagged with a stream flag. Packets tagged with the stream flag are part of a higher layer stream of packets, and the MAC layer treats them differently than others. For example, in a stream, it is likely that more packets arrive quickly. Thus the MAC protocol can keep its radio on for a while longer, knowing that more packets are about to arrive.

The ContikiMAC Radio Duty Cycling Mechanism



The ContikiMAC radio duty cycling mechanism is illustrated in the figure above. The figure illustrates the timing constraints under which ContikiMAC operates.

For ContikiMAC to work, the timing must satisfy the following constraints. First, ti, the interval between each packet transmission, must be smaller than t_c, the channel check interval. This is to ensure that either the first or the second channel check is able to see the packet transmission. If t_c would be smaller than t_i, two channel checks would not be able to reliably detect a packet train. Conversely, $t_r + t_c + t_r$ must be smaller than the transmission time of the shortest packet, t_s.

When a packet transmission has been detected, ContikiMAC keeps the radio on to be able to receive the full packet. When a full packet has been received, a link-layer acknowledgment is transmitted. ContikiMAC requires that the underlying link layer is able to transmit acknowledgment packets in response to received packets. If the link layer does not provide this by itself, it is possible to implement this as a layer between the link layer and ContikiMAC.

The time it takes for an acknowledgment packet to be transmitted, t_a, and the time it takes for an acknowledgment packet to be detected, t_d, establishes the lower bound for the check interval t_c.

We now have all the timing constraints for ContikiMAC. They can be summarized as

<math> t_a + t_d < t_i < t_c < t_c + 2 t_r < t_s. </math>

For an IEEE 802.15.4 link layer, we can substitute some of the variables in the equiation above with constants. First, t_a is defined by the IEEE 802.15.4 specification to be 12 symbols long. In 802.15.4, one symbol is 4/250 milliseconds long, meaning that t_a is $48/250 = 0.192$ milliseconds. Second, we can reliably detect the reception of the acknowledgment after the 4-byte long preamble and the 1-byte start of frame delimiter is transmitted, which takes 40/250 miliseconds. Finally, t_r is given by the CC2420 data sheet and is 0.192 milliseconds.

In conclusion, with the constants for IEEE 802.15.4 substituted, the equation above becomes

<math> 0.352 < t_i < t_c < t_c + 0.384 < t_s. </math>

The remaining variables, t_i, t_c, and t_s can now be chosen. Equation~\ref{eqn:part3:contikimac-802} gives a lower bound on $t_s > 0.736$ milliseconds, which sets a limit on the smallest packet size we can handle. With a bitrate of 250 kilobits per second, this means that packets must be at least 23 bytes long, including preamble, start of frame delimiter, and length field, which leaves 16 bytes of packet data. If ContikiMAC is asked to transmit a longer packet, the packet can be padded with null bytes to ensure this requirement.

The current ContikiMAC implementation uses the following configuration:

- $t_i = 0.4$ milliseconds,
- $t_c = 0.5$ milliseconds, and
- $t_s = 0.884$ milliseconds (28 - 6 data bytes).

Variable	Meaning	Value
t_i	the interval between each packet transmission	0.4 ms (Implementation dependant)
t_r	the time required for a stable Received Signal Strength Indication (RSSI)	0.192 ms (CC2420 dependant)
t_c	the interval between each channel check	0.5 ms (Implementation dependant)
t_a	the time between receiving a packet and sending the acknowledgment packet	0.192 ms

t_d	the time required for successfully detecting that an acknowledgment is being transmitted	0.16 ms
t_s	the time for transmitting the shortest data packet	0.884 ms (Implementation dependant)

The Contiki X-MAC

The Contiki X-MAC is based on the original X-MAC design but adds support for broadcast traffic, phase awareness, stream transport, and optional MAC layer retransmissions.

The basic mechanism in the Contiki X-MAC is shown in the figure to the right. Nodes maintain a strict schedule during which they first switch their radio on, listens for incoming strobes from neighbors, switch off their radio if no strobes were received, and keep the radio off during the rest of the duty cycle period. If a strobe was received, the node transmits a strobe acknowledgment packet to the sender. When the sender receives a strobe acknowledgment packet from the node to which it is sending strobe packets, the sender immediately sends it data packet.

Broadcast

To implement broadcast in an X-MAC-like protocol, there are at least two alternatives. The first implementation is to let the sender send strobe packets in which the address of the receiver is set to the broadcast address. When a node hears a strobe packet with the broadcast address, it does not transmit a strobe acknowledgment packet, but instead switches its radio on. The sender sends a full period of strobe packets, followed by the data packet. This mechanism ensures that all nodes are awake by the time the broadcast packet arrives, but it wastes energy because all nodes have to be awake until the data packet arrives.

The second way to implement broadcast in X-MAC is simpler. When sending a broadcast packet, instead of sending strobes, the sender simply sends the broadcast packet itself. This has the advantage that the neighbors do not need to keep their radio on for a longer time than a check interval. The problem with this is that it slightly changes the semantics of the broadcast: since nodes receive their broadcast packet at different times, the broadcast reception is not atomic and cannot be used for implicit synchronization.

Broadcast in the Contiki X-MAC.

Broadcast in the Contiki X-MAC is done with repeated transmissions of the data packet, as shown in the figure.

Phase Awareness

Phase awareness in the Contiki X-MAC is simple. Every time a node sends a packet to its neighbor, the sender records the time at which the receiver was awake. The sender knows the time at which the neighbor was awake, because the receiver sent a strobe acknowledgment packet then. When the sender is aware of the time at which the neighbor was awake, the sender then knows the phase of the neighbor.

When a sender is about to send a packet to a neighbor for which the sender is aware of its phase, the sender does not start to send its strobes until the neighbor is awake. If the sender misses the awake period of the neighbor, the sender will simply continue to send strobes for an entire period. Thus the mechanism is correct regardless if the phase is correctly recorded or not. With correct phase information, the mechanism is more efficient, however.

If perfectly tuned, the sender will only need to send a single strobe packet when it knows the phase of the neighbor. The Contiki X-MAC does allow for an amount of desynchronization between the nodes, and senders start to send their strobes one awake time earlier than its phase information tells it. On average, the sender will then send two strobe packets for every transmission. We have not yet measured the effectiveness of this mechanism.

Reliable Packet Transmissions

The Contiki X-MAC allows the higher layers to set a flag that tells the MAC layer that a packet is expected to be acknowledged. For such packets, the Contiki X-MAC does not switch off its radio after sending the packet. Instead, it explicitly keeps the radio on in anticipation for an incoming acknowledgment packet. The acknowledgment packet can then be sent without any strobes, since the neighbor knows that the sender's radio is turned on after sending such a packet. This improves the performance of packets with transport layer or application layer acknowledgments.

802.15.4 Compatibility

The Contiki X-MAC provides compatibility with the 802.15.4 link layer standard by forming its messages according to the 802.15.4 specification. All messages, both strobes and data messages, use the 802.15.4 frame and addressing format. This makes it possible to interoperate between an always-on 802.15.4 network and a Contiki X-MAC network. Although the logic is different, and the nodes cannot directly communicate with each other, other mechanisms of the 802.15.4 standard work, such as the ability to discard incoming packets that have a destination address that does not match the node's address.

Implementation

The two Contiki MAC protocols have different requirements in terms of periodicity and exactness and we have therefore used two different techniques when implementing them. In Contiki, MAC layers are a special module that is called by the radio driver when incoming packets, and by the network layer for outgoing packets. Both MAC protocols use timers to implement their behavior.

The Contiki X-MAC implementation uses the Contiki real-time timer module, rtimer, in order to provide an exact timing behavior. At the heart of the Contiki X-MAC implementation is a function called `powercycle()`.
The `powercycle()` function switches the radio on and off as part of the X-MAC idle listening. Because the periods have to be short, and because the timing has to be exact, this function is scheduled by an rtimer to be called at the exact moment at which is it scheduled. Internally, the `powercycle()` function is implemented as a protothread.
When sending or receiving packets, the idle listening is temporarily disabled by a flag being set. The `powercycle()` function is still called, but it does not switch the radio on and off when the flag is set. This ensures that the phase of the duty cycle is stable after sending or receiving packets.
The LPP implementation has lower requirements on exactness of timing. Therefore, the LPP implementation uses the less exact ctimer module for its timing. The ctimer does not provide any guarantees for when a function will be called, but it is significantly simpler to use than the rtimer module because ctimers will never preempt other code.

The LPP function uses a central duty cycling protothread called `dutycycle()` to send probes and to turn the radio on and off. This function is periodically scheduled with a ctimer.

Change mac or radio duty cycling protocols

In low-power networks, the radio transceiver must be switched off as much as possible to save energy. In Contiki, this is done by the Radio Duty Cycling (RDC) layer. Contiki provides a set of RDC mechanisms, with various properties. The default mechanism is ContikiMAC.

The MAC (Medium Access Control) layer sits on top of the RDC layer. The MAC layer is responsible for avoiding collisions at the radio medium and retransmitting packets if there were a collision. Contiki provides two MAC layers: a CSMA (Carrier Sense Multiple Access) mechanism and a NullMAC mechanism, that does not do any MAC-level processing.

This guide shows how to change RDC and MAC layer in Contiki. We assume you have your Contiki project already set up. If not, see the **Develop your first application** guide.

Table of Contents

About MAC Drivers

Contiki provides two MAC drivers, CSMA and NullMAC. CSMA is the default mechanism. The MAC layer receives incoming packets from the RDC layer and uses the RDC layer to transmit packets. If the RDC layer or the radio layer detects a radio collision, the MAC layer may retransmit the packet at a later point in time. The CSMA mechanism is currently the only MAC layer that retransmits packets if a collision is detected.

About RDC drivers

Contiki has several RDC drivers. The most commonly used are ContikiMAC, X-MAC, CX-MAC, LPP, and NullRDC. ContikiMAC is the default mechanism that provides a very good power efficiency but is somewhat tailored for the 802.15.4 radio and the CC2420 radio transceiver. X-MAC is an older mechanism that does not provide the same power-efficiency as ContikiMAC but has less stringent timing requirements. CX-MAC (Compatibility X-MAC) is an implementation of X-MAC that has more relaxed timing than the default X-MAC and therefore works on a broader set of radios. LPP (Low-Power Probing) as a receiver-initiated RDC protocol. NullRDC is a "null" RDC layer that never switches the radio off and that therefore can be used for testing or for comparison with the other RDC drivers.

RDC drivers keep the radio off as much as possible and periodically check the radio medium for radio activity. When activity is detected, the radio is kept on to receive the packet. The channel check rate is given in Hz, specifying the number of channel checks per second, and the default channel check rate is 8 Hz. Channel check rates are given in powers of two and typical settings are 2, 4, 8, and 16 Hz.

The transmitted packet must generally be repeated or "strobed" until the receiver turns on to detect it. This increases the power usage of the transmitting node as well as adding radio traffic which will interfere with the communication among other nodes. Some RDCs allow for "phase optimization" to delay strobing the tx packet until just before the receiver is expected to wake. This requires a good time sync between the two nodes; if the clocks differ by 1% the rx wake time will shift through the entire tx phase window every 100 cycles, e.g. 12 seconds at 8 Hz. This would make phase optimization useless when there are more than a few seconds between packets, as the transmitter would have to start sending well in advance of the predicted receiver wake. A clock drift correction might fix this. See RDC Phase Optimization for more details.

The Contiki RDC drivers are called:

```
contikimac_driver

cxmac_driver

nullrdc_driver
```

Step 1: Add a project-conf.h file to the Makefile

A Contiki project can have an optional per-project configuration file, called '**project-conf.h**'. This file is, however, not enabled by default. To enable it, we need to add the following line to the project's Makefile:

```
CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
```

The full Makefile may now look something like this:

```
CONTIKI = /home/user/contiki
```

```
CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"

include $(CONTIKI)/Makefile.include
```

Step 2: Create the project-conf.h file

We now need to create the project-conf.h file. This file should be called project-conf.h and reside in the project's directory.

The project-conf.h file may override a number of Contiki configuration options. In this example, we will override the radio duty cycle layer driver.

Step 3a: Specify a new RDC channel check rate

We first change the RDC channel check rate. This is done by adding a #define to the project-conf.h file that specifies the channel check rate, in Hz:

```
#define NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE 16
```

Step 3b: Specify a new RDC driver

To specify what RDC driver Contiki should use, add another #define to the project-conf.h file:

```
#define NETSTACK_CONF_RDC nullrdc_driver
```

Step 3c: Specify a new MAC driver

To specify what MAC driver Contiki should use, add another #define to the project-conf.h file:

```
#define NETSTACK_CONF_MAC nullmac_driver
```

Step 4: Recompile

After having specified the project-conf.h file in the Makefile, it is necessary to clean up existing dependencies with the make clean command:

```
make TARGET=sky clean
```

This is only needed the first time, however. Next, just compile as usual:

```
make TARGET=sky
```

Conclusions

Radio Duty Cycling (RDC) and Medium Access Control (MAC) protocols are an important part of the Contiki netstack as they ultimately determine the power consumption of the nodes and their behavior when the network is congested. This guide demonstrates how to change RDC and MAC protocols for a Contiki project by adding a project-conf.h file and adding the necessary configuration statements to it.

Contikimac

ContikiMAC is the default Contiki radio duty cycling mechanism. ContikiMAC allows nodes to keep their radio off for most of the time (> 99%) while being able to relay multi-hop messages.

The report below describes the ContikiMAC mechanism:

The ContikiMAC Radio Duty Cycling Protocol [pdf]

The ContikiMAC header described in the report is added when using the `contikimac_framer`:

```
#undef NETSTACK_CONF_FRAMER
#define NETSTACK_CONF_FRAMER contikimac_framer
```

Do not forget to raise `PACKETBUF_CONF_HDR_SIZE` accordingly.

Reducing Contiki OS firmware size

TODO: This file needs many more additions. Feel free to add and edit.

Size optimizations in gcc

Optimised Makefile

Usually `-Os` is specified as a compiler switch on each module; this tells it to do size optimization, such as discarding unused functions, putting subroutines inline if they are called once, reusing common code such as exit routines, etc. Because of this mixing of routines, the linker is forced to include the entire module if any part of it is called.
This behaviour can be altered by using the `-ffunction-sections` switch to tell the compiler not to mix functions that could be referenced externally, along with the `--gc-sections` switch telling the linker to do garbage collection and discard unreferenced routines. This can save considerable space in modules containing routines that are not used the the current build, but may also result in worse size optimization. Note, the compiler can always optimize functions with the static attribute; when using `--ffunction-sections`, so make sure it is added wherever possible.
Add the following to your Makefile:

```
 CFLAGS += -ffunction-sections
 LDFLAGS += -Wl,--gc-sections,--undefined=_reset_vector__,--undefined=InterruptVectors,--
undefined=_copy_data_init__,--undefined=_clear_bss_init__,--undefined=_end_of_init__
```

Optimised variable allocation

During startup gcc initializes variables located in RAM. Variables that are not declared with initial values (`int foo;`) are gathered into a section called `.bss` which is filled with zeros using a short loop. Initialized variables (`int foo=1;`) are stored in a `.data` section for copying to RAM; typically that storage reduces the amount of program flash memory available for code. Transferring variables to `.bss` and doing your own initialization can reduce the size, e.g. filling arrays with a constant value.
The `objdump` tool will show the size of the initialized RAM in the `.data` section, and the zeroed RAM variables in the .bss section. Toolchains may have their own tool, e.g. `avr-objdump`. Do these variables need 24 bytes of initialization? The `MEMB()` macro is convenient but if you absolutely need those bytes the static pointers could be used directly (also saving 24 bytes of RAM)

```
 $ objdump -t --section=.data udp-client.micaz
 00800100 l    d  .data  00000000 .data
 008001d0 l     O .data  00000008 packet_memb
 008001d8 l     O .data  00000008 metadata_memb
 008001c8 l     O .data  00000008 neighbor_memb
 ...
```

In the next example you can see that the variable `s` takes 34 bytes, does it really need to be static? Examine the map file to see where it is used. *PROTIP:* Searching for 1-letter variables is hard, os you should give longer names to your static variables!

```
$ avr-objdump -t --section=.bss udp-client.micaz
...
008002ce l     O .bss   00000002 outputfunc
008002a3 l     O .bss   00000001 i.3761
008002ac l     O .bss   00000022 s
008002a4 l     O .bss   00000008 periodic
...
```

Reducing the Size of Contiki

There are many features within Contiki that can easily be excluded or reduced in size by setting the correct flags in the `project-conf.h` file that you should have in your application directory. Processes headers contain pointers to ASCII strings that are used only for making out more user-friendly. You can use `#define PROCESS_CONF_NO_PROCESS_NAMES 1` to disable the name strings from being stored, reducing both flash and RAM usage by the combined length of the strings and pointers, typical around 100 bytes.

Reducing uIP/IPv6 Stack Size

TCP and UDP can be separately enabled or disabled with `#define UIP_CONF_TCP` and `#define UIP_CONF_UDP` set to `0` or `1`. Disabling TCP eliminates a MSS-sized buffer. Note, the TCP process needed for RPL-ROLL. Meshing works whether or not TCP is enabled.

Changing the radio duty cycling protocol

The RDC protocols require additional program and RAM to store neighbor information. Reduce the sequence number arrray size used for duplicate packet detection. Turn off phase optimization in contikimac to eliminate the neighbor wake time table. Use #define NETSTACK_CONF_RDC nullrdc_driver for the smallest size.

Excluding the uIP/IPv6 Stack

You can set `CONTIKI_NO_NET=1` in the Makefile and exclude all the networking code. You can then add individual files or routines (e.g. checksum calculations) as needed.
See `Makefile.ravenusbstick` and `fakeuip.c` for an example; by default the usb stick is just a network bridge with no *uIP* stack of its own, although the stack can be enabled to support an internal webserver.

Useful commands

```
$ size webserver6.elf
   text    data     bss     dec     hex filename
  58493     545   11314   70352   112d0 webserver6.elf
```

data is preinitialized RAM, bss is prezeroed RAM. You probably have an msp430-objdump tool but a vanilla objdump will do the job if you don't require disassembly

```
$ objdump -h webserver6.elf

webserver6.elf:     file format elf32-little

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         000001e8  00800100  0000e47a  0000e56e  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  1 .text         0000e47a  00000000  00000000  000000f4  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .bss          00002c32  008002e8  008002e8  0000e756  2**0
                  ALLOC
  3 .eeprom       00000036  00810000  00810000  0000e756  2**0
                  CONTENTS, ALLOC, LOAD, DATA
...
```

To drill down and see which variables are taking ram:

```
$ objdump -t --section=.data webserver6.elf
00800100 l    d  .data  00000000 .data
00800122 l     O .data  00000008 conns
0080012d l     O .data  00000006 file
00800133 l     O .data  00000006 tcp
...
$ objdump -t --section=.bss webserver6.elf
008002e8 l    d  .bss   00000000 .bss
008002e8 l     O .bss   00000002 filelength.3481
008002ec l     O .bss   00000015 scriptname.3479
```