TECHNISCHE UNIVERSITÄT DRESDEN

CENTER FOR ADVANCING ELECTRONICS DRESDEN

Networked Embedded Systems Group Dr. Marco Zimmerling

# Networked Embedded Systems WS 2016/17

## Sample Solutions to Lab: Programming an Embedded Wireless Device

Discussion date: December 14, 2016

## Task 0: Introduction

The goal of this lab is that you get some hands-on experience in programming an embedded wireless device. Therefore, we start with an introduction about the platform we use and how to program it.

### Platform

The platform we use in this lab is the MTM-CM5000MSP. It features the following hardware components:

- Microcontroller: MSP430F1611 (16 bit, 8 MHz, 10 KB RAM)
- Radio: CC2420 (2.4 GHz, IEEE 802.15.4 compliant)
- Sensors: Temperature, Humidity, Light, ...
- Others: USB, 2 Buttons, 3 LEDs

### Programming

In this lab, we use the Contiki operating system (`http://www.contiki-os.org/`). It is written in the C programming language and has become one of the most popular operating systems for resource-constrained, embedded devices. Here are some facts about Contiki:

- Open-source and active community
- Tiny memory footprint (few dozen bytes up to a few kilobytes)
- Cooperative multithreading (protothreads)
- Supports multiple Internet and low-power wireless communication standards (e.g. IPv6/4, 6lowpan)
- Cooja network simulator allows to debug applications in networks of fully emulated hardware

These features simplify the development of wireless embedded applications (e.g., for the Internet of Things).

## Setup

First of all, you should set up your workspace and download the archive `https://wwwpub.zih.tu-dresden.de/~mzimmerl/teaching/eti/labs/eti.tar`. It contains a stripped version of Contiki's code base under the `contiki/` directory. Besides that, there is the manual of our platform `cm5000msp.pdf` and a helper script `helper.py`. The code for the programming tasks is located under `contiki/tasks/`.

To compile the tasks and upload them to the microcontroller, you need to install the following software on your system:

- Docker (`https://www.docker.com/`)
  - Follow the installation instructions at `https://docs.docker.com/engine/installation/`
  - Verify that Docker is installed correctly and running:
    `$ docker version` should print information about the Docker client and server
  - Linux: If the server is not running, type the following command and then logout and login.
    `$ sudo usermod -aG docker $(whoami)`

In addition, you need:

- Python 2.7.x (`https://www.python.org/`), `$ python --version` or `$ python2 --version`
- Perl (`https://www.perl.org/`), `$ perl --version`

but these should already be installed on your system.

## Workflow

The archive contains a helper script named `helper.py`. This script helps you compile your code, upload the binary to the microcontroller, and view the serial output, produced by `printf()` statements in your code, of the program that runs on the device. Usage information are printed when typing `$ python helper.py -h`. Make sure you are in the same directory as the script when running it. The general workflow looks like this:

(1) Write code

(2) Compile code `$ python helper.py <task-number> compile`

(3) Upload the binary to the device `$ python helper.py <task-number> upload`
   - The script may ask you for your password in order to access the USB device.
   - Sometimes the first upload attempt fails; in this case, try again.

(4) Check the serial output of your program `$ python helper.py <task-number> login`

## Behind the Scenes

Compiling code for our platform requires to setup the toolchain, consisting of a compiler (e.g. `gcc-msp430`), the standard C libary for MSP430 development and serveral other files and scripts. To avoid cluttering your systems, we packaged the whole toolchain into a Docker[1] container (`fmag/nes-dev-box`), which is comparable to a virtual machine but much more lightweight. When you use the helper script to compile a task, an instance of our docker container is started. The code gets compiled inside the container and eventually the container is removed and does not occupy any resources anymore. Another advantage is that you can use your familiar development environment to solve the programming tasks.

After compilation, the application needs to be uploaded to the device; that is, the microcontroller needs to be programmed. The platform we are using provides a bootloader (BSL) interface to communicate with the embedded memory in the microcontroller. Luckily, Contiki comes with a BSL script that manages this interaction. Our helper script uses this BSL script and tells it which USB device to program.

[1]For more information `https://docs.docker.com/engine/understanding-docker/`

## Task 1: Let It Blink

The first task uses the three LEDs of our platform. LEDs are a valuable tool for debugging timing-sensitive code, where `printf()` statements would introduce delays that alter the behavior of the code. Instead, toggling a LED is very fast.

(a) Open the source code of task 1 (`contiki/tasks/task1/task1.c`) and take a look at the structure and the comments.

(b) Take a look at Contiki's LED API (`contiki/core/dev/leds.h`) and turn on all LEDs (red, green, blue) inside the `led_test()` function. Compile and upload your code to see if it works.

(c) Use Contiki's rtimer library (`contiki/core/sys/rtimer.h`) to implement the following behavior. Turn on each LED for exactly 1 second and then off in the following order, one after the other: red, green, blue, red, green, blue, ... After turning on one LED, use `rtimer_set()` to schedule when the LED should be turned off.

Calling `rtimer_set()` should be followed by `PT_YIELD(&pt)` to wait until the time, specified in the `rtimer_set()` call, has passed. The source code of this task already uses the function `rtimer_set()` to schedule the call of `led_test()`. Take this as an example for how to use `rtimer_set()`. Use macros `RTIMER_NOW()` and `RTIMER_SECOND` to compute the point in time when the rtimer should execute the callback function. The first macro returns the current system time in clock ticks, and the second macro specifies the number of clock ticks per second. In order to solve this task, it is important to understand how `PT_BEGIN(&pt)`, `rtimer_set()` and `PT_YIELD(&pt)` play together. The following figure shows a code fragment on the left side and the corresponding sequence of instructions during runtime on the right side.

```
 1  static char test(struct rtimer *t, void *ptr) {
 2     PT_BEGIN(&pt);
 3     printf("A");
 4     rtimer_set(...);
 5     PT_YIELD(&pt);
 6     printf("B");
 7     rtimer_set(...);
 8     PT_YIELD(&pt);
 9     printf("C");
10     PT_END(&pt);
11  }
```

```
 2  PT_BEGIN(&pt);      // start of protothread
 3  printf("A");
 4  rtimer_set(...);    // schedule next event
 5  PT_YIELD(&pt);      // remember position and give up CPU
    // wait for event
 2  PT_BEGIN(&pt);      // jump to last position
 6  printf("B");
 7  rtimer_set(...);    // schedule next event
 8  PT_YIELD(&pt);      // remember position and give up CPU
    // wait for event
 2  PT_BEGIN(&pt);      // jump to last position
 9  printf("C");
10  PT_END(&pt);        // end of protothread
```

(d) Take your solution from (c) and reduce the LED on-time by 100 ms after each round (red, green, blue). What happens after 11 rounds and why? Pay attention to the fact that a protothread does not have its own stack.

**Solution:** *What happens after 10 rounds and why?*
The second parameter of `rtimer_set()` has the type `rtimer_clock_t`, which is an `unsigned short` and cannot be negative. At the beginning, the LED on-time starts with 1 second and gets decremented by 100 ms each round. When the on-time reaches 0 seconds and is decremented again, an underflow occurs. As a result, a value of -1 is treated as the maximum value of `unsigned short`, this is 65535 since `unsigned short` is 16 bit on our platform. 65535 is twice as much as `RTIMER_SECOND`, hence the program continues with a LED on-time of 2 seconds.

Below is the final solution after solving subtask (d).

```
#include "contiki.h"
#include "dev/leds.h"
#include "sys/rtimer.h"

// Define a process control block that contains information about the process
// and points to a process thread.
PROCESS(task1_process, "Task 1");
// Define the process that is started automatically after boot.
```

```
AUTOSTART_PROCESSES(&task1_process);

static struct rtimer rt;
static struct pt pt;

static char led_test(struct rtimer *t, void *ptr) {
        PT_BEGIN(&pt);

        // Variable to store LED on-time. Protothreads do not have a dedicated
        // stack so we have to make this variable static in order to remember the
        // value after calling PT_YIELD().
        static rtimer_clock_t ontime = RTIMER_SECOND;

        while (1) {
                // Turn on red LED and schedule when to turn off.
                leds_on(LEDS_RED);
                rtimer_set(t, RTIMER_NOW() + ontime, 0, (rtimer_callback_t)led_test, ptr);
                PT_YIELD(&pt);

                // Turn off red LED. Turn on green LED and schedule when to turn off.
                leds_off(LEDS_RED);
                leds_on(LEDS_GREEN);
                rtimer_set(t, RTIMER_NOW() + ontime, 0, (rtimer_callback_t)led_test, ptr);
                PT_YIELD(&pt);

                // Turn off green LED. Turn on blue LED and schedule when to turn off.
                leds_off(LEDS_GREEN);
                leds_on(LEDS_BLUE);
                rtimer_set(t, RTIMER_NOW() + ontime, 0, (rtimer_callback_t)led_test, ptr);
                PT_YIELD(&pt);

                // Turn off blue LED.
                leds_off(LEDS_BLUE);
                // Compute LED on-time for the next round.
                ontime = ontime - RTIMER_SECOND / 10;
        }

        PT_END(&pt);
}

// Define a process thread that contains the code of a process.
PROCESS_THREAD(task1_process, ev, data) {
        // Beginning of a process.
        PROCESS_BEGIN();

        // Initialization of protothread structure.
        PT_INIT(&pt);
        // Timer Initialization.
        rtimer_init();

        // Schedule led_test() for the first time.
        rtimer_set(&rt, RTIMER_NOW(), 0, (rtimer_callback_t)led_test, NULL);

        // Wait forever.
        while (1) { PROCESS_WAIT_EVENT(); }

        // End of a process.
        PROCESS_END();
}
```

## Task 2: Let It Sense

Cyber-physical systems are based on a close interaction with the physical world. Sensors sample the physical environment and pass their readings to a controller. The controller computes the next control signals and forwards them to actuators that act accordingly on the physical world.

(a) Open the source code of task 2 (`contiki/tasks/task2/task2.c`) and take a look at the structure and the comments.

(b) Contiki manages different kinds of sensor data in a common data structure named `sensors_sensor`, which is defined in `contiki/core/lib/sensors.h`. Specific information for each of the sensors can be found at `contiki/platform/sky/dev/` and `contiki/dev/sht11/`. Write code so that the values of the light, temperature and humidity sensors are printed each time the USER button is pressed. Sensor readings can be retrieved in the following way: `sensor.value(TYPE)`. Test your code by logging into your device to check the output.

(c) The returned sensor readings are "raw" values in most cases. Raw means that the values are based on the internal design and circuitry of a given sensor. We need to convert these values into an useful unit of measurement. This conversion is likely to be different for each type of sensor, hence we need to check the manual of our platform. Open the manual `cm5000msp.pdf`, which is located at the top level of our archive, and look up the required conversion formulas. Change your code to print out light in lux, relative humidity in percent, and temperature in degree celsius.

**Solution:** Below is the final solution after solving subtask (c).

```
#include "contiki.h"
#include "dev/button-sensor.h"
#include "dev/light-sensor.h"
#include "dev/sht11/sht11-sensor.h"
#include "sys/rtimer.h"
#include <stdio.h>

// Define a process control block that contains information about the process
// and points to a process thread.
PROCESS(task2_process, "Task 2");
// Define the process that is started automatically after boot.
AUTOSTART_PROCESSES(&task2_process);

// Read value of light sensor (visible light).
static int get_light(void) {
        return light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
}

// Read and convert value of humidity sensor.
static int get_humidity(void) {
        int raw_hum = sht11_sensor.value(SHT11_SENSOR_HUMIDITY);
        return -4 + (0.0405 * raw_hum) + (-2.8 * 0.000001) * (raw_hum * raw_hum);
}

// Read and convert value of temperature sensor.
static int get_temperature(void) {
        return sht11_sensor.value(SHT11_SENSOR_TEMP) * 0.01 - 39.6;
}

// Define a process thread that contains the code of a process.
PROCESS_THREAD(task2_process, ev, data) {
        // Beginning of a process.
        PROCESS_BEGIN();

        // Activate platform sensors.
        SENSORS_ACTIVATE(light_sensor);
        SENSORS_ACTIVATE(button_sensor);
        SENSORS_ACTIVATE(sht11_sensor);

        while (1) {
                // Wait until USER button is pressed.
                PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event && data == &button_sensor);
                // Print sensor readings.
                printf("Light: %d lx\n", get_light());
                printf("Humidity: %d %%\n", get_humidity());
                printf("Temperature: %d C\n", get_temperature());
        }

        // End of a process.
        PROCESS_END();
}
```

## Background Information: Interrupt Handling

The source code of this task already detects when the USER button is pressed. This works by catching the interrupt signal that is fired as soon as we press the button. Interrupt handling is one of the core principles in embedded systems since it allows you to react to certain events. Contiki simplifies this for the developer and provides the API call `PROCESS_WAIT_EVENT_UNTIL()`, which allows to wait for a specific event.

Setting up interrupt handling by hand, however, involves multiple steps. The USER button is connected to a digital I/O pin of our microcontroller. This pin has to be configured to act as an input pin since our button is the source of information. Furthermore we have to explicitly enable interrupts on this pin and tell the microcontroller to watch for rising, falling, or both edges. After setting this up, an interrupt service routine (ISR), which contains the code we want to execute when an interrupt occurs, has to be implemented. Finally, when an interrupt is triggered, we have to determine the source of the interrupt because there are many possibilities. This information can be retrieved from a so-called interrupt vector that additionally has to be reset after serving an interrupt to enable future interrupts.

## Task 3: Let It Speak

Communication is an integral part in distributed systems, where sensor readings or other information must be transmitted. Our platform is equipped with an IEEE 802.15.4 compliant radio. This standard was designed for low-data-rate wireless communication among low-power battery-driven devices.

(a) Open the source code of task 3 (`contiki/tasks/task3/task3.c`) and take a look at the structure and the comments.

(b) Contiki provides a set of lightweight and easy-to-use communication primitives. Use the broadcast primitive, found at `contiki/core/net/rime/broadcast.h`, to send some user-defined data upon pressing the USER button.

Before you can send data, you need to set up the broadcast connection to operate on channel 1. The radio sends the data that is stored in the packet buffer. User-defined data needs to be copied into this buffer before calling the send operation. The API for handling the buffer is located at `contiki/core/net/packetbuf.h`. Test your code and verify that your message is successfully received by the device of your tutor.

(c) The device of the tutor periodically broadcasts some data. Implement the broadcast receive callback and print out the received data. The received data is stored in the packet buffer.

**Solution:** Below is the final solution after solving subtask (c).

```
#include "contiki.h"
#include "dev/button-sensor.h"
#include "net/rime/rime.h"
#include <stdio.h>

// Define a process control block that contains information about the process
// and points to a process thread.
PROCESS(task3_process, "Task 3");
// Define the process that is started automatically after boot.
AUTOSTART_PROCESSES(&task3_process);

// Callback function after receiving a broadcast message.
static void broadcast_recv(struct broadcast_conn *c, const linkaddr_t *from) {
        printf("Received \"%s\" from %d\n", (char *)packetbuf_dataptr(), from->u8[0]);
}

// Assigning the callback function.
static const struct broadcast_callbacks broadcast_call = {broadcast_recv};
// Structure for managing general broadcast information.
static struct broadcast_conn broadcast;

// Define a process thread that contains the code of a process.
PROCESS_THREAD(task3_process, ev, data) {
        // Beginning of a process.
        PROCESS_BEGIN();

        // Activate button sensor.
        SENSORS_ACTIVATE(button_sensor);

        // Setup broadcast connection.
        broadcast_open(&broadcast, 1, &broadcast_call);

        while (1) {
                // Wait until USER button is pressed.
                PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event && data == &button_sensor);

                // Copy user data into packet buffer.
                packetbuf_copyfrom("Hello", 6);
                // Start broadcast.
                broadcast_send(&broadcast);
        }

        // End of a process.
        PROCESS_END();
}
```