Porting Contiki To New Hardware Platforms

Author: Jelle Aerts

Supervisor: Steffen Thielemans

June 2016
Preface

This text is an extract from a Master’s Thesis conducted at the ETRO Department of the Vrije Universiteit Brussel in June 2016.
# Table of Contents

1 Introduction  
2 Contiki’s directory structure  
3 Components to port  
   3.1 Configuration files  
   3.2 Clock module  
   3.3 Rtimer  
   3.4 Watchdog  
   3.5 LEDs  
   3.6 Serial line driver  
   3.7 Sensors  
   3.8 Node MAC address & node ID  
   3.9 SLIP driver  
   3.10 Radio driver  
   3.11 Low power mode  
4 Main function  
5 Code upload mechanisms  
6 Makefiles  
Bibliography  
A Porting Contiki to LoRaMote  
   A.1 Clock module  
   A.2 Rtimer  
   A.3 Watchdog  
   A.4 LEDs  
   A.5 Serial line driver  
   A.6 Button sensor  
   A.7 Node MAC address & node ID  
   A.8 SLIP driver  
   A.9 Radio driver  
   A.10 External clock
A.11 Low power controller ................................................. 34
A.12 Low power clock module ........................................... 35
A.13 Low power rtimer .................................................... 37
A.14 Contiki main ......................................................... 40
A.15 OpenOCD configuration file ....................................... 43
A.16 CPU Makefile ....................................................... 43
A.17 Platform Makefile .................................................. 47
List of Acronyms

API  Application Programming Interface
ARM  Acorn RISC Machine
CPU  Central Processing Unit
DFU  Device Firmware Upgrade
ELF  Executable and Linking Format
GPIO  General-Purpose Input/Output
I/O  Input/Output
IP  Internet Protocol
LCD  Liquid-Crystal Display
LED  Light Emitting Diode
LPM  low power mode
MAC  Media Access Control
RPL  IPv6 Routing Protocol for Low-Power and Lossy Networks
SLIP  Serial Line Internet Protocol
TCP  Transmission Control Protocol
UART  Universal Asynchronous Receiver/Transmitter
uIP  micro IP
USART  Universal Synchronous/Asynchronous Receiver/Transmitter
USB  Universal Serial Bus
1 | Introduction

The Contiki operating system is designed to run on resource constrained, low-power Wireless Sensor Network nodes and is constructed in such a way that it should be easy to deploy to new hardware platforms. Hence, the majority of its components is completely platform-independent. When deploying Contiki to a new platform, only a handful of platform-dependent components have to be implemented to get the whole system up and running. One of the goals of this project is to provide the Contiki community with a detailed guide on how to port the Contiki operating system to new hardware platforms by implementing these platform-dependent components. The developers state that the operating system is indeed easy to port [1, 2] but a quick search in the official mailing lists [3] teaches us that a lot of users still struggle with this matter. Existing information on porting Contiki often neglects important details or does not treat certain aspects. Yair Hershkovitz and Demitry Lev for example wrote a porting guide [4] but merely described the basic porting components of Contiki. The same applies to Wincent Balin’s documentation [5] for his entry to the MSP430 Design Contest which was held in 2006. The University of Bern held a seminar in 2009 where Cyrill Schluep presented the steps of how to port Contiki to new platforms [6], but once again it only included an enumeration of the basic steps. In 2011 George Oikonomou shared his own porting experiences, but started from an existing discontinued porting effort [7] and thus does not contain any details about how to start a new port. Zhun Shen and his team even published a paper in 2013 which describes how to port a platform from Contiki to another operating system [8]. Adam Dunkels’ original presentation remains one of the best sources available although it dates from 2007 and is nowadays only available from archived copies [9]. In the same year Alexandru Stan wrote his Bachelor thesis which covers the same topic a bit more extensively [10]. Both of these works are excellent sources with examples, but are a bit outdated and lack information about low power mode implementations for microcontrollers.

All of these papers thus either lack information, are not detailed enough or are simply outdated. They also often refer to the 'native' platform which is provided within the Contiki repository and is supposed to be the simplest port for a new platform. In reality, it contains a lot of code which is not essential to get Contiki running which is confusing for users who have no experience with Contiki’s inner workings. Next to the mentioned papers, more guidelines can be found on the official Contiki wiki page and within the source files [11]. As a result of all these scattered and incomplete sources of information on how to port Contiki, one can have a hard time to actually get it working while it should
be in fact easy to do. This is the main reason why there was a need for a unified document with a detailed guide.

In the rest of this document an enumeration of ’to be ported’ components will be presented as well as the steps to be taken on how to do it. All will be accompanied by examples from our own Contiki port to the LoRaMote demo platform which wields an STM32L151C8 microcontroller [12, 13], using the STM32L1 Standard Peripherals Library [14].
2 | Contiki’s directory structure

The main Contiki directory [15] contains multiple files and folders. Those with any importance will be discussed although they are not necessarily of interest for porting purposes. Contiki’s base system resides in the /core directory and has no need to be ported [9]. Its code should only be consulted for more information on the inner workings, but should not be changed in order to 'fix' a process. To perform a port there are two relevant directories, namely /platform and /cpu. Only these two can contain architecture dependent code. Typically, the cpu folder will be used by all platforms with the same microcontroller and contains specific code like startup files, libraries, timers, watchdogs, multi-threading, low power mode controllers and an ELF loader. Generally the platform folder contains all other code which is not related to the CPU. This commonly includes the main function, LED control, sensor reading and all other functions the board is capable of. A last directory that will be used is /tools. This is the storage place for all tools that will be used during the build and upload process (e.g. hardware programmer, debugger, mote listing tools, terminal, etc.). One file of interest is Makefile.include. It contains the make instructions for the Contiki base system and will build the whole project once a destination platform has been chosen.

The first step of porting Contiki should be creating the appropriate subfolders in the /cpu, /platform and /tools directories. The two first directories should be provided with their own makefile, which are respectively /cpu/{CPU_NAME}/Makefile.{CPU_NAME} and /platform/{PLATFORM_NAME}/Makefile.{PLATFORM_NAME}. The contents of these files include instructions to compile, link and upload code. Their implementation will be discussed later. Then two new folders should be made in the personal platform directory, named 'dev' and 'net'. The dev folder will contain the files with the low level code for sensors while the net folder will contain the files with the low level code for the radio driver.

Using this information, the following directories should be created for the LoRaMote platform example:

- {CONTIKI_PATH}/cpu/arm/stm32l1
- {CONTIKI_PATH}/platform/loramote
- {CONTIKI_PATH}/platform/loramote/dev
- {CONTIKI_PATH}/platform/loramote/net
Chapter 2. Contiki’s directory structure

- `{CONTIKI_PATH}/tools/loramote`

Next, the STM32L1 Standard Peripherals Library should be placed in the `/cpu/arm/stm32l1` directory, together with linker scripts and system startup files for the STM32L1 that can be found in the LoRaMac GitHub repository [13]. These files will be crucial to build the whole project for this platform.
3 | Components to port

The native platform which is included in Contiki’s source code is often referred to as a porting template. Although it indeed contains all basic necessary implementations, it is not always clear what is happening on each line of code when one has no or limited foreknowledge as mentioned before. Needless to say, when there should be a platform already available which is similar to the one that is being ported, it is more interesting to use that code as a starting point. Some caution is advised here since not all ports are completely finished and thus can result in unexpected results. Either way, it is always easier to start from existing code in order to have the basic structures.

In the following section the different porting components will be discussed, accompanied with examples from the LoRaMote demo board. Since the multi-threading library is not used often and the ELF loader requires great knowledge of the ELF format and the CPU architecture [9], these components are out of the scope of this paper and are omitted.

3.1 Configuration files

Each platform should contain two configuration files: contiki-conf.h and platform-conf.h. The Contiki configuration file contains several configuration options related to the operating system like C compiler instructions, C types, uIP settings, duty cycling preferences, apps and widgets preferences, etc. [9]. The platform configuration file can be used for platform specific settings like clock configuration and constants used by the platform’s clocks, sensors or other components. These configuration files can be copied from the native platform or a platform that is similar to the one that is being ported. Later on during this guide, several configuration options will be added to these files.

Practically, the /platform/native/contiki-conf.h configuration file is copied to /platform/loramote/contiki-conf.h. The clock_time_t and CLOCK_CONF_SECOND definitions should be removed from this new Contiki configuration file. Then a new file /platform/loramote/platform-conf.h is created, containing the default clock configuration.

```c
#ifndef __PLATFORM_CONF_H__
define __PLATFORM_CONF_H__

typedef unsigned long clock_time_t;
typedef unsigned long long rtimer_clock_t;
```
Chapter 3. Components to port

3.2 Clock module

The clock module (clock.c) is one of Contiki’s most essential components as it handles most of the timed events used by the system and its applications. The objective of the clock module is to generate a certain amount of ticks each second and to continue processes if their etimer or ctimer have expired. Such pending processes are started by calling the etimer_request_poll() function. The amount of ticks per second can be chosen freely, but is usually a power of two. The reason for this is that their divisions can be optimized greatly [16, page 92]. A number between 64 and 512 ticks is common, depending on which granularity is requested.

According to Contiki’s official documentation on timers, there are six components of the clock module API that need porting [11, Section Timers].

- **clock_time_t clock_time();**
  Returns the amount of ticks that have passed since the initialization of the clock.

- **unsigned long clock_seconds();**
  Returns the amount of seconds that have passed since the initialization of the clock.

- **void clock_delay(unsigned int delay);**
  Shortly delays the processor by executing some no-operations (NOPs).

- **void clock_wait(int delay);**
  Delays the system for a certain amount of clock ticks.

- **void clock_init(void);**
  Initializes the clock module.

- **CLOCK_CONF_SECOND;**
  Defines the number of ticks per second.

For the LoRaMote example, the STM32 Systick structure has been used. During clock initialization the CPU is told to fire an interrupt several times per second in order to increase the tick counter and check for processes which are ready to continue their tasks. The full implementation can be found in Appendix A.1. In the implementation of the Systick interrupt, two ENERGEST calls can be found. These are used by Contiki to create an estimation of the power consumption. More of these calls will be found in the implementation of other components. The RELOAD_VALUE constant uses two other constants, namely F_CPU and CLOCK_SECOND. F_CPU refers to the frequency of the main processor and should be defined in the platform configuration file. CLOCK_SECOND is a constant defined by Contiki and will take the same value as CLOCK_CONF_SECOND. CLOCK_CONF_SECOND should be defined in the platform configuration file as well and is never supposed to be called directly.

The following two instructions are thus added to the platform-conf.h file.
3.3 Rtimer

The rtimer (rtimer-arch.c) is an independent clock. It differs from the clock module by giving extensive control to its users and not being used by Contiki itself. Its sole purpose is to precisely schedule one event at a time and it is used by configuring a wake up time and a callback function. This callback function profits from being preemptive, meaning that when the rtimer expires the current process will be stopped in favor of executing this rtimer callback function [11, Section Processes & Section Timers]. The user callback function is triggered by executing the rtimer_run_next(). The amount of ticks that this clock produces is much higher than the main clock for great precision, typically a power of two ranging between 8192 and 32768.

In order to port the rtimer, the following items have to be implemented [11, Section Timers]:

- **RTIMER_CLOCK_LT(a, b);**  
  Should return TRUE if 'a' is less than 'b', otherwise FALSE.

- **RTIMER_ARCH SECOND;**  
  Defines the number of ticks per second.

- **void rtimer_arch_init(void);**  
  Initializes the rtimer module.

- **rtimer_clock_t rtimer_arch_now();**  
  Returns the amount of ticks that have passed since the initialization of the clock.

- **int rtimer_arch_schedule(rtimer_clock_t wakeup_time);**  
  Schedules a callback to a user function.

In the LoRaMote example the STM32 TIM structure is used to implement the rtimer. The full implementation can be found in Appendix A.2. As in the clock module example, the initialization of this timer configures an interrupt that will handle the rtimer tick counter and optionally call the users’ callback function. This implementation is not necessary. It can also be implemented without a tick counter by configuring an interrupt that will immediately call the callback function. This will be shown in Section 3.11 discussing low power mode. The used RTIMER_SECOND constant is defined by Contiki which refers to RTIMER_ARCH_SECOND. Just like the CLOCK_CONF_SECOND constant, RTIMER_ARCH_SECOND should never be called directly.

Again two constants should be added to the platform-conf.h file.
## 3.4 Watchdog

A watchdog is a hardware device that is able to recover a microcontroller from system errors [17]. It makes use of a counter that will count down to zero. The system should reset this counter to its original value on a regular basis. If this would fail and the counter reaches zero, the watchdog will judge that the application is not executing in a normal or expected way and will reset the microcontroller. A variant of the normal watchdog is a windowed watchdog. The principle of the countdown counter stays the same, but an extra condition is imposed. If the application resets the counter too soon, the windowed watchdog will suspect a malfunction too and will reset the microcontroller.

Contiki’s watchdog interface (watchdog.c) requires five functions to be implemented:

- **void watchdog_init(void);**
  - Initializes the watchdog module.

- **void watchdog_start(void);**
  - Starts the watchdog countdown process.

- **void watchdog_stop(void);**
  - Stops the watchdog countdown process.

- **void watchdog_periodic(void);**
  - Resets the watchdog’s countdown counter.

- **void watchdog_reboot(void);**
  - Restarts the watchdog device.

Both watchdog variants are implemented for the LoRaMote example in Appendix A.3. The windowing feature of the windowed watchdog is not used here. The STM32 watchdogs cannot be disabled once started [18, page 135 & 553], therefore this function is implemented as a stub. Some new constants are to be added to the platform configuration file. These give the possibility to select the preferred variant and to control the watchdog activation and its timeout value.

```c
/* LSI clock frequency */
#define F_LSI 37000

/* Start watchdog */
#define ENABLE_WATCHDOG 1

/* Use independant watchdog */
#define WATCHDOG_USE_IWDG 1

/* Independant watchdog timeout in milliseconds */
#define WATCHDOG_IWDG_TIMEOUT 250
```
3.5 LEDs

One of the most basic ways to get some feedback from the platform is by using Light Emitting Diodes (LEDs). Usually it is also one of the easiest components to implement. By default, Contiki recognizes up to three differently colored LEDs: red, green and yellow/blue. The control interface (leds-arch.c) makes use of bitmasks to get and set the current LED configuration. The form of this bitmask can be chosen by the platform itself by defining the LEDS_RED, LEDS_GREEN, LEDS_YELLOW and LEDS_BLUE constants as unique power of two values. The LEDS_ALL constant then should be defined as the maximum value than can be achieved by adding all the separate LED values. The default bitmask definition can be found in the /core/dev/leds.h file. The Contiki LED interface can be used once the following three functions are implemented:

- **void leds_arch_init(void);**
  - Initializes the LED hardware structures.

- **unsigned char leds_arch_get(void);**
  - Returns a bitmask of the currently active LED configuration.

- **void leds_arch_set(unsigned char leds);**
  - Lights the LEDs according the given bitmask.

The LoRaMote example makes use of Contiki’s default bitmask and uses the LoRaMac GPIO interface to handle the LED configuration. Its full implementation can be found in Appendix A.4. In order to indicate that the platform features LEDs, an extra constant can be added to the platform configuration file.

```c
/* Indicate that LEDs are available to use */
#define PLATFORM_HAS_LEDS 1
```

3.6 Serial line driver

Solely using the LEDs as feedback is a bit restrictive. It is much more interesting to be able to get some output by the means of text messages. This can be achieved by using a hardware debugger or by configuring a serial line connection to the board. The usage of a hardware debugger will be discussed in the Code upload mechanisms section (5), while this section will focus on the establishment of a serial line connection.

Whenever a printf command is executed in the C language, its parameters are passed on through several other commands until the moment it reaches a specific system call. On a machine with an operating system, the system will make sure the output is redirected correctly to an output console. On embedded systems however, such underlying I/O dispatching service is not available. Therefore the main principle of getting text output from bare metal platforms is to implement a so called 'sink’. A sink is a framework that will capture all standard I/O calls and redirect them elsewhere. The destination of this redirection or a combination of destinations can be chosen from a range of different...
peripheral possibilities. The message can be relayed over a network for example or it can be shown on an LCD display. The goal of this module is however to redirect the message through a UART/USART hardware device so it can be read by a connected terminal application. Such UART/USART device is responsible for translating a sequence of bits to a serial series of 'logical voltage levels' [19]. These virtual voltages can then be used by a serial communication driver to transmit them physically. Usually, a terminal application physically connects to the board using an RS-232 serial port. On newer boards however, it is more common to use USB Virtual COM Ports which emulate RS-232 connections. So in order be capable of using printf commands in Contiki applications, this I/O intercepting sink framework needs to be implemented. The functions that are required by this framework depend on the used compiler toolchain.

- IAR C Compiler:
  
  ```
  size_t __write(int handle, const unsigned char *buf, size_t bufsize);
  ```

- ARM C Compiler:

  ```
  int fputc(int ch, FILE *f);
  ```

- GNU C Compiler:

  ```
  int __io_putchar(int ch);
  size_t _write(int handle, const unsigned char *buffer, size_t size);
  ```

Whereas Contiki uses the default sink framework for output redirection, it does provide an interface for user input handling [11, Section Input and Output]. The first step should be to enable an interrupt on the UART/USART device whenever it receives data. The interrupt callback function then should pass on the received character to Contiki’s serial_line_input_byte() function. This serial line interface will buffer all received characters until it detects a newline character. Whenever the newline character is entered, Contiki will send the received text string to all active processes for further usage.

In the LoRaMote example project the GNU C Compiler was used. The _write() function should call the __io_putchar() function for each character in its buffer. Then each character can be redirected to the desired end device, in this case the Virtual COM Port structure of the USB interface. All implementations can be found in Appendix A.5: the _write() function in sysexcalls.c and __io_putchar() in serial-line-arch.c. The latter file also includes the code for serial user input handling.

### 3.7 Sensors

Contiki provides its users with a unified sensor interface which can be used to retrieve the current measured value or to get notified on sensor changes. This way all sensors can be accessed and controlled in a similar way regardless of the underlying peripheral technology and microcontroller. As a result of this abstraction, applications written for Contiki can become completely platform independent. The Contiki sensor structure consists of the seven functions listed below and interrupt handlers if needed. Note that only the first four functions have to be implemented since Contiki provides the implementation of the last
Chapter 3. Components to port

three.

- **static void init(void);**  
  Initializes the sensor’s hardware

- **static void activate(void);**  
  Activates the sensor.

- **static void deactivate(void);**  
  Deactivates the sensor.

- **static int value(int type);**  
  Returns the sensor’s current value.

- **static int active(void);**  
  Returns the sensor’s current activation status.

- **static int configure(int type, int value);**  
  Contiki’s interface to control the sensor.

- **static int status(int type);**  
  Contiki’s interface to get the sensor’s status.

In the sensor showcase found in Appendix A.6, the LoRaMote’s SX9500 proximity sensor is implemented as a button. It is a device that is connected to the SX1509 I/O Expander, therefore a cascading interrupts implementation is needed. When the SX9500 detects a value higher than the configured threshold, an interrupt will be generated. The SX1509 will detect that the SX9500 fired an interrupt and will generate another interrupt itself. This interrupt will be captured by the STM32 EXTI structure which will call Contiki’s sensors_changed() function in order to let the system know that a sensor event occurred. The implementation of this cascaded interrupts construction is pretty low level and required the help of the datasheets of the SX1509 I/O Expander [20], the SX9500 proximity sensor [21] and the iM880A microcontroller module [22]. Lastly, to indicate that this platform wields a button, a new entry is made in the platform configuration file.

```c
#define PLATFORM_HAS_BUTTON 1
```

### 3.8 Node MAC address & node ID

In order to be able to connect to a device on the data link layer, it needs a unique MAC address. This MAC address will be used by the network stack to send encapsulated network packets, called frames, to other devices. Most network interfaces already come with a unique MAC address, but sometimes it can happen that they have to be generated. If a MAC address has to be generated, other unique properties of the device are to be used. Contiki’s devices also bear a node ID which is usually derived from the MAC address. It can also automatically derived from another source or even manually set. The way in which this ID is assigned can be freely chosen by the developer. Contiki requires two functions (node-id.c) related to the node’s MAC address and ID:
3.9 SLIP driver

In Section 3.6 the process of redirecting the device’s input and output was described. Of course, this way of communication is a bit basic and thus restrictive. There are other technologies available which allow a more complicated way of communicating. One of them is the Serial Line Internet Protocol or SLIP in short. The protocol encapsulates IP packets in order to send TCP/IP traffic over serial line connections [23].

Contiki provides an interface (slip-arch.c) for easy SLIP implementing on new platforms which is similar to the one for serial line handling. This time the configured serial line input interrupt should call the \texttt{slip\_input\_byte()} function. On the other hand, the output redirection is handled a little differently. There are no more toolchain dependent sink functions involved anymore, just one function that is required by Contiki, namely \texttt{slip\_arch\_writeb()}.

- \texttt{void slip\_arch\_init(unsigned long ubr);}  
  Initializes the serial connection for SLIP handling. Enable the serial input interrupt here.

- \texttt{void slip\_arch\_writeb(unsigned char c);}  
  Forwards all outgoing SLIP packets to the serial line connection.

The LoRaMote’s SLIP driver implementation can be found in Appendix A.8. Once again, the USB’s capability of emulating a serial COM port has been used.

3.10 Radio driver

When consulting Contiki’s Wiki pages on how to implement a port for a microcontroller’s radio capabilities, not much useful information will be found. All radio related pages treat higher level topics like RPL, radio duty cycling and MAC protocols. Adam Dunkels’ porting Contiki workshop only reveals a few hints [9] and in the meanwhile Contiki’s radio interface has changed. Even though radio communication is a key feature in the existence of Contiki, the steps that need to be taken to implement a new radio remains highly undocumented. By examining the source of existing radio drivers however, a radio\_driver struct can be found. Each new radio implementation should use this C struct
to define its own radio driver functions. The following list contains the most important functions of the data structure that are required to get a working radio device.

- **int (* init)(void);**  
  Initializes the radio hardware.

- **int (* prepare)(const void *payload, unsigned short payload_len);**  
  Prepares a packet to be sent by the radio.

- **int (* transmit)(unsigned short transmit_len);**  
  Sends the previously prepared packet.

- **int (* send)(const void *payload, unsigned short payload_len);**  
  Prepares & transmits a packet.

- **int (* read)(void *buf, unsigned short buf_len);**  
  Copies a received packet to Contiki’s input buffer.

- **int (* channel_clear)(void);**  
  Performs a Clear Channel Assessment (CCA) to find out if another device is currently transmitting.

- **int (* receiving_packet)(void);**  
  Checks if the radio driver is currently receiving a packet.

- **int (* pending_packet)(void);**  
  Checks if the radio driver has just received a packet.

- **int (* on)(void);**  
  Turns the radio on.

- **int (* off)(void);**  
  Turns the radio off (or go to into Low Power Mode).

Next to these interface functions provided by Contiki, some other interrupt functions should be implemented before a working radio driver can be achieved. These interrupt functions are completely platform dependent, but will usually include callbacks for radio events like received packet, transmitted packet and maybe error handling functions.

Since this porting project reuses parts of the LoRaMac system, the implementation of the LoRaMote’s radio driver for Contiki mainly consists of forwarding all data between Contiki and the SX1272 radio driver functions. The precise implementation is presented in Appendix A.9.

### 3.11 Low power mode

Since lots of Internet of Things and wireless sensor network applications wait for specific events to occur, the processor keeps itself busy most of the time by polling all processes to know if there are new instructions to execute. A significant amount of power can be saved by putting the microcontroller in a low power mode (LPM). In such a low power mode
several peripherals of the board are deactivated, for example the main processor. By using
the low power mode capabilities of the board, its lifetime can be extended greatly since it
is generally powered by a sole battery. Usually a board has several low power modes. The
datasheets should then be checked to select the most suited one. When there are no more
processes to run, the microcontroller should switch to the chosen low power mode. The
microcontroller exits the low power mode by powering up all needed peripherals once the
clock module or the rtimer detects that a process is pending. Depending on the selected
low power mode it is possible that the hardware clocks on which Contiki’s clock module
and rtimer are based, are deactivated as well. In this case both timers just stop ticking
which will break the application’s normal execution. This is the main reason why the clock
module and rtimer should be using an external clock that is not deactivated by the selected
low power mode.

The STM32L1, LoRaMote’s microcontroller, features several low power modes. After
analyzing the differences between them in the STM32L1 datasheet [24], the STOP mode
was chosen. It offers a fast wake up time while still greatly reducing the power consumption.
The STOP mode also deactivates the main processor. Both the clock module and the rtimer
use it as their clock source which means that their functionality will render useless once
the low power mode is entered. Therefore these modules need to be implemented with an
external clock. The STM32L1 real-time clock (RTC) is used for this purpose since it can
be sourced from the low speed external oscillator (LSE). The RTC structure initialization
is shown in Appendix A.10. The functions to control the low power mode can be found in
Appendix A.11.

Contiki’s new clock module implementation makes use of the RTC Wake Up interrupt.
This interrupt is fired a couple of times per second to fulfill the clock module’s purpose.
The amount of ticks per second stays configurable by changing the CLOCK_CONF-_SECOND constant in the platform configuration file. The new code for the clock module
with LPM support is set out in Appendix A.12.

In the previous implementation of the rtimer, event scheduling was done in a similar way
as in the clock module: by generating interrupts many times per second. By using the
RTC Alarm structure, it is possible to schedule a callback time on the hardware itself. The
interrupt will only be fired once the process’ rtimer expires. As a result, the amount of
interrupts is greatly reduced. The full implementation of the rtimer with LPM support can
be found in Appendix A.13.

Lastly, again two extra constants are added to the platform configuration file.

```
1 /* The frequency of the LSE clock */
2 #define F_LSE 32768
3 /* Time in microseconds to wake up from low power mode */
4 #define MCU_WAKE_UP_TIME 3400
```
4 | Main function

Contiki’s main function implementation (contiki-{PLATFORM_NAME}-main.c) is rarely discussed in porting guides, although when not configured correctly it can be an unsuspected source of weird bugs. It loads the required Contiki components, initializes the platform’s hardware and configures its network capabilities. Then it will start the user processes and schedule them accordingly. Some caution is required in the order of initialization. Both the etimer and sensor modules are implemented as Contiki processes, therefore they need to be initialized after the Contiki process initialization has been completed. Since ctimers are implemented as etimers, its initialization should happen after the etimer process has started. The core of this main function is the process scheduler loop. It is an infinite loop that will schedule the started processes according their cooperative context [11, Section Processes].

```c
// Initialize here the hardware, Contiki components & network stack
/* Start the process scheduler loop */
while(1) {
    int r;
    do {
        r = process_run();
    } while(r > 0);
    // Enter low power mode here
}
```

The implementation of the LoRaMote main file for Contiki is demonstrated in Appendix A.14. Its process scheduler loop is a bit more extensive in order to integrate the low power mode and the proper execution of the standard I/O redirection.
Now that all components are covered, they have to be programmed on the platform somehow. The idea consists of compiling the whole project and generating a single binary file. This binary file then needs to be programmed on the microcontroller. There are two main techniques that can be used. Either the application gets flash programmed to the microcontroller using a hardware programmer or a bootloader that will put the device in a firmware update mode (DFU) is installed so applications can be uploaded through a USB connection.

Such hardware programmer often comes with debug and I/O redirection capabilities and is thus indeed a great tool to use during the early stages of the porting project. A downside of this method is that applications which use the I/O redirection sink framework will not work without the hardware debugger. Since the system will not know how to handle these I/O calls as a consequence of the missing sink framework, the whole system will halt once the hardware debugger is no longer used. In order to get these applications running without such debugger, the serial line driver discussed in Section 3.6 must be implemented.

When the hassle of hardware programmers is not desired, a bootloader can be used. This bootloader initially must be installed by the hardware programmer though. Once installed correctly, it is possible to load new applications to the platform by the means of a USB connection. Note that applications launched from bootloaders can require different build options than those which are not launched from a bootloader. As long as the serial line driver is not set up, the microcontroller will still require the hardware debugger in order to handle any I/O calls. Once both the bootloader and serial line driver are correctly set up, the platform can be programmed and interacted with by just a USB connection.

To program Contiki onto the LoRaMote board, the ST-Link in-circuit debugger/programmer [25] is used. Its debug and I/O redirection capabilities are empowered by the usage of OpenOCD [26]. This I/O redirection mechanism is called semihosting on ARM targets [27, page 8-2]. The LoRaMac project [13] includes a bootloader to be used for this platform and is placed in the /tools/loramote directory. In order to open DFU capable devices and upload new firmware over USB, the dfu-util tool [28] is used. Since these three tools are usable by all STM32 devices, a new folder is created namely /tools/stm32. Each tool was downloaded and built to this directory. Further usage of these tools will be discussed in the next section when the required makefiles are explained. The custom OpenOCD configuration file for the LoRaMote platform can be found in Appendix A.15.
The last step in creating a successful port of Contiki is being able to set up the makefiles. These files contain the instructions to compile all source files, link them into a binary and upload it to the microcontroller. As mentioned before, several makefiles are needed to create a working port. Contiki’s top-level makefile takes care of its core components. Hence a makefile for the microcontroller drivers is required alongside a separate makefile for the platform dependent code. When building an application for Contiki, the desired target platform is indicated by defining the TARGET environment variable. Contiki will automatically include the chosen platform’s makefile. In this makefile there should be a reference to the makefile of the platform’s microcontroller which makes the makefile chain complete.

The objective of the platform makefile is to specify the target’s source files and the directories where they reside. These must respectively be defined in the CONTIKI_-TARGET_SOURCEFILES and CONTIKI_TARGET_DIRS variables [9]. As said before, there must be an inclusion of the CPU’s makefile defined. Furthermore, this file should contain the instructions to detect connected devices, to program binaries using the hardware programmer or bootloader and to connect to the device’s serial output. If the platform needs specific Contiki components or compiler flags (CFLAGS), these also should be defined here.

The microcontroller’s architecture source files, like startup files, C library and CPU drivers must be listed in the CPU makefile. It is also here that the support for one or more compiler toolchains and their linker scripts should be implemented. For a successful build, each toolchain requires its own set of compiler and linker flags (LDFLAGS). Contiki features a default set of makefile rules to compile and link all source files. If for some reason these makefiles rules do not suffice, they can be redefined here.

The example makefiles for the LoRaMote platform can be found in Appendix A.16 and A.17.
Bibliography


Bibliography


[22] IMST, iM880A Data Sheet, July 2013.


Appendix A

Porting Contiki to LoRaMote

A.1 Clock module

This clock module should be located at /cpu/arm/stm32l1/clock.c.

```c
#include <stdio.h>
#include "contiki.h"
#include "stm32l1xx_conf.h"
#include "stm32l1xx_systick.h"

/*---------------------------------------------------------------------------*/
/* After how many clock cycles should the systick interrupt be fired */
#define RELOAD_VALUE ((F_CPU/CLOCK_CONF_SECOND) - 1)
/*---------------------------------------------------------------------------*/
static volatile unsigned long seconds;
static volatile clock_time_t ticks;

/*---------------------------------------------------------------------------*/
/* This interrupt function will increase the tick counter */
void SysTick_Handler(void)
{
    ENERGEST_ON(ENERGEST_TYPE_IRQ);
    ticks++;
    if((ticks % CLOCK_SECOND) == 0) {
        seconds++;
        energest_flush();
    }
    /* If an etimer expired, continue its process */
    if(etimer_pending()) {
        etimer_request_poll();
    }
    ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}

/* Select the main processor as systick clock source */
SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);
/* Set the reload value */
SysTick_SetReload(RELOAD_VALUE);
/* Enable the systick interrupt */
SysTick_ITConfig(ENABLE);
```
Appendix A. Porting Contiki to LoRaMote

```c
/* Enable the systick timer */
SysTick_CounterCmd(SysTick_Counter_Enable);

unsigned long clock_seconds(void)
{
  return seconds;
}

void clock_set_seconds(unsigned long sec)
{
  seconds = sec;
}

clock_time_t clock_time(void)
{
  return ticks;
}

void clock_delay(unsigned int i)
{
  for(; i > 0; i--) {
    __NOP();
  }
}

void clock_wait(clock_time_t i)
{
  clock_time_t start;
  start = clock_time();
  while(clock_time() - start < i);
}

A.2 Rtimer

This rtimer module should be located at /cpu/arm/stm32l1/rtimer-arch.c.

#include "rtimer-arch.h"

#ifndef RTIMER_PRESCALER
#define RTIMER_PRESCALER (F_CPU / (RTIMER_SECOND*2))
#endif

static volatile rtimer_clock_t rtimer_clock;
static volatile rtimer_clock_t timeout_value;

void TIM2_IRQHandler(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);
  rtimer_clock++;
  TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
  if(rtimer_clock==timeout_value){
    rtimer_run_next();
  }
```
Appendix A. Porting Contiki to LoRaMote

A.3 Watchdog

The watchdog implementation should be located at /cpu/arm/stm32l1/watchdog.c.

```c
#include "dev/watchdog.h"
#include "stm32l1xx_conf.h"
#include "contiki-conf.h"

static uint8_t counterValue;

void watchdog_init(void)
{
  /* Get the LSI frequency: 37kHz according to stm32l1x datasheet,
   * but can be measured with a high speed oscillator for greater precision */
  ...
}
```
Appendix A. Porting Contiki to LoRaMote

```c
uint32_t LsiFreq = F_LSI;

/* Enable write access to IWDG_PR and IWDG_RLR registers */
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);

/* IWDG counter clock: LSI/32 */
IWDG_SetPrescaler(IWDG_Prescaler_32);

/* Set counter reload value to obtain 250ms IWDG TimeOut. 
   (the timeout may varies due to LSI frequency dispersion) 
   Counter Reload Value = IWDG counter clock period/250ms 
   = (LSI/32) / 250ms 
   = (LsiFreq/32) / 0.25s 
   = LsiFreq / (32 * 4) 
   = LsiFreq / 128 */
IWDG_SetReload(LsiFreq/(32 * (1000/WATCHDOG_IWDG_TIMEOUT)));

#else
/* Enable WWDG clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);

/* WWDG clock counter = (PCLK1 (32MHz)/4096)/8 = 977 Hz (~1024 us) */
WWDG_SetPrescaler(WWDG_Prescaler_8);

/* Set Window value to 127; WWDG counter should be refreshed only when the counter 
is below 127 (and greater than 64) otherwise a reset will be generated */
WWDG_SetWindowValue(127);

/* Set counter value to 127; WWDG timeout = ~1024 us * 64 = 65.53 ms 
In this case the refresh window is: 
~1024us * (127-127) = 0 ms < refresh window < ~1024us * 64 = 65.53ms */
counterValue = 127;
#endif
#endif

void watchdog_start(void)
{
#if ENABLE_WATCHDOG
/* We setup the watchdog to reset the device after a specific time, 
   unless watchdog_periodic() is called */
#if WATCHDOG_USE_IWDG
IWDG_ReloadCounter();
IWDG_Enable();
#else
WWDG_Enable(counterValue);
#endif
#endif
}

void watchdog_periodic(void)
{
/* This function is called periodically to restart the watchdog timer */
#if WATCHDOG_USE_IWDG
IWDG_ReloadCounter();
#else
WWDG_SetCounter(counterValue);
#endif
}

void watchdog_stop(void)
{
/* Impossible to stop watchdogs once started */
}

void watchdog_reboot(void)
{
  watchdog_stop();
  watchdog_init();
  watchdog_start();
}
```
 Appendix A. Porting Contiki to LoRaMote

A.4 LEDs

This LED module should be located at /platform/loramote/dev/leds-arch.c.

```c
#include "dev/leds.h"
#include "lora-contiki-interface.h"

void leds_arch_init(void)
{
    /* Initialize LED structures */
    GpioInit(&Led1, LED_1, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
    GpioInit(&Led2, LED_2, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
    GpioInit(&Led3, LED_3, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
}

unsigned char leds_arch_get(void)
{
    /* Create LED bitmask using bitwise OR'ing the separate values */
    return (GpioRead(&Led1) ? LEDS_RED : 0)
           | (GpioRead(&Led2) ? LEDS_GREEN : 0)
           | (GpioRead(&Led3) ? LEDS_YELLOW : 0);
}

void leds_arch_set(unsigned char leds)
{
    /* Test the LED bitmask and set the LEDs accordingly */
    GpioWrite(&Led1, (leds & LEDS_RED) ? 0 : 1);
    GpioWrite(&Led2, (leds & LEDS_GREEN) ? 0 : 1);
    GpioWrite(&Led3, (leds & LEDS_YELLOW) ? 0 : 1);
}
```

A.5 Serial line driver

This I/O redirection module should be located at /cpu/arm/stm32l1/syscalls.c.

```c
#include <stdio.h>
#include <errno.h>
#include "contiki.h"

/* Register name faking - works in collusion with the linker */
register char *stack_ptr asm ("sp");
extern int errno;
extern int __io_putchar(int ch);
extern int __io_putstring(const unsigned char *buffer, int size);

#if defined(REDIRECT_STDIO)

/* Write a character to a file. 'libc' subroutines will use this system routine for output to all files, including stdout. Returns number of bytes sent */
size_t _write(int handle, const unsigned char *buffer, size_t size)
{
    return __io_putchar(*buffer);
}
#endif
```
This I/O redirection module should be located at /platform/loramote/dev/serial-line-arch.c.
Appendix A. Porting Contiki to LoRaMote

A.6 Button sensor

This button sensor implementation should be located at /platform/loramote/dev/button-sensor.c.

```c
#include "lib/sensors.h"
#include "dev/button-sensor.h"
#include "lora-contiki-interface.h"

static int _initialized = 0;
static int _active = 0;
extern Gpio_t NIrqSx9500;
extern Gpio_t TxEnSx9500;

void cascaded_button_interrupt(void)
{
    uint8_t statusSX1509 = 0;
    uint8_t statusSX9500 = 0;
    SX1509Read(RegInterruptSourceB, &statusSX1509);
    SX9500Read(SX9500_REG_IRQSRC, &statusSX9500);
    if(((statusSX1509 & 0x01) == 0x01) && ((statusSX9500 & 0x40) == 0x40)){
        sensors_changed(&button_sensor);
    }
    /* Check if interrupt is generated by SX1509 IOE8 pin & SX9500 close proximity sensor */
    if((statusSX1509 & 0x01) == 0x01) && ((statusSX9500 & 0x40) == 0x40){
        sensors_changed(&button_sensor);
    }
    /* Clear NINT interrupt */
    /* (NIRQ interrupt is automatically cleared by reading interrupt source register) */
    SX1509Read(RegInterruptSourceB, &statusSX1509);
    SX9500Read(SX9500_REG_IRQSRC, &statusSX9500);
    ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}

static void init(void)
{
    uint8_t status = 0;
    /* Initialize the SX9500 proximity sensor */
```
Appendix A. Porting Contiki to LoRaMote

GpioInit(&NIRQ_Sx9500, N_IRQ_SX9500, PIN_INPUT, PIN_PUSH_PULL, PIN_NO_PULL, 1);
GpioInit(&TxEn_Sx9500, TX_EN_SX9500, PIN_OUTPUT, PIN_PUSH_PULL, PIN_NO_PULL, 1);

SX9500Init();
SX9500Write(SX9500_REG_IRQ_MSK, 0x10);
SX9500Write(SX9500_REG_IRQ_SRC, 0x10);

do {
   SX9500Read(SX9500_REG_IRQ_SRC, &status);
} while((status & 0x10) == 0x00); /* While compensation for CS0 is pending */

/* Enable SX9500 close proximity interrupt (NIRQ) */
SX9500Write(SX9500_REG_IRQ_MSK, 0x40);
SX9500Write(SX9500_REG_IRQ_SRC, 0x00);
/* Increase proximity detection threshold */
SX9500Write(SX9500_REG_PROX_CTRL6, 0x1F);

/* Enable the SX1509 IO Expander interrupt (NINT) */
SX1509Write(RegInterruptMaskB, 0xFE);
/* Set edge detection to falling */
SX1509Write(RegSenseLowB, 0x02);
SX1509Read(RegPullUpB, &status);
SX1509Write(RegPullUpB, (status | 0x01));

/* Initialize WKUP1/EXTI0 interrupt */
Gpio_t Irq_SX1509;
GpioInit(&Irq_SX1509, WKUP1, PIN_INPUT, PIN_PUSH_PULL, PIN_NO_PULL, 0);
GpioSetInterrupt(&Irq_SX1509, IRQ_FALLING_EDGE, IRQ_VERY_LOW_PRIORITY, &cascaded_button_interrupt);

_initialized = 1;
_active = 1;
}

static void activate(void)
{
   if(!_initialized){
      init();
   }
   _active = 1;
}

static void deactivate(void)
{
   _active = 0;
}

static int active(void)
{
   return _active;
}

static int value(int type)
{
   uint8_t regValue = 0;
   uint16_t offset = 0;
   /* Read 1st sensor offset */
   SX9500Read(SX9500_REG_OFFSET_MSB, (uint8_t*)&regValue);
   offset = regValue << 8;
   SX9500Read(SX9500_REG_OFFSET_LSB, (uint8_t*)&regValue);
   offset |= regValue;
   return (offset > 2000);
}

static int configure(int type, int value)
{
A.7 Node MAC address & node ID

This node MAC and ID restorer functions should be located at /cpu/arm/stm32l1/node-id.c.

```c
#include <string.h>
#include "sys/node-id.h"
#include "contiki-conf.h"

#define DEVICE_ID_REG0 (*((volatile uint32_t *)0x1FF80050))
#define DEVICE_ID_REG1 (*((volatile uint32_t *)0x1FF80054))
#define DEVICE_ID_REG2 (*((volatile uint32_t *)0x1FF80064))

unsigned short node_id = 0;
unsigned char node_mac[8];
volatile uint32_t device_id[3];

void node_id_restore(void)
{
    device_id[0] = DEVICE_ID_REG0;
    device_id[1] = DEVICE_ID_REG1;
    device_id[2] = DEVICE_ID_REG2;

    *((uint32_t *)node_mac) = DEVICE_ID_REG1;
    *((((uint32_t *)node_mac) + 1)) = DEVICE_ID_REG2 + DEVICE_ID_REG0;
    node_id = (unsigned short)DEVICE_ID_REG2;
}

void node_id_burn(unsigned short id){}
```

---

Appendix A. Porting Contiki to LoRaMote
A.8 SLIP driver

The SLIP driver should be placed at `/platform/loramote/dev/slip-arch.c`.

```c
#include "dev/slip.h"
#include "lora-contiki-interface.h"
extern bool Virtual_ComPort_IsOpen(void);

void slip_arch_input_callback(UartNotifyId_t id){
    watchdog_periodic();
    if(id == UART_NOTIFY_RX){
        uint8_t ch = 0;
        while(UartGetChar(&UartUsb, &ch));
        slip_input_byte(ch);
    }
}
```

```c
#define CLEAR_RXBUF() (lora_radio_rxbuf[0] = 0)
#define IS_RXBUF_EMPTY() (lora_radio_rxbuf[0] == 0)
/* Incoming data buffer, the first byte will contain the length of the packet */
static uint8_t lora_radio_rxbuf[LORA_MAX_PAYLOAD_SIZE + 1];
static RadioEvents_t RadioEvents;
static int packet_is_prepared = 0;
static const void *packet_payload;
static unsigned short packet_payload_len = 0;
static packetbuf_attr_t last_rssi = 0;
```

A.9 Radio driver

The radio driver should be placed at `/platform/loramote/net/lora-radio-arch.c`. Related files like radio configurations also should reside in this directory.

```c
#include "lora-radio-arch.h"
#define DEBUG 0
#if DEBUG
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...) 
#endif
/* Incoming data buffer, the first byte will contain the length of the packet */
static uint8_t lora_radio_rxbuf[LORA_MAX_PAYLOAD_SIZE + 1];
```
static int lora_radio_send(const void *data, unsigned short len);
static int lora_radio_read(void *buf, unsigned short bufsize);
static int lora_radio_channel_clear(void);
static int lora_radio_receiving_packet(void);
static int lora_radio_pending_packet(void);
static int lora_radio_on(void);
static int lora_radio_off(void);
void OnTxDone(void);
void OnRxDone(uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr);
void OnTxTimeout(void);
void OnRxTimeout(void);
void OnRxError(void);

/*---------------------------------------------------------------------------*/
PROCESS(lora_radio_process, "LoRa radio driver process");
/*---------------------------------------------------------------------------*/
const struct radio_driver lora_radio_driver =
{
    lora_radio_init,
    lora_radio_prepare,
    lora_radio_transmit,
    lora_radio_read,
    lora_radio_channel_clear,
    lora_radio_receiving_packet,
    lora_radio_pending_packet,
    lora_radio_on,
    lora_radio_off,
};

/*---------------------------------------------------------------------------*/
static int lora_radio_init(void)
{
    PRINTF("RADIO INIT IN\n");
SpiInit(&SX1272.Spi, RADIO_MOSI, RADIO_MISO, RADIO_SCLK, NC);
SX1272IoInit();

/* Radio initialization */
RadioEvents.TxDone = OnTxDone;
RadioEvents.RxDone = OnRxDone;
RadioEvents.TxTimeout = OnTxTimeout;
RadioEvents.RxTimeout = OnRxTimeout;
RadioEvents.RxError = OnRxError;

Radio.Init(&RadioEvents);
Radio.SetChannel(RF_FREQUENCY);

#if defined(USE_MODEM_LORA)
Radio.SetTxConfig(MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
LORA_FREQUENCY_HOPPING_ON, LORA_HOPPING_PERIOD,
LORA_IQ_INVERSION_ON, TX_TIMEOUT_VALUE);
Radio.SetRxConfig(MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR, 0, LORA_PREAMBLE_LENGTH,
LORA_SYMBOL_TIMEOUT, 0, LORA_FIXED_PAYLOAD_LENGTH, LORA_CRC_ON,
LORA_FREQUENCY_HOPPING_ON, LORA_HOPPING_PERIOD,
LORA_IQ_INVERSION_ON, RX_CONTINUOUS_MODE);
#elif defined(USE_MODEM_FSK)
Radio.SetTxConfig(MODEM_FSK, TX_OUTPUT_POWER, FSK_FDEV, 0,
FSK_DATARATE, 0, FSK_PREAMBLE_LENGTH, FSK_FIX_LENGTH_PAYLOAD_ON,
FSK_CRC_ON, 0, 0, 0, TX_TIMEOUT_VALUE);
Radio.SetRxConfig(MODEM_FSK, FSK_BANDWIDTH, FSK_DATARATE, 0,
FSK_AFC_BANDWIDTH, FSK_PREAMBLE_LENGTH,
0, FSK_FIX_LENGTH_PAYLOAD_ON, FSK_FIXED_PAYLOAD_LENGTH,
FSK_CRC_ON, 0, 0, 0, RX_CONTINUOUS_MODE);
#else
#error "Please define a modem in the compiler options."
#endif
Appendix A. Porting Contiki to LoRaMote

```c
#ifdef
        process_start(&lora_radio_process, NULL);
        PRINTF("RADIO INIT OUT\n");
        return 0;
    }
#endif

static int lora_radio_prepare(const void *payload, unsigned short payload_len)
{
    PRINTF("PREPARE IN: %u bytes\n", payload_len);
    packet_is_prepared = 0;

    /* Checks if the payload length is supported */
    if(payload_len > LORA_MAX_PAYLOAD_SIZE) {
        return RADIO_TX_ERR;
    }

    packet_payload = payload;
    packet_payload_len = payload_len;
    packet_is_prepared = 1;

    PRINTF("PREPARE OUT\n");
    return RADIO_TX_OK;
}

static int lora_radio_transmit(unsigned short payload_len)
{
    PRINTF("TRANSMIT IN\n");

    if(!packet_is_prepared) {
        return RADIO_TX_ERR;
    }

    Radio.Send((uint8_t *)packet_payload, packet_payload_len);
    packet_is_prepared = 0;

    PRINTF("TRANSMIT OUT\n");
    return RADIO_TX_OK;
}

static int lora_radio_send(const void *payload, unsigned short payload_len)
{
    if(lora_radio_prepare(payload, payload_len) == RADIO_TX_ERR) {
        return RADIO_TX_ERR;
    }
    return lora_radio_transmit(payload_len);
}

static int lora_radio_read(void *buf, unsigned short bufsize)
{
    PRINTF("READ IN\n");

    /* Checks if the RX buffer is empty */
    if(IS_RXBUF_EMPTY()) {
        PRINTF("READ OUT: RX BUFFER EMPTY\n");
        return 0;
    }

    /* Checks if buffer has the correct size */
    if(bufsize < lora_radio_rxbuf[0]) {
        PRINTF("READ OUT: TOO SMALL BUFFER\n");
        return 0;
    }

    /* Copies the packet received */
    memcpy(buf, lora_radio_rxbuf+1, lora_radio_rxbuf[0]);
    packetbuf_set_attr(PACKETBUF_ATTR_RSSI, last_rssi);
    bufsize = lora_radio_rxbuf[0];
```
CLEAR_RXBUF();
PRINTF("READ OUT\n");
return bufsize;
}
/*-----------------------------*/
static int lora_radio_channel_clear(void)
{
PRINTF("CHANNEL CLEAR IN\n");
bool channel_clear;
#if defined(USE_MODEM_LORA)
channel_clear = Radio.IsChannelFree(MODEM_LORA, RF_FREQUENCY, CCA_THRESHOLD);
#elif defined(USE_MODEM_FSK)
channel_clear = Radio.IsChannelFree(MODEM_FSK, RF_FREQUENCY, CCA_THRESHOLD);
#else
#error "Please define a modem in the compiler options."
#endif
PRINTF("CHANNEL CLEAR OUT\n");
return channel_clear;
} /*-----------------------------*/
static int lora_radio_receiving_packet(void)
{
return 0;
} /*-----------------------------*/
static int lora_radio_pending_packet(void)
{
PRINTF("PENDING PACKET\n");
if(!IS_RXBUF_EMPTY()) {
process_poll(&lora_radio_process);
}
} /*-----------------------------*/
static int lora_radio_off(void)
{
Radio.Sleep();
PRINTF("RADIO OFF\n");
return 0;
} /*-----------------------------*/
static int lora_radio_on(void)
{
Radio.Rx(RX_TIMEOUT_VALUE);
PRINTF("RADIO ON\n");
return 0;
} /*-----------------------------*/
PROCESS_THREAD(lora_radio_process, ev, data)
{
PROCESS_BEGIN();
int len;
while(1) {
PROCESS_YIELD_UNTIL(ev == PROCESS_EVENT_POLL);
PRINTF("LoRa radio: polled\n");
packetbuf_clear();
len = lora_radio_read(packetbuf_dataptr(), PACKETBUF_SIZE);
if(len > 0) {
packetbuf_set_datalen(len);
NETSTACK_RDC.input();
}
if(!IS_RXBUF_EMPTY()) {
process_poll(&lora_radio_process);
}
Appendix A. Porting Contiki to LoRaMote

A.10 External clock

This external clock implementation should be located at /cpu/arm/stm32l1/rtc-arch.c.

```c
#include "rtc-arch.h"

/* The prescaler assumes the RTC timer is sourced with the LSE
   and uses the minimum asynchronous division factor (2) */
#define RTC_PRESCALER ((F_LSE/2) / RTIMER_SECOND)

static int rtcInitialized = 0;

void init_rtc(void)
{
    if(rtcInitialized)
        return;

    /* Initialize the RTC */
    RTC_InitTypeDef RTC_InitStructure;
    RTC_TimeTypeDef RTC_TimeStruct;
    RTC_DateTypeDef RTC_DateStruct;
```
/* Enable the PWR clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);

/* Allow access to RTC */
PWR_RTCAccessCmd(ENABLE);

/* Reset RTC Domain */
RCC_RTCResetCmd(ENABLE);
RCC_RTCResetCmd(DISABLE);

/* Enable the LSE OSC */
RCC_LSEConfig(RCC_LSE_ON);

/* Wait till LSE is ready */
while(RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET){}

/* Select the RTC Clock Source */
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

/* Enable the RTC Clock */
RCC_RTCCLKCmd(ENABLE);
RTC_TimeStructInit(&RTC_TimeStruct);
RTC_DateStructInit(&RTC_DateStruct);
RTC_SetTime(RTC_Format_BIN, &RTC_TimeStruct);
RTC_SetDate(RTC_Format_BIN, &RTC_DateStruct);

/* Wait for RTC APB registers synchronisation */
RTC_WaitForSynchro();

/* Configure the RTC data register and RTC prescaler */
RTC_Init(&RTC_InitStructure);

/* Wait for RTC APB registers synchronisation */
RTC_WaitForSynchro();

rtcInitialized = 1;

/*---------------------------------------------------------------------------*/

A.11 Low power controller

The low power control functions should be located at /cpu/arm/stm32l1/lpm-arch.c.

#include "lpm-arch.h"

static int stopModeActivated = 0;

void lpm_enter_stopmode(void)
{
    stopModeActivated = 1;
    /* Disable the Power Voltage Detector */
PWR_PVDCmd(DISABLE);

    /* Set MCU in ULP (Ultra Low Power) */
PWR.UltraLowPowerCmd(ENABLE);

    /* Disable fast wakeUp */
PWR_FastWakeUpCmd(DISABLE);
Appendix A. Porting Contiki to LoRaMote

/* Enter Stop Mode */
PWR_EnterSTOPMode(PWR_Regulator_LowPower, PWR_STOPEntry_WFI);
} /*-----------------------------------------------------*/

void lpm_exit_stopmode(void)
{
  if(!stopModeActivated)
    return;

  /* Disable IRQ while the MCU is not running on HSE */
  __disable_irq();

  /* Enable HSE */
  RCC_HSEConfig(RCC_HSE_ON);

  /* Wait till HSE is ready */
  while (RCC_GetFlagStatus(RCC_FLAG_HSERDY) == RESET){}

  /* Enable PLL */
  RCC_PLLCmd(ENABLE);

  /* Wait till PLL is ready */
  while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET){}

  /* Select PLL as system clock source */
  RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

  /* Wait till PLL is used as system clock source */
  while (RCC_GetSYSCLKSource() != 0x0C){}

  /* Set MCU in ULP (Ultra Low Power) */
  PWR_UltraLowPowerCmd(DISABLE); // add up to 3ms wakeup time

  /* Enable the Power Voltage Detector */
  PWR_PVDCmd(ENABLE);
  __enable_irq();

  stopModeActivated = 0;
} /*-----------------------------------------------------*/

A.12 Low power clock module

The low power clock module replaces the /cpu/arm/stm32l1/clock.c file.

#include "contiki.h"
#include "stm32l1xx_conf.h"
#include "rtc-arch.h"
#include "lpm-arch.h"

/*-----------------------------------------------------*/
#define DEBUG 0
#if DEBUG
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...) 
#endif

/*-----------------------------------------------------*/

static volatile unsigned long seconds;
static volatile clock_time_t ticks;
Appendix A. Porting Contiki to LoRaMote

```c
void RTC_WKUP_IRQHandler(void)
{
  ENERGEST_ON(ENERGEST_TYPE_IRQ);
  /* Check on the WakeUp flag */
  if(RTC_GetITStatus(RTC_IT_WUT) != RESET)
  {
    ticks++;
    if((ticks % CLOCK_SECOND) == 0){
      seconds++;
      energest_flush();
      PRINTF("second %i (%i ticks)\n", seconds, ticks);
    }
  }
  /* If an etimer expired, continue its process */
  if(etimer_pending()){
    lpm_exit_stopmode();
    etimer_request_poll();
  }
  /* Clear RTC Wakeup flags */
  RTC_ClearITPendingBit(RTC_IT_WUT);
}

void clock_init(void)
{
  seconds = 0;
  ticks = 0;
  /* Initialize the RTC clock */
  init_rtc();
  /* Initialize the RTC Wakeup interrupt */
  EXTI_InitTypeDef EXTI_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;
  /* EXTI configuration */
  EXTI_ClearITPendingBit(EXTI_Line20);
  EXTI_InitStructure.EXTI_Line = EXTI_Line20;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);
  /* Enable the RTC Wakeup Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = RTC_WKUP_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
  /* RTC Wakeup Interrupt Generation: Clock Source: RTCDiv_16, Wakeup Time Base: 7.8ms at 128Hz */
  RTC_WakeUpClockConfig(RTC_WakeUpClock_RTCCCLK_Div16);
  RTC_SetWakeUpCounter(RTC_WKUPCOUNTER-1);
  /* Enable the Wakeup Interrupt */
  RTC_ITConfig(RTC_IT_WUT, ENABLE);
  /* Enable Wakeup Counter */
  RTC_WakeUpCmd(ENABLE);
}```
Appendix A. Porting Contiki to LoRaMote

A.13 Low power rtimer

The low power rtimer replaces the /cpu/arm/stm32l1/rtimer-arch.c file.

```c
#include "rtimer-arch.h"
#define MCU_WAKE_UP_TIME_TICKS (MCU_WAKE_UP_TIME / (1000000/RTIMER_SECOND))

static const uint8_t SecondsInMinute = 60;
static const uint16_t SecondsInHour = 3600;
static const uint32_t SecondsInDay = 86400;
static const uint8_t HoursInDay = 24;
static const uint16_t DaysInYear = 365;
static const uint16_t DaysInLeapYear = 366;
static const double DaysInCentury = 36524.219;
static const uint8_t DaysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
static const uint8_t DaysInMonthLeapYear[] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
static uint8_t PreviousYear = 0;
static uint8_t Century = 0;

void RTC_Alarm_IRQHandler(void)
{
ENERGEST_ON(ENERGEST_TYPE_IRQ);
/* Check on the AlarmA flag */
if(RTC_GetITStatus(RTC_IT_ALRA) != RESET)
{
```

JELLE AERTS
37
Appendix A. Porting Contiki to LoRaMote

```c
lpm_exit_stopmode();
rtimer_run_next();

/* Clear RTC AlarmA Flags */
RTC_ClearITPendingBit(RTC_IT_ALRA);
}

/* Clear the EXTI line 17 */
EXTI_ClearITPendingBit(EXTI_Line17);

ENERGEST_OFF(ENERGEST_TYPE_IRQ);
}

/* Initialize the RTC clock */
init_rtc();

/* Initialize the RTC Alarm interrupt */
EXTI_InitTypeDef EXTI_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;

/* EXTI configuration */
EXTI_ClearITPendingBit(EXTI_Line17);
EXTI_InitStructure.EXTI_Line = EXTI_Line17;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);

/* Enable the RTC Alarm Interrupt */
NVIC_InitStructure.NVIC_IRQChannel = RTC_Alarm_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

/* Disable AlarmA interrupt */
RTC_ITConfig(RTC_IT_ALRA, ENABLE);

/* Disable the AlarmA */
RTC_AlarmCmd(RTC_Alarm_A, DISABLE);

rtimer_clock_t rtimer_arch_now(void)
{
  rtimer_clock_t calendarValue = 0;
  uint8_t i = 0;
  RTC_TimeTypeDef RTC_TimeStruct;
  RTC_DateTypeDef RTC_DateStruct;
  if((PreviousYear == 99) && (RTC_DateStruct.RTC_Year == 0)){
    Century++;  
    PreviousYear = RTC_DateStruct.RTC_Year;
  }
  for(i = 0; i < Century; i++){
    calendarValue += (rtimer_clock_t)(DaysInCentury * SecondsInDay);
  }
  for(i = 0; i < RTC_DateStruct.RTC_Year; i++){
    if((i == 0) || (i % 4 == 0)){
      calendarValue += DaysInLeapYear * SecondsInDay;
    }
  }
  return calendarValue;
}
```
Appendix A. Porting Contiki to LoRaMote

```c
} else{
    calendarValue += DaysInYear * SecondsInDay;
}

/* Months */
if((RTC_DateStruct.RTC_Year == 0) || (RTC_DateStruct.RTC_Year % 4 == 0)){
    for(i = 0; i < (RTC_DateStruct.RTC_Month - 1); i++){
        calendarValue += DaysInMonthLeapYear[i] * SecondsInDay;
    }
} else{
    for(i = 0; i < (RTC_DateStruct.RTC_Month - 1); i++){
        calendarValue += DaysInMonth[i] * SecondsInDay;
    }
}

/* Days */
calendarValue += ((uint32_t)RTC_TimeStruct.RTC_Seconds +
    ((uint32_t)RTC_TimeStruct.RTC_Minutes * SecondsInMinute) +
    ((uint32_t)RTC_TimeStruct.RTC_Hours * SecondsInHour) +
    ((uint32_t)(RTC_DateStruct.RTC_Date * SecondsInDay)));
return calendarValue;
```

```c
void rtimer_arch_schedule(rtimer_clock_t wakeup_time)
{
    // Clear Previous Alarm
    RTC_ClearFlag(RTC_FLAG_ALRAF);
    RTC_AlarmCmd(RTC_Alarm_A, DISABLE);

    RTC_GetTime(RTC_Format_BIN, &RTC_TimeStruct);
    RTC_GetDate(RTC_Format_BIN, &RTC_DateStruct);
    wakeup_time = wakeup_time - floor(MCU_WAKE_UP_TIME_TICKS + 0.5); // Round ticks
    rtcSeconds = (wakeup_time % SecondsInMinute) + RTC_TimeStruct.RTC_Seconds;
    rtcMinutes = ((wakeup_time/SecondsInMinute) % SecondsInMinute) + RTC_TimeStruct.RTC_Minutes;
    rtcHours = (((wakeup_time/SecondsInMinute) / SecondsInMinute) + rtcMinutes) % SecondsInHour;
    rtcDays = (wakeup_time/SecondsInDay) + RTC_DateStruct.RTC_Date;
    rtcAlarmSeconds = rtcSeconds % SecondsInMinute;
    rtcAlarmMinutes = rtcAlarmSeconds / SecondsInMinute;
    rtcAlarmHours = rtcAlarmMinutes / SecondsInMinute;
    rtcAlarmDays = rtcAlarmHours / SecondsInMinute;
    if((RTC_DateStruct.RTC_Year == 0) || (RTC_DateStruct.RTC_Year % 4 == 0))
    {
        if(rtcAlarmDays > DaysInMonthLeapYear[RTC_DateStruct.RTC_Month-1])
```
Appendix A. Porting Contiki to LoRaMote

```c
rtcAlarmDays = rtcAlarmDays % DaysInMonthLeapYear[RTC_DateStruct.RTC_Month-1];
else
{
    if(rtcAlarmDays > DaysInMonth[RTC_DateStruct.RTC_Month-1])
    {
        rtcAlarmDays = rtcAlarmDays % DaysInMonth[RTC_DateStruct.RTC_Month-1];
    }
}
RTC_AlarmStructure.RTC_AlarmTime.RTC_Seconds = rtcAlarmSeconds;
RTC_AlarmStructure.RTC_AlarmTime.RTC_Minutes = rtcAlarmMinutes;
RTC_AlarmStructure.RTC_AlarmTime.RTC_Hours = rtcAlarmHours;
RTC_AlarmStructure.RTC_DateWeekDay = (uint8_t)rtcAlarmDays;
RTC_AlarmStructure.RTC_DateWeekDaySel = RTC_AlarmDateWeekDaySel_Date;
RTC_AlarmStructure.RTC_AlarmMask = RTC_AlarmMask_None;

/* Enable the AlarmA */
RTC_SetAlarm(RTC_Format_BIN, RTC_Alarm_A, &RTC_AlarmStructure);
RTC_AlarmCmd(RTC_Alarm_A, ENABLE);
```

A.14 Contiki main

This main function file should be located at `/platform/loramote/contiki-loramote-main.c`.

```c
#include "lora-contiki-interface.h"
SENSORS(&button_sensor,
    &radio_sensor,
    &temperature_sensor,
    &altitude_sensor,
    &pressure_sensor,
    &battery_sensor);
extern unsigned char node_mac[8];
static linkaddr_t rime_addr;
static uip_ipaddr_t ipaddr;
static void print_processes(struct process * const processes[]);
static void print_device_config(void);
static void set_rime_addr(void);
extern bool Virtual_ComPort_IsOpen(void);
int main(int argc, char **argv)
{
    /* Initialize hardware */
    BoardInitMcu_Contiki();
    leds_init();
    rtimer_init();
    serial_line_arch_init();
    printf("\nInitializing hardware... Done!\n");
    /* Initialize Contiki */
    printf("Initializing Contiki... "); fflush(stdout);
    clock_init();
    watchdog_init();
    process_init();
    process_start(&etimer_process, NULL);
    ctimer_init();
    serial_line_init();
}
```

JELLE AERTS
process_start(&sensors_process, NULL);
printf("Done!\n");
/* Initialize networking */
printf("Initializing network..."); fflush(stdout);
/* Restore node id if such has been stored in external mem */
#ifdef NODEID
node_id = NODEID;
#else /* NODE_ID */
node_id_restore(); /* also configures node_mac[] */
#endif /* NODE_ID */
set_rime_addr();
random_init(node_id);
netstack_init();
#ifdef UIP_CONF_IPV6
memcpy(&uip_lladdr.addr, node_mac, sizeof(uip_lladdr.addr));
queuebuf_init();
process_start(&tcpip_process, NULL);
uiip_ipaddr_t ipaddr;
uip_ip6addr(&ipaddr, 0xfc00, 0, 0, 0, 0, 0, 0, 0);
uip_ds6_set_addr_iid(&ipaddr, uip_lladdr);
uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
#endif /* UIP_CONF_IPV6 */
printf("Done!\n");
/* Initialize energy estimation */
energest_init();
ENERGEST_ON(ENERGEST_TYPE_CPU);
/* Start user processes */
print_device_config();
leds_off(LEDS_ALL);
printf("----------[ Running %s on LoRaMote ]----------\n\n", CONTIKI_VERSION_STRING);
print_processes(autostart_processes);
autostart_start(autostart_processes);
/* Start the process scheduler loop */
watchdog_start();
while(1) {
    int r;
    do {
        watchdog_periodic(); // Reset watchdog
        r = process_run();
    } while(r > 0);
    /* Avoid LPM when a device is connected to serial I/O */
    if(process_nevents() == 0 && !Virtual_ComPort_IsOpen()){
        ENERGEST_OFF(ENERGEST_TYPE_CPU);
        ENERGEST_ON(ENERGEST_TYPE_LPM);
        watchdog_stop();
        lpm_enter_stopmode(); // Enter LPM: Stop mode with RTC
        watchdog_start();
        ENERGEST_OFF(ENERGEST_TYPE_LPM);
        ENERGEST_ON(ENERGEST_TYPE_CPU);
    }
}
return 0;
}/*-----------------------------*/
static void print_processes(struct process * const processes[])
{
    /* const struct process * const * p = processes; */
    print("Starting");
    while(*processes != NULL) {
        printf(" %s", (*processes)->name);
        processes++;
        /*
Appendix A. Porting Contiki to LoRaMote

```c
static void print_device_config(void)
{
    int i;
    uint8_t longaddr[8];
    uint16_t shortaddr;

    printf("Rime started with address ");
    for(i = 0; i < sizeof(rime_addr.u8) - 1; i++) {
        printf("%d.", rime_addr.u8[i]);
    }
    printf("%d\n", rime_addr.u8[i]);

    shortaddr = (linkaddr_node_addr.u8[0] << 8) + linkaddr_node_addr.u8[1];
    memset(longaddr, 0, sizeof(longaddr));
    linkaddr_copy((linkaddr_t *)&longaddr, &linkaddr_node_addr);
    printf("MAC %02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x ",
            longaddr[0], longaddr[1], longaddr[2], longaddr[3],
            longaddr[4], longaddr[5], longaddr[6], longaddr[7]);

    if(node_id) {
        printf("Node id is set to %u\n", node_id);
    } else {
        printf("Node id not set.\n");
    }

    printf("%s, %s, radio frequency %iMHz\n",
            NETSTACK_MAC.name, NETSTACK_RDC.name, RF_FREQUENCY/1000000);

#if UIP_CONF_IPV6
    printf("Tentative link-local IPv6 address ");
    if (!UIP_CONF_IPV6_RPL) {
        printf("Tentative global IPv6 address ");
    }
#endif /* UIP_CONF_IPV6 */

    if(UIP_CONF_IPV6_RPL) {
        printf("Tentative global IPv6 address ");
        for(i = 0; i < 7; ++i) {
            printf("%02x%02x:, ipaddr.u8[i * 2], ipaddr.u8[i * 2 + 1]);
        }
        printf("%02x%02x\n", ipaddr.u8[7 * 2], ipaddr.u8[7 * 2 + 1]);
    }

#if UIP_CONF_IPV6
    memcpy(rime_addr.u8, node_mac, sizeof(rime_addr.u8));
#else
    if(node_id == 0) {
        for(int i = 0; i < sizeof(linkaddr_t); ++i) {
            rime_addr.u8[i] = node_mac[7 - i];
        }
    } else {
        rime_addr.u8[0] = node_id & 0xff;
        rime_addr.u8[1] = node_id >> 8;
    }
#endif /* UIP_CONF_IPV6 */
```

JELLE AERTS
Appendix A. Porting Contiki to LoRaMote

A.15 OpenOCD configuration file

This configuration file is to be placed at /tools/loramote/loramote.cfg.

```c
# This is the OpenOCD config file for a LoRaMote Demo board with a single STM32L151C8 chip
source [find interface/stlink-v2.cfg]
transport select hla_swd
set WORKAREASIZE 0x4000
source [find target/stm32l1.cfg]

# Use hardware reset, connect under reset
reset_config srst_only

# Print stdout to console (GDB commands)
init
arm semihosting enable
```

A.16 CPU Makefile

This make should be located at /cpu/arm/stm32l1/Makefile.stm32l151.

```make
# Makefile for the STM32L151C8 Cortex M3 medium-density microcontroller
.SUFFIXES:

# CPU folder
CONTIKI_CPU = $(CONTIKI)/cpu/arm/stm32l1

# Source folders for Contiki CPU files, ARM CMSIS and STM32L1 libraries
CONTIKI_CPU_DIRS = . \
    ../common/CMSIS \
    $(CONTIKI_MCU_DIRS)

# Source files: proprietary sources for startup. Refer to CMSIS docs.
PROP_SYS_ARCH_C = system_stm32l1xx.c
PROP_SYS_ARCH_S = startup_stm32l1xx_md.s

ifndef IAR
    GCC = 1
endif

# Source files: Contiki arch source files
CONTIKI_CPU_ARCH = \
    clock.c \
    watchdog.c \
    lpm-arch.c \
    rtimer-arch.c \
    rtc-arch.c \
    uart-arch.c

ifdef GCC
```
Appendix A. Porting Contiki to LoRaMote

CONTIKI_CPU_PORT = syscalls.c
STM32_TOOLS ?= $(CONTIKI)/tools/stm32
GCC_BINS ?= $(STM32_TOOLS)/gcc-arm-none-eabi/bin
else
 CONTIKI_CPU_PORT =
endif

UIPDRIVERS = uip-arch.c

# To be implemented
ELFLOADER =

# Source files: STM32L1 libraries
CONTIKI_MCU_DIRS = \n STM32L1xx_StdPeriph_Driver/inc \n STM32L1xx_StdPeriph_Driver/src \n ../../../stm32_common/ \n ../../../stm32_common/STM32_USB-FS-Device_Driver/inc \n ../../../stm32_common/STM32_USB-FS-Device_Driver/src

FULL_MCU_DIRS = ${wildcard $(addprefix $(CONTIKI_CPU)/, $(CONTIKI_MCU_DIRS))}
CONTIKI_MCU_SOURCEFILES = ${foreach d, $(FULL_MCU_DIRS), ${subst ${d}/,,${wildcard $(d)/*.c}}}

# Add CPU folder to search path for .s (assembler) files
ifdef GCC
 vpath %.s $(CONTIKI_CPU)/arm-gcc
else
 vpath %.s $(CONTIKI_CPU)/arm-std
endif

# Include all files above
ssubst = ${patsubst %.s,%.o,${patsubst %.s?9,%.o,$(1)}}
CONTIKI_SOURCEFILES += $(PROP_SYS_ARCH_C) $(CONTIKI_CPU_ARCH) $(CONTIKI_CPU_PORT) $(ELFLOADER) $(UIPDRIVERS) $(CONTIKI_MCU_SOURCEFILES)
PROJECT_OBJECTFILES += $(addprefix $(OBJECTDIR)/,${call ssubst, $(PROP_SYS_ARCH_S)})

# Defines common for IAR and GCC

# Set CPU speed in Hz, NB this might have unexpected side-effects if not at 32
# MHz as it is not immediately clear how specialized the startup code etc is.
# That being said, setting to 24MHz seems to work fine, looking at Contiki clocks
# at least.
F_CPU = 32000000

CFLAGS += \n -DHSE_VALUE=$(F_CPU)ul \n -DUSE_STDPERIPH_DRIVER \n -DSM32L1XX_MD \n -DIAR_ARM_CM3

# IAR

# GCC

### Compiler definitions
GCC = 1
CC = $(GCC_BINS)/arm-none-eabi-gcc
LD = $(GCC_BINS)/arm-none-eabi-ld
SIZE = $(GCC_BINS)/arm-none-eabi-size
AS = $(GCC_BINS)/arm-none-eabi-as
AR = $(GCC_BINS)/arm-none-eabi-ar
NM = $(GCC_BINS)/arm-none-eabi-nm
Appendix A. Porting Contiki to LoRaMote

OBJCOPY = $(GCC_BINS)/arm-none-eabi-objcopy
STRIP = $(GCC_BINS)/arm-none-eabi-strip
GDB = $(GCC_BINS)/arm-none-eabi-gdb
ASFLAGS += -mcpu=cortex-m3 -mthumb

# This platform wields a STM32L151C8 medium-density device
CFLAGS += -DSTM32L1XX_MD=1
CFLAGS += \
  -I. \n  -I$(CONTIKI)/core \n  -I$(CONTIKI_CPU) \n  -I$(CONTIKI)/platform/$(@TARGET) \n  $(addprefix -I,$(APPDIRS)) \n  $(addprefix -I,$(CONTIKI_CPU_DIRS)) \n  -Wall -q -g2 \n  -DWITH_UIP -DWITH_ASCII \n  -mcpu=cortex-m3 \n  -mthumb \n  -mfixed-cortex-m3-ldrd \n  -std=gnu99 \n  -Wno-strict-aliasing \n  -Wno-pointer-sign \n  -Wno-unused-function \n  -Wno-unused-variable \n  -Wno-unused-but-set-variable

LDFLAGS += \
  -L$(CONTIKI_CPU)/arm-gcc \n  -T$(LDSCRIPT) \n  -mcpu=cortex-m3 \n  -mfloat-abi=soft \n  -nostartfiles \n  --specs=rdimon.specs \n  -Wl,-Map=$(OBJECTDIR)/contiki-$(@TARGET).map,--cref $(LDLIBS)

ifeq ($(strip $(REDIRECT_STDIO)),1)
  CFLAGS += -DREDIRECT_STDIO
else #REDIRECT_STDIO
  LDFLAGS += --specs=rdimon.specs
  LDLIBS += -lrdimon
endif #REDIRECT_STDIO

ifeq ($(strip $(SMALL)),1)
  CFLAGS += -Os -ffunction-sections -fdata-sections
  LDFLAGS += -Wl,--gc-sections --specs=nano.specs
  LDFLAGS += -Wl,--undefined=_reset_vector__,--undefined=InterruptVectors,--undefined=_copy_data_init__,--undefined=_clear_bss_init__,--undefined=_end_of_init__
endif #SMALL

# Build rules ------------------------------------------------------------------
CUSTOM_RULE_C_TO_OBJECTDIR_O=yes
CUSTOM_RULE_C_TO_CE=yes
CUSTOM_RULE_C_TO_CO=yes
CUSTOM_RULE_C_TO_O=yes
CUSTOM_RULE_S_TO_OBJECTDIR_O=yes
CUSTOM_RULE_LINK=yes

%.o: %.c
  $(TRACE_CC)
  $(Q)$(CC) $(CFLAGS) -c $< -o $@

%.o: %.s
  $(Q)$(CC) $(CFLAGS) -c $< -o $@

JELLE AERTS
45
Appendix A. Porting Contiki to LoRaMote

```bash
$(TRACE_AS)
$(Q)$(AS) $(ASFLAGS) -c $< -o $@

define FINALIZE_CYGWIN_DEPENDENCY
  sed -e 's/\([A-Z]\):// /cygdrive/\\L\1//' -e 's/\([^ ]\)/\1/g' \
  <$(@:.o=.d) >$(@:.o=.d); \
  rm -f $(@:.o=.P)
endef

$(OBJECTDIR)/%.o: %.c | $(OBJECTDIR)
$(TRACE_CC)
$(Q)$(CC) $(CFLAGS) -c $< -o $@

$(OBJECTDIR)/%.o: %.s | $(OBJECTDIR)
$(TRACE_AS)
$(Q)$(AS) $(ASFLAGS) $< -o $@

%.co: %.c
$(TRACE_CC)
$(Q)$(CC) $(CFLAGS) -c -DAUTOSTART_ENABLE -c $< -o $@

%.ce: %.o
$(TRACE_LD)
$(Q)$(LD) $(LDFLAGS) --relocatable -T $(CONTIKI_CPU)/merge-rodatald $< -o $@ $(LDLIBS)
  $(STRIP) -K _init -K _fini --strip-unneeded -g -x $@

%.co: %.c
$(TRACE_CC)
$(Q)$(CC) $(CFLAGS) -c $< -o $@
$(STRIP) --strip $@

%.o: $(PROJECT_OBJECTFILES) $(PROJECT_LIBRARIES) $(STARTUPFOLDER) # $(OBJECTDIR)/empty-symbols.o
$(TRACE_LD)
$(Q)$(LD) $(LDFLAGS) --relocatable -T $(CONTIKI_CPU)/merge-rodatald $< -o $@ $(LDLIBS)
  $(STRIP) -K _init -K _fini --strip-unneeded -g -x $@

%.co: %.c
$(TRACE_CC)
$(Q)$(CC) $(CFLAGS) -c -DAUTOSTART_ENABLE -c $< -o $@

%.ce: %.o
$(TRACE_LD)
$(Q)$(LD) $(LDFLAGS) $(TARGET_STARTFILES) $(TARGET_LIBFILES) -Wl,-$(CONTIKI_CPU_OBJS) -o $@ $(LDLIBS)
  $(echo >> $(OBJECTDIR)/contiki-$(TARGET).map
  $(Q)$(SIZE) $(SIZEFLAGS) $@ >> $(OBJECTDIR)/contiki-$(TARGET).map
endif IAR

%.$(TARGET): %.o $(PROJECT_OBJECTFILES) contiki-$(TARGET).a $(STARTUPFOLDER) # $(OBJECTDIR)/empty-symbols.o
$(TRACE_LD)
$(Q)$(LD) $(LDFLAGS) -o $@ $(filter-out %.a,$^) $(filter %.a,$^) $(LDLIBS)

$(OBJECTDIR)/%.o: $(PROJECT_OBJECTFILES) $(PROJECT_LIBRARIES) $(TARGET).a $(OBJECTDIR)/empty-symbols.o
$(TRACE_LD)
$(Q)$(LD) $(LDFLAGS) $(filter-out %.a,$^) -Wl,-$(CONTIKI_CPU_OBJS) -o $@ $(LDLIBS)
  $(echo >> $(OBJECTDIR)/contiki-$(TARGET).map
  $(Q)$(SIZE) $(SIZEFLAGS) $@ >> $(OBJECTDIR)/contiki-$(TARGET).map
endif

%.ihex: %.$(TARGET)
  $(Q)$(OBJCOPY) -O ihex $^ $@

%.hex: %.ihex
  @rm $*.hex
  @mv -f $*.ihex $*.hex

%.bin: %.$(TARGET)
  $(Q)$(OBJCOPY) -O binary $^ $@

.PHONY: symbols.c
symbols.c:
  $(Q)cp $(CONTIKI)/tools/empty-symbols.c symbols.c
```

JELLE AERTS

46
Appendix A. Porting Contiki to LoRaMote

\begin{verbatim}
$(Q)cp $(CONTIKI)/tools/empty-symbols.h symbols.h

# Don't use core/loader/elfloader.c, use elfloader-otf.c instead
$(OBJECTDIR)/elfloader.o:
  echo -n >$@

A.17 Platform Makefile

The LoRaMote platform makefile should be located at /platform/loramote/Makefile.

```make
# User settings --------------------------------------------------------------
VERBOSE = 0     # Verbosity control
SMALL = 1       # Create small binaries
REDIRECT_STDIO ?= 1 # Redirect standard I/O to USART/USB
USER_CFLAGS +=

# Target settings -----------------------------------------------------------
STM32_TOOLS = $(CONTIKI)/tools/stm32
LORA_TOOLS = $(CONTIKI)/tools/loramote
STLINK = $(STM32_TOOLS)/stlink
DFU_UTIL = $(STM32_TOOLS)/dfu-util
OPENOCD = $(STM32_TOOLS)/openocd
SERIALDUMP = $(STM32_TOOLS)/serialdump

CONTIKIVERSIONX:=${shell git --git-dir $(CONTIKI)/.git describe --tags --always}
ifeq ($(findstring 3.,$(CONTIKIVERSIONX)),)
  CONTIKI3 = 1
  CFLAGS += -DCONTIKI3=1
endif

ifeq ($(strip $(VERBOSE)),1)
  V ?= 1
endif

ifeq ($(strip $(CONTIKI_WITH_RIME)),1)
  UIP_CONF_IPV6 = 1
  CONTIKI_WITH_IPV6 = 1
  CFLAGS += -DWITH_UIP6=1 -DUIP_CONF_IPV6=1
endif

CFLAGS += -DUSE_DEBUGGER -DUSE_NO_TIMER -DLOW_POWER_MODE_ENABLE -DUSE_BAND_868 -DUSE_MODEM_LORA -DUSE_USB_CDC $(USER_CFLAGS)

MODULES += core/net/core/net/mac core/net/mac/contikimac
ifdef CONTIKI3
  MODULES += core/net/core/net/mac/llsec
else
  MODULES += core/net/core/net/ip core/net/core/net/ipv6 core/net/core/net/rime core/net/core/rpl
endif

CONTIKI_TARGET_DIRS = . apps dev LoRaMac board LoRaMac/mac LoRaMac/mac/peripherals LoRaMac/system LoRaMac/system/crypto LoRaMac/system/usb LoRaMac/system/usb/cdc LoRaMac/system/usb/cdc/inc LoRaMac/system/usb/cdc/src LoRaMac/system/usb/cdc/src LoRaMac/system/usb/cdc/sx1272

CONTIKI_TARGET_MAIN = contiki-loramote-main.c
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Appendix A. Porting Contiki to LoRaMote

FULL_TARGET_DIRS = \{wildcard \{addprefix \{CONTIKI\}/platform/loramote/, \{CONTIKI_TARGET_DIRS\}\}\}\}

CONTIKI_TARGET_SOURCEFILES = altitude-sensor.c battery-sensor.c button-sensor.c leds-arch.c lora-contiki-interface.c pressure-sensor.c radio-sensor.c serial-line-arch.c rtc-board.c sx1272-board.c timer-board.c uart-board.c uart-usb-board.c ubc-cdc-board.c

usb_desc.c usb_istr.c usb_prop.c usb_pwr.c \nadc.c fifo.c gpios.c i2c.c loratimer.c uart.c \nsx1272.c lora-radio-arch.c

#CONTIKI_TARGET_SOURCEFILES = \{foreach d, \{FULL_TARGET_DIRS\}, \{subst \{d\}/,,$\{wildcard \{d\}/*.c\}\}\}\}

include \{CONTIKI\}/cpu/arm/stm32l1/Makefile.stm32l151

CONTIKI_SOURCEFILES += node-id.c \{CONTIKI_TARGET_SOURCEFILES\}

ifndef MOTELIST

USBDEVPREFIX = \nMOTELIST = \{LORA_TOOLS\}/motelist-lora
MOTES = \{shell \{MOTELIST\} -c 2>\&- | \ncut -f 2 -d , | \nperl -ne 'print $$1 . " " if(m-/(/dev/\w+)-);'}\)
CMOTES = \{MOTES\}
endif

# Build rules -----------------------------------------------

CLEAN += *.loramote symbols.c symbols.h

# Show connected motes
motelist:

\{Q\} \{MOTELIST\}
motes:

@echo \{CMOTES\}

# Compile flashable binary
flash-init:

\{Q\}[ -f latest_build_dfu~ ] && rm -f ./obj_loramote/lora-contiki-interface.o || : # Make sure that the previously built DFU object is not used
\{Q\}rm latest_build_dfu- -f
\{Q\}touch latest_build_flash-

%flash: LDSCRIPT = stm32l1xx_md_flash.ld
%flash: | flash-init %bin
\{Q\}mv \{@:\.flash=.$\{TARGET\}\} \{@:\.flash=.flash.$\{TARGET\}\}
\{Q\}cp \{@:\.flash=.bin\} \{@:\.flash=.flash.bin\}
\{Q\}\{MAKE\} flash \{@:\.flash=.flash.bin\}

# Compile DFU-uploadable binary
upload-init:

\{Q\}[ -f latest_build_flash~ ] && rm -f ./obj_loramote/lora-contiki-interface.o || : # Make sure that the previously built flash object is not used
\{Q\}rm latest_build_flash- -f
\{Q\}touch latest_build_dfu-

%upload: LDSCRIPT = stm32l1xx_md_flash_offset.ld
%upload: | upload-init %bin
\{Q\}mv \{@:\.upload=.$\{TARGET\}\} \{@:\.upload=.dfu.$\{TARGET\}\}
\{Q\}cp \{@:\.upload=.bin\} \{@:\.upload=.dfu.bin\}
Appendix A. Porting Contiki to LoRaMote

$(Q)$(MAKE) upload $(@:.upload=.dfu.bin)

# Erase flash memory
erase-flash:
  $(Q)$(STLINK)/st-flash erase

# Install bootloader
install-bootloader:
  $(Q)$(MAKE) erase-flash
  $(Q)$(STLINK)/st-flash --reset write $(LORA_TOOLS)/loramote-bootloader/loramote_bootloader.bin 0x8000000

# Flash executable code to loramote
flash:
  $(Q)$(STLINK)/st-flash --reset write $(filter $(wildcard *.flash.bin),$(MAKECMDGOALS)) 0x8000000

# Flash executable code to loramote, when a bootloader is present
bootloader-flash:
  $(Q)$(STLINK)/st-flash --reset write $(filter $(wildcard *.dfu.bin),$(MAKECMDGOALS)) 0x8003000

# Upload executable code to loramote
upload:
  $(Q)$(DFU_UTIL)/src/dfu-suffix --add $(filter $(wildcard *.dfu.bin),$(MAKECMDGOALS)) --pid df11 --vid 0483 > /dev/null
  $(Q)$(DFU_UTIL)/src/dfu-util --device 0483:df11 --dfuse-address 0x8003000:leave --download $(filter $(wildcard *.dfu.bin),$(MAKECMDGOALS))

ifdef MOTE
serialview:
  $(Q)sleep .5
  $(Q)$(SERIALDUMP)/serialdump-linux -b115200 $(USBDEVPREFIX)$(MOTE) | $(CONTIKI)/tools/timestamp
else
serialview:
  ifeq ($(strip $(CMOTES)),)
  $(Q)$(MAKE) --no-print-directory serialview
  endif
  $(Q)$(SERIALDUMP)/serialdump-linux -b115200 $(USBDEVPREFIX)$(firstword $(CMOTES)) | $(CONTIKI)/tools/timestamp
endif

# Start openocd to debug
openocd:
  @echo "REMARK: Set the REDIRECT_STDIO variable in the makefile’s user settings section to 0 and rebuild the project before using the OpenOCD debugger!"
  $(Q)$(GDB) -ex 'target extended-remote localhost:3333' -ex 'monitor reset halt' -ex 'continue' -ex 'quit' & # Merge output of both commands to same CLI window
  $(Q)$(OPENOCD)/src/openocd -f $(LORA_TOOLS)/loramote.cfg -s $(OPENOCD)/tcl