Porting the Core of the Contiki operating system to the TelosB and MicaZ platforms

Alexandru Stan

Computer Science
International University Bremen
Campus Ring 1
28759 Bremen
Germany

Type: Guided Research Final Report (Bachelor Thesis)
Date: May 7, 2007
Supervisor: Prof. J. Schönwälder

Executive Summary

During the last couple of years wireless communication has been one of the fastest growing technologies. It has been envisioned that, in the future, small wireless devices will be abundantly scattered everywhere, autonomously exchanging relevant information and creating the so-called ”smart dust”. In this context, wireless sensor networks have become a hot topic for research. These small network devices usually have a very limited memory and their practical usage requires minimal power consumption. Combined with the scalability constrains, the problem of designing suitable operating systems(OS) for these devices has been a challenge for several research groups. Some solutions have already been implemented, each having their own benefits. One such light-weight operating system is Contiki. Being relatively new, its architecture learned from the success stories and flaws of its predecessors and therefore, it offers several advantages over its competitors. Unfortunately, in order to run Contiki on a specific platform one has to do some adjustments to the OS. Our current research project extended the reach of Contiki, by porting it to the TelosB and MicaZ platforms.

Contents

1 Summary

The aim of this Guided Research project is to investigate and actually implement a port of the Contiki OS on the TelosB and MicaZ motes from Crossbow Technologies. The port for the TelosB platform has been completely implemented and tested, while the port for the MicaZ platform is only minimal. What is still to be done for the MicaZ platform is the implementation of sensor drivers and the hardware specific libraries that are responsible for the communication between the computer and the mote through a *tun* interface and the *serial interface* (*SLIP*) and testing for the radio and the ELF loader. However, the current port for MicaZ can be considered functional since it is possible to upload monolithic images of Contiki running some applications, such as a ”blinking lights” program.

The reason for selecting these platforms was that, at the time, these are the only ones available to the research group at Jacobs University. Additionally, when the project started, in the community there were no working ports of Contiki to these platforms, and therefore our work is relevant also to other research groups who would like to use Contiki on TelosB and MicaZ.

The paper continues with an introduction to wireless sensor networks (in order to better understand the requirements for programming such systems) and to the ContikiOS (such that the reader gets some understanding of how it works). In this section it is also discussed why Contiki might be a good choice over the already existing operating systems for embedded devices, such as the popular TinyOS. After the introduction, a more detailed description of the porting is presented. In the end, the results of this project are analyzed, the direction of future work in this area is discussed and, a conclusion of the whole project is drawn.

2 Introduction

2.1 Wireless Sensor Networks

Wireless Sensor Networks(WSN) constitute at the moment the topic of many research papers worldwide. WSNs are generally defined as ”lots of computers interacting within the world” [1]. To be more precise, WSNs are comprised of tiny wireless devices, often called motes, that have several sensors and send data readings of interest to a specific central node, which is responsible for a complex analysis of the data and/or for passing forward the information gathered to some else. WSNs are most commonly used in science for habitat monitoring, in industry for equipment monitoring and control, in military for vehicle tracking, or by the government for vehicle traffic monitoring. Other future usage of WSN include process control and verification in transport and logistic processes [2], disaster monitoring and management, or other ”smart life” scenarios [3]. Since 2000, when research in this area was still at the beginning, WSNs have now been successfully developed and deployed outside the labs in real life scenarios.

The designers of operating systems for sensor motes are often faced with the challenge of finding the right abstractions and lightweight mechanisms that offer a rich enough execution environment, while remaining within the systems strict resource limitations. Typically, a mote has between 2 to 10 kB of RAM, 4 to 16 kB of EEPROM, 48 to 128kB of Flash memory and are equipped with 8bit or 16bit RISC processors running at around 8MHz. Moreover, the motes should consume as little power as possible since, in spite of running on batteries, typical deployments require that they function without any type of interaction for a long time. Having these limitations in mind it is easy to see how important it is to have an OS that uses as little resources as possible while at the same time providing all the necessary functionality

2.2 The Contiki Operating System

Contiki is an open source, C language written, minimal operating system. Its footprint is usually not bigger that 4kB, making it suitable for resource constrained systems [4], [5]. It was written by Adam Dunkels, from the Networked Embedded Systems group at the Swedish Institute of Computer Science. Contiki supports dynamic loading and replacement of individual programs and services. Moreover, optional preemptive multi-threading is implemented as a library that is linked on-demand with the programs that explicitly require this feature. The kernel is event-driven, and interprocess communication is done using message passing signals. Other distinct features of this OS include the native TCP/IP support using the uIP or lwIP TCP/IP stack [6], and the graphical subsystem that can be used with either direct graphic support for directly connected terminals, or networked virtual dis-

play with VNC or Telnet. What is also worth noticing about Contiki is the usage of a programming abstraction called *protothreads*, which frees programmers from the constraint of using state machines whenever working with event driven kernels. A quick description of this is given later in this document, while a more detailed overview can be found in [7] and [8].

To better understand Contiki one should take a look at its system architecture [4]. Typically, a running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. Communication between the processes always goes through the kernel, which does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware. A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events. Looking at it from a higher perspective, the Contiki system is made up of two parts: the core and the loaded programs. Typically, the core consists of the kernel, the program loader, the most commonly used parts of the language runtime, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image and is usually not modified after deployment (although it is possible to use a special boot loader to overwrite or patch the core). The program loader is in charge of loading/unloading the programs into the system either by using the communication stack or directly attached storage (such as EEPROM). It is also important to note here the abstraction model: programs know the core, the core does not know the program.

Using Contiki as the operating system of the motes offers many advantages. As mentioned before, Contiki's kernel is event based, which means that the block wait abstraction is not supported. For this reason, when programming such a system one will have to use a state machine to implement the control flow for high level logic that cannot be expressed as a single event handler. However this approach is cumbersome and programmers usually have problems with it. To overcome this shortcoming and simplify the programs, Contiki implements a programming abstraction called *protothreads*. One has to remember though, that as the name implies, they are merely tools for creating a conditional blocking wait statement. Protothreads have a low memory overhead and are also stackless, i.e. they all run on the same stack and context switching is done by stack rewinding. Since processes in Contiki are nothing more than protothreads it makes sense to take a closer look at this abstraction.

Protothreads are written in C and no compiler changes are required. The only problem with this abstraction are the following restrictions to the programmers: automatic variables not stored across a blocking wait and no *switch* statements are allowed. To overcome this one has to use static local variables and avoid switch statements completely Below we show the simple implementation of protothreads:

```
struct pt { unsigned short lc; };
#define PT_INIT(pt)             pt->lc = 0
#define PT_BEGIN(pt)            switch(pt->lc) { case 0:
#define PT_EXIT(pt)             pt->lc = 0; return 2
#define PT_WAIT_UNTIL(pt, c)    pt->lc = __LINE__; case __LINE__: \
                                if(!(c)) return 0
#define PT_END(pt)              } pt->lc = 0; return 1
```

Analyzing this implementation it is easy to see why the restrictions mentioned above exist. As previously mentioned, Contiki processes are just protothreads. To better illustrate how this works we present the following piece of code which gets light sensor readings and prints it to the screen when the mote is connected to the computer via USB and a special program, *tunslip*, is running (it creates the tun and SLIP interface for communication over the serial line).

```
/*declare the process*/
PROCESS(light_process, "light process");

/*make the process start when the module is loaded*/
AUTOSTART_PROCESS(&light_process);

/*define the actual process code*/
PROCESS_THREAD(light_process, ev, data)
{
  PROCESS_BEGIN();  /*must begin with this*/
  /*note that we have to make the variable static*/
  static struct etimer etimer;
/* Photosynthetically Active Radiation. */
  unsigned static reading1;
/* Total Solar Radiation. */
  unsigned static reading2;
  printf("light_process starting\n");
  /*initialize the light sensors*/
  sensors_light_init();
  while(1) {  /*do this forever*/
  /*set active timer*/
    etimer_set(&etimer, CLOCK_SECOND);
   /*take readings at discrete time intervals*/
    PROCESS_WAIT_UNTIL(etimer_expired(&etimer));
    reading1 = sensors_light1();
    reading2 = sensors_light2();
    printf("READING1: %2d\nREADING2: %u\n", reading1, reading2);
  }

  PROCESS_END(); /*finish process*/
}
```

In the example above we also saw how to use timers. There are basically four types of timers that can be used depending on the needs of the programmer:
- struct timer : passive and only keeps track of its expiration time
- struct etimer: active and sends an event when it expires

- struct ctimer: active and calls a function when it expires
- struct rtimer: real-time timer and calls a function when it expires

There are two ways to make a process run. You can post an event and asynchronously run the process: $process\_post(process\_ptr, eventno, ptr)$, or run it immediately: $process\_post\_synch(process\_ptr, eventno, ptr)$. However if you post a process you should not call it from an interrupt. The other way to make a process run is to poll the process using: $process\_poll(process\_ptr)$.

When developing software for large sensor networks it is very important to be able to dynamically download program code into the network. One possible advantage of this is the ability to patch bugs in operational networks. As opposed to most operating systems for embedded systems, such as TinyOS, which require a complete binary image of the entire system to be flushed into each device, Contiki has the ability to load and unload individual applications or services at runtime. Of course, this greatly reduces the actual time required to do the modifications. As an example just think of a network of 30 motes at the time when some routing protocol is tested and modifications to the code are bound to be numerous. If it takes us around 2 minutes to program one mote than we would need an hour to program the entire network. With Contiki, this reprogramming can be done in under 5 minutes! This is why dynamically loading programs into the system and programming over the radio is one of the key features of Contiki. Dynamic loading of programs is done using ELF object files and more information can be found in [9].

The support of a native TCP/IP protocol stack is very important in the context of embedded systems because it provides interoperability with the existing systems and makes it easy to integrate Contiki into the existing IP network infrastructure. As mentioned in [1] one of the goals of future research in WSN should be the consideration of the Internet side of them, i.e. the exchange of data between different WSN and organizations.

Contiki uses two communication stacks uIP and RIME. uIP is a minimal RFC compliant implementation that has a 5kB footprint. It is important to remember that uIP has several limitations: no IP options, no sliding window, can handle only one network interface, uses a single buffer for both incoming and outgoing packets and does not buffer sent packets. A more detailed description of these can be found in [10]. Note however that this paper is a bit old and UDP is now implemented by Contiki uIP (contrary to what is written there). The other communication stack used by Contiki is RIME. Rime is a new lightweight communication stack designed for low-power radios. Rime provides a wide range of communication primitives, from best-effort local area broadcast, to reliable multi-hop bulk data flooding [11]. The relation between the two communication stacks is best described in Figure 1.

Another noticeable feature of Contiki is the ability to support multi-threading by implementing it as a library that can be optionally linked with programs that explicitly require it. This feature is important whenever a lengthy computation is

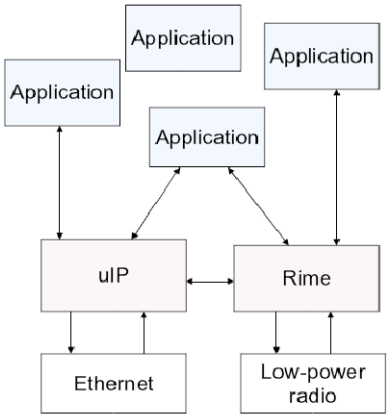

Figure 1: The Contiki communication overview

performed, which given the event-driven kernel, will monopolize the CPU and will make the system unresponsive to external events.

Other differences between Contiki and the more popular embedded operating systems, like TinyOS include the code footprint and also in the underlying architecture, such as scheduling. While the native TCP/IP support offered by Contiki is nice, it requires more resources from the system (and might not be essential for all WSN deployments). Concerning scheduling, Contiki uses FIFO event queue and pool handlers with priority, as opposed to only a FIFO event queue, in TinyOS.

The ability to do simulations for WSNs is incredibly important for system development. Although a variety of simulators exist, they allow simulations only at a fixed level, such as application, operating system or hardware level. One more advantage of Contiki is that is has a specially designed cross-level simulator, named COOJA, which has been shown to perform well [12]. This means that developers of Contiki based WSNs have a powerful tool on their hands for thoroughly testing different system ideas before they are to be implemented in practice, thus enabling them to save time and money.

One last advantage of Contiki is that once the porting for a platform is done, there are a number of very useful applications available, such as Telnet. These applications can come in very handy for developers. The Contiki repository is well maintained by the small community involved in this project and the feedback received so far by those who adopted this OS is positive.

3 Porting the Contiki OS

Because WSNs are highly application dependent, there are quite a few available platforms, each with its own flavor of sensors and maybe even different processor or radio systems [13], [14]. Since a WSN might be composed of different platforms which ideally run the same OS, it is easy to see why portability has become a system requirement for the operating systems running on the nodes. One of the few common characteristics of the platforms is the CPU architecture which uses a memory model without segmentation or memory protection mechanisms[4]. Usually, program code is stored in reprogrammable ROM and data in RAM. Contiki's design addresses the problem of portability by limiting the abstraction provided by the base system to just basic CPU multiplexing, event handling features and support for loadable programs and services. All the other abstractions are provided by the libraries that have nearly full access to the underlying hardware. Therefore porting Contiki to other platforms requires, at least in theory, only several modifications depending on the hardware of the nodes. However, in practice, although not a lot of lines are required for a port, it might prove to be difficult to debug low-level hardware specific problems.

3.1 A general port

Before discussing the details of the individual ports, we should first see what are the general steps required for a port. There are basically two directories which are relevant when doing a port: */platform* and */cpu*. The typical way to proceed is to create a subdirectory in the */platform* directory. There one should copy and modify the files from */platform/native*, which contains the simplest port. In some cases it might be more useful to modify the ports of some platforms which are much more similar to the platform to be ported. For example in the case of TelosB the */platform/sky* directory was used, since that contained the port to Tmote Sky (which is similar to TelosB). One thing to always keep in mind is that some of the ports present in that directory might not be complete, and it is generally not a good idea to trust them completely. The CPU specific code is found in */cpu/cpu_name* and although the most popular CPUs have been ported, it is advisable not to exclude this as a possible bug source.

Probably the most painstaking file to be created when porting a platform is *contiki-conf.h* found in */platform/platform_name*. It contains all the platform specific configuration options and more: C types, compiler configuration, clock configuration, uIP configuration, low-level memory addresses, pin configurations, etc. Usually to edit this file correctly one must have the platform datasheet in front of him.

In order to integrate the new platform into the Contiki systems a *Makefile.platform_name* has to be present in the */platform/platform_name* directory. This makefile gener-

ally contains platform specific options required at compile time. More precisely, it usually specifies the files containing low-level hardware specific code (typically creating a variable named *$ARCH*), the type of processor used and the path to its source, the bootstraploader, and some specific rules. The *Makefile* from */platform/platform_name* usually just specifies which files to be made and includes *Makefile.include* which includes all the other options needed. This is however the ideal case and in practice, when developing a port it is usually the case that the *Makefile* from */platform/platform_name* takes care of everything. When the port is complete one can then try to clean up this Makefile and make it look much more similar to the ideal one. Another Makefile to be taken care of is the CPU specific one located in */cpu/cpu_name*. This Makefile takes care of the cross-compiler rules and definitions and specifies the path where to look for source files.

For every new type of CPU used a specific *clock.c* file has to be written. This file can be found in */cpu/cpu_name*. Although it might contain more functions just 3 are essential: *clock_init()*, *clock_delay()* and *clock_time()*. The ELF loader is another element that needs porting just for new processor types. Since msp430 and AVR are very popular microcontrollers in the WSN world, ports for these are already available. Additionally, for every CPU the multi-threading library needs to be ported. The file */cpu/cpu_name/mtarch.c* contains all the low level implementation.

After this is done, the next thing to worry about is communication. The network device drivers are concerned with two basic operations: sending out a packet, and receiving one back. What is to be done is to ensure that the proper interface between uIP and hardware specific code and between RIME and hardware specific code exists. Additionally, another file should contain hardware specific code (cpu - radio chip communication).

In the end, in order to be able to run Contiki the *contiki-paltform_name-main.c* should be created. This is more of a dummy implementation - it just initializes the harware and then schedules and runs the processes defined. In reality one would upload a much more complicated core to the mote, for example one that makes the mote act as a gateway for the other nodes of the network. Hence the *contiki-paltform_name-main.c* file is present more for consistency than for actual necessity.

After this is done, one should be able to successfully upload an imagine of the Contiki system on the new platform. In practice you would want to add more features to your port, such as sensor support. Sensors work in the system by sending events whenever something happens. The code for the general sensor implementation is found in */core/lib/sensors.c*. The low level code for controlling individual sensors should be located in a different directory (*platform/platform_name/dev* usually).

3.2 TelosB Port

TelosB [15] was developed at UC Berkeley and is widely used in research, although a commercial offspring (Tmote Sky) [16] is used in industry. TelosB is based on a 8MHz 16-bit RISC TI MSP430 processor with 10kB RAM, 16kB of configuration EEPROM, 48kB flash memory. It has a digital I/O, I2C and SPI interface in addition to the UART serial communication provided by the FT232BM (FTDI) chip. It draws 1.8mA in active mode and only 5.1in sleep mode. It is IEEE 802.15.4 compliant having a common CC2420 radio chip. The mote has several sensors, such as visible and IR light, humidity and temperature.

Porting to the TelosB platform implied making some changes to the *sky* port. Luckily no very low-level code changes had to be done. Work done was towards system integration, more specifically fixing makefiles and doing modifications to various files in charge of providing higher level functionality. The resulting port is complete and incorporates all the functionalities of a contiki system.

The code for the port can be found in */cpu/platform/telosb* and in */cpu/msp430*. In order to get it running the MSPGCC Tool Chain needs to be installed. Note that the package does not contain the bootstraploader msp430-bsl which needs to be downloaded separately. For simplicity there will be a special directory */utils* (included in the tarball to be submitted with the code), which contains all relevant files needed. Once this Tool Chain is installed, simply connect the mote to a USB port and use: *make TARGET=telosb file-to-be-used-as-kernel.u*. The memory of the mote is erased and the new kernel is loaded. Make sure though that you adjusted the Makefile accordingly before issuing the *make* command:

```
KERNELS = file-to-be-used-as-kernel.ihex
file-to-be-used-as-kernel.out: file-to-be-used-as-kernel.o $(LIB)
```

Now in order to load modules you have two options: connect every mote individually to the USB or use a so-called gateway kernel on one of the motes connected to the USB and a client kernel on the other motes. The code for this is in */platform/telosb/gateway.c* and */platform/telosb/client.c* and you flush it to the mote as described above. Note that IP addresses are assigned statically in the code. Example from */platform/telosb/client.c* :

```
/*assign address of node*/
struct uip_fw_netif cc2420if =
{UIP_FW_NETIF(172,16,0,9, 255,255,0,0, cc2420_send_uaodv)};
....
/*default gateway used*/
uip_ipaddr(&uip_draddr, 172,16,0,1);
```

Then in order to upload the loadable module, again make sure you first adjust the relevant part of the Makefile:

```
PROGS = program-to-be-loaded.ko
 program-to-be-loaded.ko:  program-to-be-loaded.o
```

Now by using the *make TARGET=telosb* you can make the *.ko* files and the necessary tools required for uploading: *tunslip* and *codeprop*. *tunslip* is a tool that creates the tun and slip interface in order to be able to talk TCP/IP with the mote over the USB. Run it as: *sudo ./tunslip address netmask*. The nice feature about this tool is that now all the *printfs* done on the mote are shown on the terminal. After *tunslip* is running you can load a module by calling the *codeprop* application with the name of the desired module and the IP address of the mote you wish to upload code to(you can use the IP of a client mote if *tunslip* is connected to a mote running *gateway*).

If you for example wish to get some sensor readings from a mote somewhere in an environment and print it to the screen of a computer connected via *tunslip* to another mote you can use the *udprecv.c* and *udpsend.c*. Just load these as modules on the mote connected to the computer and on the one taking the sensor readings, respectively. Source code is self explanatory. For example *udpsend.c* waits for some time (a set timer to expire), then it polls the uIp connection and waits for a *tcpip_event* to be received. Afterwards it buffers the sensor readings and just sends the contents of the buffer over the radio.

3.3 MicaZ Port

MicaZ (MPR2400) [17] is specifically designed for deeply embedded sensor networks. It has an IEEE 802.15.4 compliant radio interface based also on the CC2420 chip ,and plug and play capability with all of Crossbow's data acquisition boards, gateways, and software. It features a ATMEL ATMega128L 8bit RISC CPU running at 7.37Mhz and 32 registers. The memory capacity is 4kB RAM, 128kB Flash and 4kB EEPROM. Programing of the board was done using the Mote Interface Board MIB520, which has a on-board in-system processor (ISP) ATmega16L. This ISP is getting the code to be programmed on the mote through the USB port and then does the actual code programming of the ATMega128L. The MIB has an additional JTAG pod for in-circuit debugging, which was, however, not used. The MicaZ does not come with any on-board sensors but a special sensor board can be attached as needed.

As opposed to TelosB port, the MicaZ port also implied low level programming. The datasheet for the ATMEL ATMega128L microprocessor and the MPR/MIB user manual were extensively used. A lot of low-level instructions had to be rewritten since for the ATMega128 CPU a 0 means that a bit is set and a 1 means that it is not (as opposed to MSP430 and the other CPUs). This obliged us to rewrite parts of code relying on logical operators so that in the end the low level functions will provide the same abstractions as for other CPU. Later in this section we will show an example from the code that determines the LEDs activation. Apart from this we had to modify the contiki-conf.h to so that it incorporates the right definitions for pins and memory addresses. In the end, a minimal port was completed

and a ”blinking lights” program successfully tested as part of the uploaded system image.

As an example of how one of the files in charge of providing abstractions from the low level code look like we can take a looks at a part of *platform/dev/qleds.c*. Also for simplicity, the relevant lines from *contiki-conf.h* and *leds.h* have been shown in the beginning

```
/*high level definition of leds in leds.h*/
#define LEDS_GREEN  1
#define LEDS_YELLOW 2
#define LEDS_RED    4

...

/* low level definition of leds in contiki-conf.h (as specified by
 the datasheet) */
/*a metal leg is controlled by DDR, PORT and PIN
DDR bit tells a leg to act as input (0) or output (1)*/
/*LEDS_PxDIR to be later set to 1*/
#define LEDS_PxDIR     DDRA
#define LEDS_PxOUT     PORTA
/*a zero means that a led is on*/
#define LEDS_CONF_RED    0x03 //011
#define LEDS_CONF_GREEN    0x05 //101
#define LEDS_CONF_YELLOW   0x06 //110

...

/*implementation in qleds.c*/
/*works similar to an enum:  l2p[0]=000, meaning all leds are on*/
static const unsigned char l2p[LEDS_ALL + 1 ] = {
  LEDS_CONF_RED & LEDS_CONF_YELLOW & LEDS_CONF_GREEN,
  LEDS_CONF_RED & LEDS_CONF_GREEN,
  LEDS_CONF_RED & LEDS_CONF_YELLOW,
  LEDS_CONF_RED,
  LEDS_CONF_YELLOW & LEDS_CONF_GREEN,
  LEDS_CONF_GREEN,
  LEDS_CONF_YELLOW,
  7 // 111 - all leds are off
};

/*leds - as described in leds.h*/
void leds_toggle(unsigned char leds)
{
  //*give lets the low level meaning for ease of implementation*/
  if (leds == LEDS_GREEN)
    leds = LEDS_CONF_GREEN;
  if (leds == LEDS_YELLOW)
    leds = LEDS_CONF_YELLOW;
  if (leds == LEDS_RED)
    leds = LEDS_CONF_RED;
  if (leds == LEDS_ALL)
    leds = 0; // 000 means that all pins are set
```

```
  unsigned char aux;
/*after pen and paper derivation the bit operation that toggles
the leds was found to be negating the result of XOR
example : LEDS_PxOUT=010; red and yellow on
 leds = 011  - toggle red
010^010=001; !001 =110 - red is off; yellow stays on
*/
  LEDS_PxOUT ^= l2p[leds & LEDS_ALL];
/*now we do the not; have to use a small trick since we are
interested just in the last 3 bits but we are using a whole byte*/
  aux=LEDS_PxOUT;
  aux = aux <<5 ;
  aux = ~aux;
  aux = aux>>5;
  LEDS_PxOUT=aux; //all done!
}
```

As with the TelosB port, one needs to get hold of a special cross-compiler. For the MicaZ the GCC-AVR compiler was used. Additionally, a special BSL was used for loading the Contiki imagine to the mote: *uisp*. Although this tool is no longer maintained by the community, problems were experienced with *avrdude*, which is at the moment the popular tool for programming the microcontroller through the parallel port of the computer. Of course this meant that the specific makefiles created had to be written such that these tools are used. Once this was done the same procedure described in the TelosB section can be used in order to build and upload the Contiki system or individual modules.

4 Results Analysis

The aim of the Guided Research project was to port the Contiki OS to the TelosB and MicaZ platforms. This task presented several challenges which were tackled sequentially. At first it was important to familiarize oneself with the concepts of this operating system. What this meant was that a series of papers were read and several websites containing useful information visited. Next came the challenge of having to work with a very large and diverse code base. To overcome this, we identified the points of focus (the directories and source files relevant to our task) and concentrated on those. During the TelosB port we got very well familiarized with the integration issues, high-level source files necessary to run Contiki, Makefiles, the applications of the OS and how to write new ones. During the MicaZ port we were faced with the challenge of editing files containing the low-level implementation and debugging weird behaviour to the datasheet level. Although painstaking at first, through a process of educated guesses and trial and error we managed to overcome these problems and get a minimal port going. Probably one of the most difficult obstacles was figuring out what exactly is responsible for various bugs in the system. Most often, the case was that the problem did not lie in just one source file, but in several. This was very hard to debug especially when its effects were that the communication between the mote and the computer through *tunslip* did not work.

In the end, a complete port was done for the TelosB platform. All functionalities of the system were exhaustively tested through the help of the already existing applications and by writing new ones. For the MicaZ, only a limited port was achieved. A Contiki kernel containing a ”blinking lights” process was successfully flushed to a MicaZ mote and successfully tested.

Future work would aim at extending the MicaZ port by first fixing the problem of communicating with the mote via *tunslip*. A possible problem might be the low level interaction between the cpu and the serial port. Following this, the dynamical loading of modules and the radio can be tested. Some more work would also be required for the sensors of the sensor board. The multithreading library port should follow afterwards. Optionally, what can also be done is to organize and clean up the Makefiles and the working directory, because the porting process created a bit of a mess.

Thanks to this project we were also able to deepen our knowledge of operating systems and especially memory constrained operating systems. We learned about the state of the art in this field and about the challenges that exist. We also learned how to develop code for event driven kernels and the limitations of it. Moreover, we got a bit of know-how on the hardware that is usually present in WSN motes and how to program it. Also, we got some experience in debugging weird behaviour of motes.

5 Conclusion

The main contribution of this project was to provide a working implementation of the Contiki port to TelosB and a minimal port to MicaZ. The ports were successfully tested within their provided functionality. At the moment the online repository does not contain these specific ports, so this work could also be integrated in future versions of the system.

The Contiki OS was proved to be portable, although we cannot say at the moment if it is more portable than other operating systems (simply because this was the only OS ported). What can be certainly stated is that developing applications for Contiki is relatively straightforward once the developer gets familiar with the basic coding style. Once accustomed to the system, the difficulty of the port resides mostly in understanding the datasheets of the given platform. Additionally, this document gives an introduction to Contiki and provides a step-by-step how-to for those who want to use this operating system.

References

[1] John Heidemann. Sensornets and the next big thing. Keynote talk at the European Conference on Wireless Sensor Networks, January 2007.

[2] Leon Evers, Mark J. J. Bijl, Mihai Marin-Perianua, Raluca Marin-Perianu, and Paul J. M. Havinga. Wireless sensor networks and beyond: A case study on transport and logistics. In *International Workshop on Wireless Ad-Hoc Networks (IWWAN 2005)*, May 2005.

[3] Frank Oldewurtel and Petri Mähönen. Neural wireless sensor networks. In *The International Conference on Systems and Networks Communications (ICSNC, best paper award)*. RWTH Aachen University, Department of Wireless Networks, November 2006.

[4] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, November 2004.

[5] A. Dunkels. *Contiki/ESB Reference Manual*. Swedish Institute of Computer Science, July 2005.

[6] A. Dunkels. Towards tcp/ip for wireless sensor networks. Licentiate Thesis. Mälardalen University and Swedish Institute of Computer Science, March 2005.

[7] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[8] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[9] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[10] Adam Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003.

[11] Adam Dunkels. Rime — a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sen-*

sor Networks (EWSN), Poster/Demo session, Delft, The Netherlands, January 2007.

[12] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Proceedings of the 31st Annual IEEE International Conference on Local Computer Networks*, November 2006.

[13] Snm - the sensor network museum. http://www.btnode.ethz.ch/ Projects/SensorNetworkMuseum. This is an electronic document. Date retrieved:March 8, 2007.

[14] Can Basaran (YTU), Sebnem Baydere (YTU), Giancarlo Bongiovanni (CINI)and Adam Dunkels (SICS), M. Onur Ergin (YTU), editor) Laura Marie Feeney (SICS, Isa Hacioglu (YTU), Vlado Handziski (TUB), Andreas Kopke (TUB), Maria Lijding (UT), Gaia Maselli (CINI), Nirvana Meratnia (UT), Chiara Petrioli (CINI), Silvia Santini (ETHZ), Lodewijk van Hoesel (UT), Thiemo Voigt (SICS), and Andrea Zanella (DEI). Research integration: Platform survey. critical evluation of platforms commonly used in ebedded wisents research. Technical report, June 2006.

[15] Crossbow Technology. *TelosB Datasheet*.

[16] Moteiv Corporation. *Tmote Sky Datasheet*.

[17] Crossbow Technology. *MicaZ Datasheet*.

[18] Adam Dunkles. New contiki web site. http://www.sics.se/contiki/news/new-contiki-website.html. This is an electronic document. Date retrieved:March 8, 2007.

[19] Wireless sensor networks blog. http://www.wsnblog.com. This is an electronic document. Date retrieved:March 8, 2007.

[20] A. Dunkels. *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, Swedish Institute of Computer Science, February 2007.