MICAz and TelosB Sensor Device Driver Port to Contiki

Adrian Đokić

May 7, 2007

Computer Science
Jacobs University Bremen
Campus Ring 1
28759 Bremen
Germany

Type: Guided Research Project
Supervisor: Prof. J. Schönwälder

Contents

1 Executive Summary

A Wireless Sensor Network (WSN) consists of physically spaced communicating sensors that probe the environment and can make it intelligible. WSNs have the potential to channel external data to information systems and decision makers to provide advanced monitoring, automation, and control solutions for a range of industries. The use of wireless sensor networks is almost limitless for industries and applications with specific needs such as reliability, battery life, range, frequencies, and topologies, size of the network, sampling rate and sensor use. Software, the driving force of any digital system, must work in an efficient and reliable manner. A well-functioning operating system, a selection of programs that manage the hardware and software resources of a computer, is the key to a reliable hardware platform. The core of any operating system, namely the kernel, interfaces between available hardware and running applications, e.g. device drivers.
In contrast to popular operating systems and hardware platforms, e.g. Microsoft® Windows® running on desktop computers, wireless sensor devices are much more resource-constrained in terms of energy and memory demand. It is the function of any operating System and kernel to provide proper application loading and execution mechanism, which largely determines processor and memory usage, and thus energy consumption.
Contiki, a small, open source computer operating system, was developed for use on memory-constrained systems. It consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime.
Since it is typically the radio chip that consumes the most power on a wireless sensor device, Contiki's dynamism promises an advantage over other operating systems. During a system software update, for example, less data are transferred between the wireless sensor network, because only part of the operating system devices or applications require change.
While Contiki shows many advantages over other embedded operating systems, one drawback is that it has not been ported to many hardware platforms, primarily because it is a relatively new OS. This document shows how Contiki has been ported to two relatively popular WSD hardware platforms, namely the TelosB and MICAz. The knowledge gained by this project aims to be applied to other platforms.

2 Summary

Wireless Sensor Networking (WSN) is a relatively modern computing concept which has introduced a completely new paradigm for distributed sensing. Wireless Sensor Networks consist of physically spaced communicating sensors that probe the environment and can make it intelligible. There exist a number of environments in which such wireless sensor devices (WSDs) can be deployed, ranging from the 20th century dream of having an intelligent house to being scattered in the field for military applications. One clear advantage is that a WSN is easy to set up.
Software, the driving force of any digital system, must, however, be programmed to perform the desired functions, such as creating an ad-hoc network. Since energy is the most vital resource for a WSD, a proper application loading and execution mechanism is needed, for it largely determines processor and memory usage, and thus energy consumption.
Contiki is a small open source computer operating system (OS) that has been developed for use on memory-constrained systems. Despite being designed for embedded systems with small amounts of memory, a full installation of Contiki offers advances features such as multitasking, TCP/IP networking, protothreads, a low-overhead mechanism for concurrent programming, a windowing system and Graphical User Interface (GUI), as well as the world's smallest web browser.
Contiki was designed to and also runs on a variety of platform ranging from embedded microcontrollers such as the MSP430 and the AVR to old homecomputers. It is also possible to compile and build both the Contiki system and Contiki applications on many different development platforms.
The TelosB and MICAz hardware platforms have been developed and published to the research community by the University of California, Berkeley. These two wireless sensor devices feature an MSP430 and AVR microcontroller, respectively. The porting process thus involves making use of the present interfaces, which includes manipulating pin, timer, and other internal register values. Since the Contiki kernel has already been ported to other hardware platforms such as TelosSky and ESB, the general kernel code structure can be taken from these and changed to suit a given platform, i.e. perform proper initialization functions and such.
In addition to the Contiki core, device drivers were written to interface the readings obtained from the IR light sensor on the TelosB, for example. The process of writing lower-level software, i.e. device drivers,

requires the block diagrams and other hardware platform-specific documentation to, for example, apply a voltage to a certain pin in order to turn on the radio chip, and another to read data from it. This process has been documented further in the following sections.

3 Introduction

Both the TelosB and MICAz platforms feature a CC2420 single-chip IEEE 802.15.4 compliant RF transceiver designed for low-power and low-voltage wireless applications. While the TelosB features some on-board sensors such as visible and infrared light as well as for humidity and temperature, the MicaZ features an expansion port on which, for example, the MTS300 Mica Sensor Board can be attached. In addition to the MTS300, which offers a temperature, light, and acoustic sensor, additional boards in the product line can be stacked and used to expand the system.
The Contiki kernel along with some of these device drivers have been ported to both the TelosB and MICAz hardware platforms. Some of the features of Contiki as stated in [3]are described below.

> Contiki is an open source multi-tasking operating system designed for embedded systems with small amounts of memory. A typical Contiki configuration takes up 2KB of RAM and 40KB of ROM. The operating system consists of an event-driven kernel on top of which applications are dynamically loaded and unloaded at runtime. Contiki processes use light-weight stackless *protothreads* that provide a linear, thread-like programming style on top of the event-driven kernel. The code footprint is on the order of kilobytes and memory usage can be configured to be as low as tens of bytes. Contiki is written in the C programming language and is freely available as open source under a BSD-style license.

Protothreads can be used with or without an underlying operating system to provide blocking event-handlers. They provide sequential flow of control without complex state machines or full multi-threading.[6]
The Contiki kernel is event driven, and also supports per-process optional preemptive multi-threading, interprocess communication using message passing through events, as well as an optional GUI subsystem with either direct graphic support for locally connected terminals or networked virtual display with VNC.[3].
The two wireless device platforms at our disposal, namely the Crossbow TelosB and MICAz with an MST300 sensor board[1]. Both the TelosB and MICAz have at least a temperature and light sensor at their disposal, are IEEE 802.15.4 compliant, and have USB program-

[1] MICAz and MST300 will be used interchangeably throughout. The device should be made clear by the context.

ming capabilities.
They come with out-of-the-box support for the default TinyOS operating system, an already widespread embedded system targeting WSNs. Nevertheless, widespread use does not necessarily mean the best[2]. Each node runs a single statically-linked system image, making it hard to run multiple applications or incrementally update applications.[7]
[4]states that:

> Contiki features a dynamic linker that can link, relocate, and load standard ELF object code files.
>
> The need for correcting software bugs in sensor networks was early identified[sic]. Even after careful testing, new bugs can occur in deployed sensor networks caused by, for example, an unexpected combination of inputs or variable link connectivity that stimulate untested control paths in the communication software. Software bugs can occur at any level of the system. To correct bugs it must therefore be possible to reprogram all parts of the system.

The Contiki build system has been designed to be able to easily write and compile applications and the core for different platforms. Given the nature of the build system it was, in fact, relatively simple and easy to set up a new platform working environment. The first way to evaluate a proper port of the core was to write an interface for the LEDs situated on each of the boards. This was a relatively simple task, namely looking up which pin corresponded to which LED in each of the hardware platform manuals and applying a voltage.[3]

[2] Best is, obviously, a relative term, depending on the application. A certain corporation will, however, always come into question.

[3] In the case of the MICAz, a 0-bit means that a pin is programmed.

4 Work Documentation

As has already been mentioned in the previous section, the Contiki build system has been designed to a) make it easy to recompile applications for different platforms, and b) keep application code out of the Contiki directories. Ideally, no changes are needed to the programs except for when developing low-level software. If some Contiki source code needs to be altered for a specific platform, then a local copy should be created. The general Contiki directory structure is shown below.

apps/ architecture-independent applications

core/ system source code

cpu/ CPU-specific

doc/ documentation

examples/ example project directories

platform/ platform-specific code

tools/ software for building Contiki, sending files

The general Contiki kernel structure is shown below.

```
int
main(int argc, char **argv)
{
  /*
   * Initalize hardware.
   */
  cpu_init();clock_init();/* ... */
  /*
   * Initialize Contiki and our processes.
   */
  process_init();/* ... */
  /* Configure IP stack. */
  uip_init();
  /* ... */
  /* Start IP stack. */
  process_start(&tcpip_process, NULL);/* ... */
  /*
   * This is the scheduler loop.
   */
  while (1) {
    do {
      /* Reset watchdog. */
    } while(process_run() > 0);
  }
  return 0;
}
```

A Contiki system is partitioned into two parts: the core and the loaded programs. The partitioning is made at compile time and is specific to the deployment in which Contiki is used. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. The core is generally not modified after deployment, even though it should be noted that it is possible to use a special boot loader to overwrite or patch the core.[5]
The porting process began by creating a folder under `platform/` and setting up the respective Makefiles and basic core source code.

4.1 Porting Contiki to the TelosB

It should be noted that a Contiki port to the Tmote Sky platform was released during the progression of this project. Just like the TelosB, the Sky platform consists of an MSP430 microcontroller and a CC2420 radio chip. The porting of the Contiki core to the TelosB, therefore, required relatively few changes to the Sky source code.[8]
In order to be able to develop the device drivers, the circuit diagrams of the TelosB MCU and sensors shown in Figures 1 and 2 were needed.
After the Contiki core was ported to TelosB[8], it had to be tested whether or not it was working properly. The simplest way to determine this was to write a simple API to switch the LEDs on and off.
Referring back to Figures 1 and 2, we see that the red, green, and blue LEDs—LED1, 2, and 3—correspond to pins P5.4, P5.5, and P5.6, respectively, i.e. pins 4, 5, and 6 on port 5.
Firstly, the API found in `core/dev/leds.h` had to be implemented; to be more precise, the following functions were implemented:

```
unsigned char leds_get(void);
void leds_on(unsigned char leds);
void leds_off(unsigned char leds);
void leds_toggle(unsigned char leds);
void leds_invert(unsigned char leds);
void leds_arch_init(void);
unsigned char leds_arch_get(void);
void leds_arch_set(unsigned char leds);
```

In addition to the general Contiki API, the platform-specific header files, i.e. for the MSP430x1611 microcontroller, were used in order to determine the function calls for certain low-level operations. As an example of how this was done is shown in the codebelow.

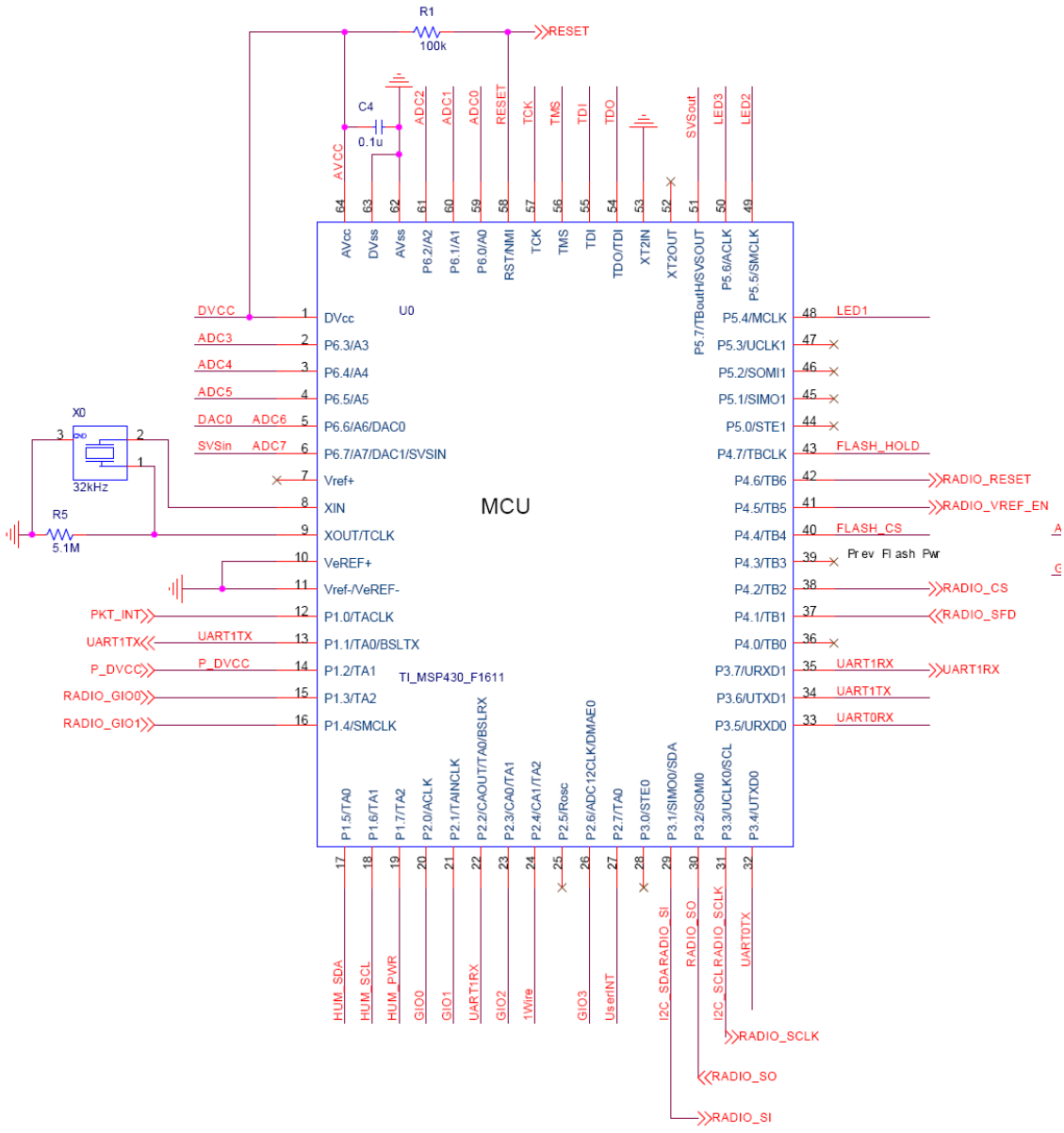

Figure 1: TelosB MCU circuit diagram[2]

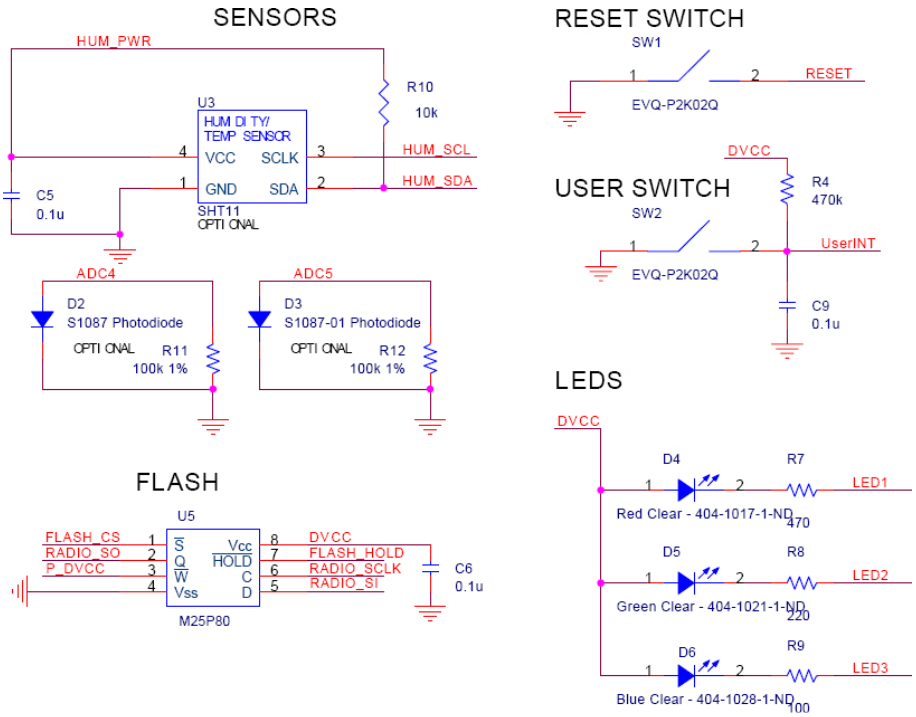

Figure 2: TelosB sensor, switch, and LED circuit diagram[2]

```
/* ... */
#define LEDS_CONF_RED     0x10 /* Program pin 4 (00010000) */
#define LEDS_CONF_GREEN   0x20 /* Program pin 5 (00100000) */
#define LEDS_CONF_YELLOW  0x40 /* Program pin 6 (01000000) */
/* ... */
void
leds_arch_init(void)
{
  /* Set I/O port 5 to output data */
  P5DIR |= (LEDS_CONF_RED | LEDS_CONF_GREEN | LEDS_CONF_YELLOW);
  /* Program I/O port 5, pins 4-6, i.e. set the LEDs to light up */
  P5OUT |= (LEDS_CONF_RED | LEDS_CONF_GREEN | LEDS_CONF_YELLOW);
}
```

It has already been mentioned that, ideally, no changes are needed to the programs except for when developing low-level software. Since the CC2420 radio chip driver has already been implemented, essentially certain variables needed to be redefined to program the appropriate pins for a given operation. See below for example code taken from `platform/telosb/contiki-conf.h`.

```
#define FIFO_P          0  /* P1.0 - Input: FIFOP from CC2420 */
#define FIFO            3  /* P1.3 - Input: FIFO from CC2420 */
#define CCA             4  /* P1.4 - Input: CCA from CC2420 */
```

```
#define SFD             1  /* P4.1 - Input:  SFD from CC2420 */
#define CSN             2  /* P4.2 - Output: SPI Chip Select (CS_N) */
#define VREG_EN         5  /* P4.5 - Output: VREG_EN to CC2420 */
#define RESET_N         6  /* P4.6 - Output: RESET_N to CC2420 */
/* ... */
```

Besides the LEDs and RF chip, the TelosB has many more peripherals in stock; two buttons, namely the user switch and reset switch, which,when pushed can be used to perform a desired function or a system reset, respectively. `platform/telosb/dev/button-sensor.c` implements the interface defined in `core/lib/sensors.h`. From Figure 2 we see that the user switch is activated by UserINT, which, if we go back to Figure 1, corresponds to P2.7. The two functions below are defined as macros found in `cpu/msp430/dev/hwconf.h`, which specifies the CPU-specific methods for manipulating IRQ and pin values, such as `_ENABLE_IRQ()` or `_READ()` for reading the value of a given pin.

```
HWCONF_PIN(BUTTON, 2, 7);
HWCONF_IRQ(BUTTON, 2, 7);
```

With the defined macros given in `hwconf.h`, we can, for example, write the `init()` function for a given sensor. Some brief sample code is shown below, along with the expanded macro—in the case of the two function calls above—and description in comments.

```
static void
init(void)
{
  /* Port 2 Interrupt Enable */
  BUTTON_IRQ_EDGE_SELECTD();   /* P2IES |= 1 << bit; */
  /* Port 2 Selection */
  BUTTON_SELECT();             /* P2SEL &= ~(1 << bit); */
  /* Port 2 Direction */
  BUTTON_MAKE_INPUT();         /* P2DIR &= ~(1 << bit); */
}
```

Finally, we will look at the driver implementation for the sensors on the TelosB platform. Some sample source code for `sensors\_light\_init()` is shown below.

```
void
sensors_light_init(void)
{
  P6SEL |= 0x30;   /* Activate port 6, pins 4 and 5 (ADC4 and ADC5) */
  P6DIR = 0xff;    /* Read data from port 6 */
  P6OUT = 0x00;    /* Initialize readings to zero */

  /* Set up the 12-bit ADC. */
  /* Adapted TinyOS source code for Contiki
```

```
      please refer to the TOS repository:
        tos|chips|msp430|adc12*/
  /* ADC12 Control 0-Setup ADC12, ref., sampling time */
  ADC12CTL0 = REF2_5V + SHT0_6 + SHT1_6 + MSC;
              /*  ADC12 Ref 0:1.5V / 1:2.5V - reference voltages,
                  Sample and hold ref0:128 samples,
                  Sample and hold ref1:128 samples,
                  ADC12 Multiple Sample Conversion */
  /* ADC12 Control 1-Use sampling timer, repeat-sequence-of-channels */
  ADC12CTL1 = SHP + CONSEQ_3 + CSTARTADD_0;
              /*  ADC12 Sample/Hold Pulse Mode,
                  Repeat-sequence-of-channels,
                  ADC12 Conversion Start Address 0 */

  ADC12MCTL0 = (INCH_4 + SREF_0); /* Photodiode 1, VR+ = AVCC and VRŨ = AVSS */
  ADC12MCTL1 = (INCH_5 + SREF_0); /* Photodiode 2, VR+ = AVCC and VRŨ = AVSS */

  ADC12CTL0 |= ADC12ON + REFON; /* ADC12 On/enable, ADC12 Reference on */

  ADC12CTL0 |= ENC;       /* ADC12 Enable Conversion */
  ADC12CTL0 |= ADC12SC; /* ADC12 Start Conversion */
}
```

4.2 Porting Contiki to the MICAz

As with the TelosB, the kernel and the service layer did not require any changes during the porting process.[8]
Also, as before, the MCU circuit diagram in Figure 3 will be referred to in order to identify the appropriate pin numbers for given low-level operations. Moreover, the optional attachable sensor boards are interfaced through the 51-pin connector on the MICAz. Since the MTS300 was not available during the last stages of porting, the sensor device drivers have not been implemented.
Nevertheless, the Contiki interface for LEDs found in `core/dev/leds.h` was fully implemented. This process was, of course, very similar to that of the TelosB. However, as we can see in 3, the three LEDs are connected to pins 49-51, Port A Pin 2-Pin0, respectively.
In addition to this, it should be noted that on the AVR platform a bit is programmed if it is set to '0'. Thus, in order to program LED3, for example, one needs to send the byte string $0x06$ to Port A, $0x05$ to turn on LED2, and $0x03$ to turn on LED1. A small part of the source code file implementing the leds.h interface can be seen below.

```
void
leds_on(unsigned char leds)
{
  if (leds == LEDS_GREEN)
```

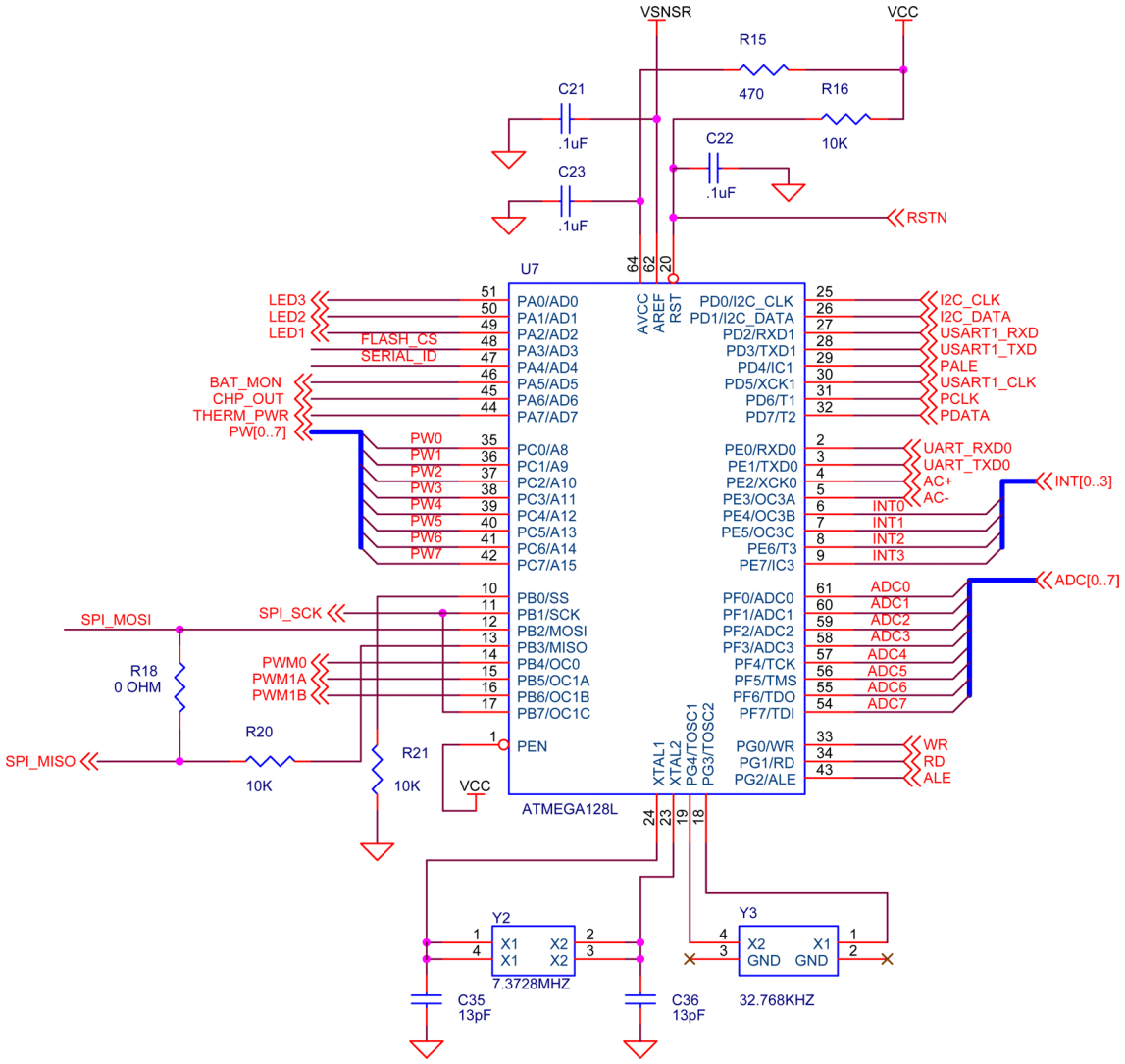

Figure 3: MICAz MCU circuit diagram[1]

```
    leds = LEDS_CONF_GREEN;     /* 0x05 */
  if (leds == LEDS_YELLOW)
    leds = LEDS_CONF_YELLOW;    /* 0x06 */
  if (leds == LEDS_RED)
    leds = LEDS_CONF_RED;       /* 0x03 */
  if (leds == LEDS_ALL)
    leds = 0;
  LEDS_PxOUT &= l2p[leds & LEDS_ALL];
}
```

`l2p` in this case is an `unsigned char` array that consists of all (eight) possible states of the LEDs. `LEDS_PxOUT`, as can be determined from Figure 3, is, in fact, `PORTA`, which is defined in the system AVR header files.

5 Results

After the Contiki core was successfully ported to the TelosB and MICAz platforms, it had to be tested whether or not they were actually functioning. A simple USB I/O system was successfully implemented for the TelosB through which messages could be passed between the host computer and the WSD. Since this was not successfully implemented for the MICAz, the LED API was implemented to determine whether or not the core, and, more specifically, that Blink was in fact able to run on the system.

Nevertheless, in addition to blinking lights, the Contiki port to TelosB features over-the-air programming and dynamic program loading, a feature that Contiki is renowned for. In order to test this functionality, a client and gateway kernel were implemented which initialize different processes and peripherals, e.g. the client core does not have to initialize the USB interface.

Contiki comes with very useful development tools. Two of these programs, namely Tunslip and Codeprop, are used to connect to a WSD via USB, and to upload `.ce` files to a Contiki system during runtime, respectively. The TelosB testing environment consisted of two WSDs, one with the gateway core and the other with the client core uploaded and running. Using Tunslip, a SLIP connection was made to the gateway via USB. The gateway obviously had two network addresses, one for the SLIP connection, and one for wireless. When sending an application to a given node via Codeprop, the data would first be shipped to the gateway, which would then forward it to the appropriate node. More specifically, this was done with the 'Blink' application, in order to determine whether or not the nodes could communicate via wireless and make each other blink.

The MICAz, on the other hand, could only be tested with the implemented LED interface.

6 Conclusion

The Contiki operating system has been successfully ported to the TelosB platform, and the device drivers for onboard peripherals have been implemented fully. Adam Dunkels, the main developer of Contiki OS, claimed that, "Ideally, no changes are needed to the programs except for when developing low-level software." As most of the low-level software was already implemented, i.e. software to drive a CC2420 RF transceiver as well as CPU-specific code for AVR and MSP430 microcontrollers, the porting process was reasonably straightforward. The build system allows for seamless integration of already existing (even low-level) code; one way that this is achieved is by defining interfaces in the subdirectories of `core`. Platform-specific code then is defined in the folder itself. An example of such a case is defining a function that initializes the CPU/microcontroller, which is universally defined as `cpu_init()`, but is obviously platform-specific.

Also higher level applications, such as Blink, were successfully compiled for the various platforms without the need to change any application code.

As the MICAz port was not completely finished, this could be a future goal, namely to fully integrate the SLIP interface, and test the functionality of the wireless and dynamic program loader.

References

[1] Crossbow, *Micaz datasheet.*

[2] ———, *Telosb datasheet.*

[3] Adam Dunkels, *The contiki operating system — documentation*, 2007, [Online; accessed May-2007].

[4] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt, *Run-time dynamic linking for reprogramming wireless sensor networks*, SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems (New York, NY, USA), ACM Press, 2006, pp. 15–28.

[5] Adam Dunkels, Björn Grönvall, and Thiemo Voigt, *Contiki - a lightweight and flexible operating system for tiny networked sensors*, Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I) (Tampa, Florida, USA), November 2004.

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

[7] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava, *A dynamic operating system for sensor nodes*, MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services (New York, NY, USA), ACM Press, 2005, pp. 163–176.

[8] Alexandru Stan, *Porting the core of the contiki operating system to the telosb and micaz platforms*, 2007.