Porting TinyOS on to BTnodes

Attila Dogan-Kinali

Winter Term 2004/2005

Student Thesis SA-2005-07

Supervisors: Jan Beutel, Matthias Dyer, Prof. Dr. Lothar Thiele

Contents

Figures

1 Introduction

In recent days, more and more measurements are conducted in areas where no sufficient infrastructure exist. Particularly for power supplies and for the transmission of collected data. Wireless sensor networks can be of use in these situations. They consist of small, low-cost devices that are deployed in large numbers in the target area. These devices, also called nodes, organize themselves in a *ad hoc* network to exchange messages containing measurements and other data. The receiver part is commonly represented by a root or master node.

Nodes must not disturb their environment, and with it the measured data. They have to be as unintrusive as possible which requires them to be as small as possible. Because nodes must be present in large numbers, they must be as cheap as possible. These two constraints pose, among other things severe restrictions on the computational power and radio transmission capabilities. Because of this and because obstacles may be in the way, it's not always possible that all nodes can directly send their data to the desired receiver. But since they come in large numbers and the distances between two adjacent nodes is small, the nodes themselves can be used to relay messages from one to another.

Currently a number of universities are using different approaches to explore the field of wireless sensor networks. These approaches do not only result in different hardware architectures for the nodes, but also in different architectures of the operating systems running on these nodes. One of the big players in the field of operating systems is TinyOS which is developed at the University of Berkeley.

TinyOS[5][16] was designed with modularity and scalability in mind. Both was achieved by rigorously decomposing the operating system into small building blocks. Trough this partitioning TinyOS became more a system frame work for embedded applications than an operating system in the classical sense. But this partitioning is what makes TinyOS scale, not only up, but also down to smaller devices. Various applications were developed using TinyOS, like area monitoring, location tracking[17] or even small robots[3].

The goal of this semester project at TIK[1] was to evaluate TinyOS as an alternative operating system on the BTnode[1][12] platform. The BTnodes, which were developed here at TIK are currently running BTnut, an Ethernut derived operating system only. The main subjects were to find out whether and what TinyOS applications can be run on BTnodes and how portable TinyOS is.

As a starting point, the TinyBT[8] project from the University of Copenhagen was used. TinyBT is a port of TinyOS to the second revision of the BTnode platform, never the less it includes also preliminary support for the in this project used, newer BTnode revision. As strategy it was chosen to use various applications to the new platform. By starting from simple applications that use only a small fraction of TinyOS and going to more complex ones, more and more of TinyOS has been ported.

[1]Computer Engineering and Networks Laboratory (TIK) at the Department of Information Technology and Electrical Engineering of the Swiss Federal Institute of Technology (ETH) Zürich `http://www.tik.ee.ethz.ch`

2

Related Work

Various operating systems have been developed in the past for embedded applications. QNX[10], which is one of the most used operating systems for embedded systems is a fully featured POSIX compliant real time operating system. Although it was designed with embedded systems in mind, it does not scale down well to the region of 8 bit systems used in wireless sensor networks. uClinux[9] is a derivative of the Linux kernel optimized for systems without a memory management unit. It is like QNX meant as a POSIX compliant operating system for small devices and has similar properties in scalability.

nesC explores on the language side a component model which was made popular by M. Flatt and M. Felleisen[13]. Although some research was conducted in this field, only a few languages were developed and used. Most notable exception is the Knit[11] language which is also an extension to the C language. Like nesC it was designed for research in the operating system field, namely the OSKit Project[4] of the University of Utah.

3

TinyOS

TinyOS has been developed as a component based operating system to experiment with ad hoc wireless networks based on a hardware platform called ”Mica2 Motes”[2] at the UC Berkeley. It uses an event driven approach, meaning that every processing step is triggered by some kind of event. Every triggered ”task” is enqueued into a worker queue and processed by the main loop until none is left. After that TinyOS waits for the next event to occur. TinyOS is written in nesC[15][14], a C-based, component model oriented language, and is also developed at Berkeley. To simplify the building process, the whole operating system and the applications are built together in one step. The nesC compiler combines all components of the application and operating system and builds a C file which is passed to a C compiler.

3.1 nesC

Component model languages allow to decompose an application into building blocks. These building blocks are connected together at compile time, using some handwritten description. Although generally speaking this works similar to the object orientated approach it does not have the overhead of runtime decisions and does not require memory management system as the components and their memory are instantiated at compile time. This means that it is impossible to instantiate additional components at run time though. Albeit this may seem like a severe restriction, it is not as the target platforms are embedded systems with very limited resources that do not allow the same programming style as on normal, bigger sized computers.

nesC splits the components of an application into the entities *interface*, *configuration* and *module*. The *interfaces* define an abstract way of accessing different modules with a consistent calling convention comparable to the interfaces used in the Java programming language. *Configurations* are instructions on how to wire the used components together. Typically there is one configuration file for every component interconnected to at least one other component. The *module* contains the actual implementation of the component in a language that is basically C with additional keywords and case-related slightly modified semantics. These three entities

```
configuration LedsC {
  provides interface Leds;
}
implementation
{
     components PowerStateM, LedsM;
     Leds = LedsM.Leds;
     LedsM.PowerState -> PowerStateM;
}
```

Listing 3.1
tos/platform/pc/LedsC.nc

```
interface Leds {
  async command result_t init();
  async command result_t redOn();
  async command result_t redOff();
  async command result_t redToggle();
  async command result_t greenOn();
  async command result_t greenOff();
  async command result_t greenToggle();
  async command result_t yellowOn();
  async command result_t yellowOff();
  async command result_t yellowToggle();
  async command uint8_t get();
  async command result_t set(uint8_t value);
}
```

Listing 3.2
tos/interfaces/Leds.nc

are, by convention, split into 3 different files, called `<name>.nc` for the *interface*, `<name>C.nc` for the *configuration* and `<name>M.nc` for the *module*.

As an example the `Leds` component (Listing 3.1) provides routines to access the LEDs of a node. The *configuration* file of it (`LedsC.nc`) implements the `Leds` interface and uses the `PowerStateM` and `LedsM` components. The internally used symbol `Leds` references to `LedsM` and the `LedsM.PowersState` symbol, which denotes an internal symbol of `LedsM` is connected to `PowerStateM`. The `Leds` *interface* definition (Listing 3.2) states various function, or in this case command prototypes, similar to header files in C. The *module* definition of the `Leds` component partly shown in Listing 3.3 lists the implemented and used interfaces in the with brackets enclosed block labeled ”`module`”

The additional keywords and semantics allow the nesC compiler to perform simple static race condition analysis.

```
module LedsM {
  provides interface Leds;
  uses interface PowerState;
}
implementation
{
  uint8_t ledsOn;

  enum {
    RED_BIT = 1,
    GREEN_BIT = 2,
    YELLOW_BIT = 4
  };

  void updateLeds() {
    LedEvent e;
    e.red    = ((ledsOn & RED_BIT) > 0);
    e.green  = ((ledsOn & GREEN_BIT) > 0);
    e.yellow = ((ledsOn & YELLOW_BIT) > 0);
    sendTossimEvent(NODE_NUM,
      AM_LEDEVENT, tos_state.tos_time, &e);
  }

  async command result_t Leds.init() {
    atomic {
      ledsOn = 0;
      dbg(DBG_BOOT, "LEDS: initialized.\n");
      updateLeds();
    }
    return SUCCESS;
  }
 [...]
}
```

Listing 3.3
tos/platform/pc/LedsM.nc

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Listing 3.4
apps/Blink/Blink.nc

3.2 The Directory Structure

The TinyOS distribution consists beside of the so called ”`tos`” directory, which contains the whole operating system and all platform definitions also some standard applications (`apps`) which serve as examples on how to use different parts of TinyOS. The `tos` directory itself consists of the platform independent operating system part (`system`), common interface definitions (`interfaces`), a library with commonly used functions (`lib`), platform dependant hardware definitions and access driver functions (`platform`) and definitions for the different sensor boards that can be used in combination with the motes (`sensorboards`).

The shipped interface definitions cover a wide range of available hardware for wireless sensor systems (ADC, UART, I2C, Timer, ...), some abstraction for communication classes (byte level abstraction, packet level abstraction) and for various, often used software systems (scheduler, resource controller, routing, memory management).

The `system` directory contains generic implementations of various hardware components (ADC, I2C, Clock), hardware independent core functions (CRC, framing, logger, operating system initialization) and a few utility components.

Various high level functions are provided trough ”libraries” in the `lib` directory.

Files in the different directories are superseded by files with the same name in other directories. But this functionality is not documented.

3.3 Boot Strapping and Hardware Initialization

The boot up of a simple application like for example Blink (Listing 3.4, see also Section 7.1), which uses only very few components follows a simple scheme. The boot up process of TinyOS starts in `system/RealMain.nc` (Listing 3.6) calling `hardwareInit()` which is connected to `HPLInit` (Listing 3.7) via the `Main` component (Listing 3.5). `HPLInit` does nothing but call `TOSH_SET_PIN_DIRECTIONS()` from `hardware.h` (Listing 3.8, see also Section 3.4) which calls macros to set the direction registers of the AVR microcontroller. After calling `Pot.init()` which initializes the potentiometer module, `StdControl.init()` and `StdControl.start()` are called. These are commandos that dispatch to all connected modules. In this case

```
configuration Main {
  uses interface StdControl;
}
implementation
{
  components RealMain, PotC, HPLInit;

  StdControl = RealMain.StdControl;
  RealMain.hardwareInit -> HPLInit;
  RealMain.Pot -> PotC;
}
```

Listing 3.5
tos/system/Main.nc

```
module RealMain {
  uses {
    command result_t hardwareInit();
    interface StdControl;
    interface Pot;
  }
}
implementation
{
  int main() __attribute__ ((C, spontaneous)) {
    call hardwareInit();
    call Pot.init(10);
    TOSH_sched_init();

    call StdControl.init();
    call StdControl.start();
    __nesc_enable_interrupt();

    while(1) {
      TOSH_run_task();
    }
  }
}
```

Listing 3.6
tos/system/RealMain.nc

```
module HPLInit {
  provides command result_t init();
}
implementation
{
  // Basic hardware init.
  command result_t init() {
    TOSH_SET_PIN_DIRECTIONS();
    return SUCCESS;
  }
}
```

Listing 3.7
tos/platform/btnode3_2/HPLInit.nc

```
// LED assignments
TOSH_ASSIGN_PIN(RED_LED, A, 2);
TOSH_ASSIGN_PIN(GREEN_LED, A, 1);
TOSH_ASSIGN_PIN(YELLOW_LED, A, 0);

// ChipCon control assignments
TOSH_ASSIGN_PIN(CC_CHP_OUT, A, 6);
TOSH_ASSIGN_PIN(CC_PDATA, D, 7);
TOSH_ASSIGN_PIN(CC_PCLK, D, 6);
 [...]
void TOSH_SET_PIN_DIRECTIONS(void)
{
  TOSH_MAKE_CC_CHP_OUT_INPUT();
 [...]
  TOSH_MAKE_CC_PALE_OUTPUT();
  TOSH_MAKE_CC_PDATA_OUTPUT();
  TOSH_MAKE_CC_PCLK_OUTPUT();
  TOSH_MAKE_SPI_OC1C_INPUT();
}
```

Listing 3.8
tos/platform/mica2/hardware.h

```
 [...]
bool TOSH_run_next_task ()
{
  __nesc_atomic_t fInterruptFlags;
  uint8_t  old_full ;
  void (*func)(void);

  fInterruptFlags = __nesc_atomic_start();
   old_full  = TOSH_sched_full;
  func = TOSH_queue[old_full].tp;
  if  (func == NULL)
    {
      __nesc_atomic_end(fInterruptFlags);
      return 0;
    }

  TOSH_queue[old_full].tp = NULL;
  TOSH_sched_full = (old_full + 1) & TOSH_TASK_BITMASK;
  __nesc_atomic_end(fInterruptFlags);
  func();

  return 1;
}

void TOSH_run_task() {
  while (TOSH_run_next_task())
    ;
  TOSH_sleep();
  TOSH_wait();
}
```

Listing 3.9
tos / system / sched.c

`SingleTimer.StdControl` and `BlinkM.StdControl`. After enabling interrupts, TinyOS enters its main loop in which it calls `TOSH_run_task()` from `sched.c` (Listing 3.9) repeatedly. This function processes all pending tasks in the queue until none is left and sleeps afterwards until an IRQ occurs.

3.4 Platform Definitions

The `platform` directory contains a subdirectory for each supported hardware platform, where files specific to this platform can be found. Two files have to be present at least in each platform definition, `hardware.h` and `.platform`.

`.platform` (Listing 3.10) is a perl script used by the nesC compiler to set several architecture specific options, parameters for the C compiler and internal variables.

```
@opts = ("-gcc=avr-gcc",
         "-mmcu=atmega128",
         "-fnesc-target=avr",
         "-fnesc-no-debug");

push @opts, "-mingw-gcc" if $cygwin;

@commonplatforms = ("btnode3_2");
```

Listing 3.10
tos/platform/btnode3_2/.platform

```
#define TOSH_ASSIGN_PIN(name, port, bit) \
static inline void TOSH_SET_##name##_PIN() \
                   {sbi(PORT##port , bit);} \
static inline void TOSH_CLR_##name##_PIN() \
                   {cbi(PORT##port , bit);} \
static inline char TOSH_READ_##name##_PIN() \
                   {return 0x01 & (inp(PIN##port) >> bit);} \
static inline void TOSH_MAKE_##name##_OUTPUT() \
                   {sbi(DDR##port , bit);} \
static inline void TOSH_MAKE_##name##_INPUT() \
                   {cbi(DDR##port , bit);}

#define TOSH_ASSIGN_OUTPUT_ONLY_PIN(name, port, bit) \
static inline void TOSH_SET_##name##_PIN() \
                   {sbi(PORT##port , bit);} \
static inline void TOSH_CLR_##name##_PIN() \
                   {cbi(PORT##port , bit);} \
static inline void TOSH_MAKE_##name##_OUTPUT() \
                   {;}
```

Listing 3.11
tos/platform/btnode3_2/avrhardware.h

It is executed by `ncc` which is the first step of the nesC compiler.

`hardware.h` (Listing 3.8 and Listing 6.1) is used to assign functions to the pins of the microcontroller. The macros `TOSH_ASSIGN_PIN` and `TOSH_ASSIGN_OUTPUT_ONLY_PIN` which are defined in `avrhardware.h` (Listing 3.11) and are used for this. These two macros define functions to set, clear, read the pin, to make it an output or to make it an input port. These functions are called at boot time trough `TOSH_SET_PIN_DIRECTIONS()` and various components providing access to the hardware and thus need the symbolic names of the associated pins. `TOSH_SET_PIN_DIRECTIONS()` is often not only used to set the directions of pins but also to do some preliminary initialization of the hardware before the first TinyOS components are used.

The `platform` directory contains also components to access hardware features and I/O subsystems specific to the hardware platform, like the components for the various radio modules, ADCs, EEPROMs, special timers and the like. This directory is also used to mask components in the `system` and `lib` directories, if special implementations are required, like for example the `LedsC` component for the PC architecture.

3.5 Toolchain

The compilation of TinyOS requires the avr-gcc and nesC toolchain to be installed. As this can be quite a hassle due to the dependency on specific versions. Thus it's suggested to use the packages provided on the TinyOS website.

nesC has been implemented as a wrapper around gcc using a few scripts written in perl. As its development is inseparable related to TinyOS, it has some dependencies on the existance of certain files and structures. Though there is a remark in the `ncc` manpage on how to call the nesC compiler in an enviroment outside TinyOS, it is not clear how well this will work due to the lack of proper documentation of the inner working of the nesC compiler. Also some of its configuration options are set in the beginning of the ncc file, which may lead to strange error messages if not set properly. The installation of nesC will only succeed if everything is in place as it expects it to be and thus the procedure should be followed exactly as it is described on the website[6], otherwise strange errors will occur.

4

Hardware Platforms

In this project only two of the various available hardware projects were used. One of them is the ”Mica2 Mote” developed at the University of Berkeley being the main development platform for TinyOS. Secondly BTnode revision 3 was used, developed at TIK.

4.1 Mica2 Mote

The ”Mica2 Mote” (Figure 4-1) which was developed as a replacement for the older ”Mica Mote”, is a node build around an Atmel ATmega128L 8 bit microcontroller and features an ChipCon1000 low power radio module, and a 512 kB Flash module. Beside the general purpose I/O pins it also has two serial interfaces (UART).

4.2 BTnode

BTnode family started out as a pure, all-in-one BlueTooth module but became a fully featured dual radio wireless networking platform. Its newest member, the

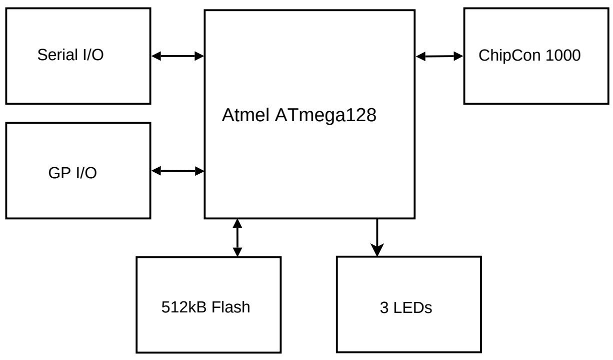

Figure 4-1
Mica2 Mote hardware structure

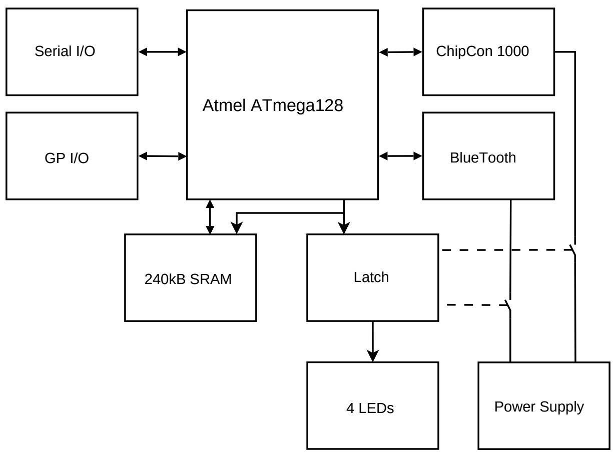

Figure 4-2
BTnode hardware structure

BTnode revision 3 does not only come equipped with a BlueTooth module, but also features a ChipCon1000 module in a very similar configuration as the ”Mica2 Motes”. Around an Atmel ATmega128L microcontroller, the mentioned ChipCon1000 low-power radio module, a Zeevo ZV 4002 BlueTooth module, 4 LEDs and various I/O interfaces (Figure 4-2). 240 kB of banked SRAM[1] are available, of which 60 kB are always visible and can be used with out any modifications of the application. The BTnode allowes to completely shutdown all radio modules by a switch in their power lines, which gives additional power saving opportunities.

4.3 Differences between Mica2 and BTnodes

Platform	**BTnode revision 3**	**Mica2 Mote**
BlueTooth	Zeevo ZV4002	no
Low-power radio	CC1000	CC1000
Data Memory	64 kB	4 kB
Storage	180 kB SRAM	512 kB Flash
LEDs	4	3
Serial ID	no	yes
Debug UART	UART0	UART0, UART1
Connector	Molex 15 Pin or Hirose DF17 40 Pin	Hirose DF9 51 Pin
Regulated power supply	yes, 2xAA cells or DC input	no
Switchable power for radios and extensions	yes	no

Table 4-1: BTnode revision 3 hardware details compared to the Mica2 Mote.

[1]the lowest 4 kB of each bank are masked by the internal SRAM

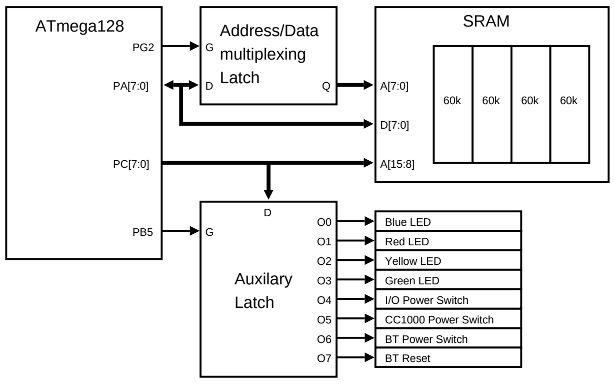

Figure 4-3
Detail view of the Latches

The differences between the ”Mica2 Motes” and BTnodes are mainly, that the BTnodes not only carries a ChipCon1000 low power radio but also a BlueTooth module. The external SRAM provides, even when the additional banks are not used, additional 60 kB of memory, so that the full adress range of 64 kB can be used, even without any modification of the application. There are also two latches to multiplex two ports of the microcontroller (Figure 4-3). The ”Address/Data multiplexing Latch” is used to multiplex port `PA` as data and the lower half of the address lines for the SRAM. The gate pin of this latch is connected to the pin `PG2` of the ATmega128 which is directly controlled by the internal adress generation unit, whenever the address range of the external SRAM is accessed. The ”Auxilary Latch” multiplexes the upper half of the address lines of the SRAM with the LEDs, the power switches and the reset pin of the BlueTooth module.

5 *Latch Problems*

As noted in the previous section, the BTnodes contains two latches to multiplex ports of the microcontroller. As the "Address/Data multiplexing Latch" controlled by the microcontroller directly is of no concern in this paper it wil not be refered anymore. Instead, the "Auxilary Latch", which multiplexes the upper half of the address lines of the external SRAM and among other things the LEDs, will be refered as "latch" only. As this port is also used for the adress lines of the external SRAM, the latch will be intransparent most of the time in normal operating mode. Hence every time either one of the LEDs or power switches are accessed the latch has to be put into transparent mode and switched back to intransparent afterwards[1].

While porting the `Blink` application (see Section 7.1) it was seen that the LEDs did not work as expected. Particularily, that the LEDs after being switched on, could not be switched off again. Further investigation showed that the LEDs worked properly when the latch was always transparent or the transparent gate time was very long. It was found that the reason for this was an unexpectedly high settling time of the latch outputs when doing a transition of the output pin from *high* to *low*. Figure 5-1 shows a measurement of this behaviour. In the test setup the latch output pin was set to *high* at the beginning. Before asserting Latch Enable, the input pin was set to *low*. As can be seen, it takes several 100 milliseconds until the latch output settles to *low*. Measurements on different BTnodes showed a variation of this settling time between 200ms and 700ms. Due to the limited access to the latch on the circuit board not all relevant signals could be measured, hence the cause of this strange behaviour is still unclear.

As no hardware solution could be found, it was choosen to set the latch permanently into transparent mode on start up. Although this renders the external SRAM module useless, it is of no importance for this project as the TinyOS port does only use the 4 kB of internal RAM.

[1]Note that during this operation any access to the external SRAM has to be avoided

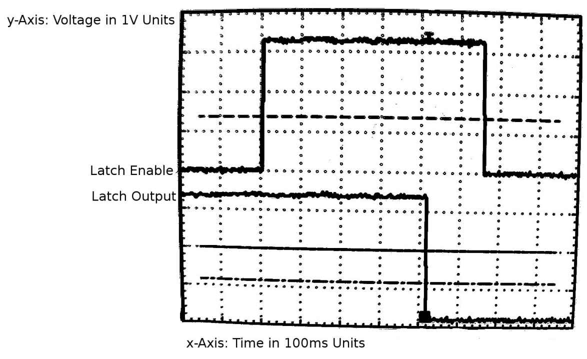

Figure 5-1
Latch switching anomaly. After setting the latch input, latch enable is asserted. The settling time for the latch output is over 400ms

6

TinyOS on BTnode

Although the current BTnodes are similar to the ”Mica2 Motes”, they are not completely the same. Thus there is some work required to port the existing code to the new architecture. This was also used as an oppurtunity to test the portability of TinyOS, although it is only a partial test, due to the similarity to an already existing platform.

As a starting point for porting the preliminary `btnode3_2` platform definition from the TinyBT project of the University of Copenhagen was used. This platform definition was originaly written by Jan Beutel for a short test during the development of the current BTnode generation and thus only necessary steps to get a compiling system were performed. Most of the platform specific settings are collected in `hardware.h` (Listing 6.1). The most notable change in this file, beside the different pin definitions, is the special handling for the latch which has to be enabled to set one of the LEDs or to toggle one of the power switches of the radio modules. But, due to hardware problems of unknown origin, this had to be partially disabled (see Section 5).

At the beginning of `hardware.h` the `TOSH_ASSIGN_PIN` macro is called to give the ChipCon1000 and latch control pins symbolic names. Setting any pins of the latch requires some care, as always all 8 pins are set at the same time. This is taken care of by the `set_latch()` function. It allows to set a single bit of the latch by giving its position and the new value. The current status of the latch is stored in the `latch_status()` variable. Because the TinyOS components expect functions with certain names to access the LEDs a couple of wrappers around `set_latch` are defined. At the bottom of `hardware.h` is the `TOSH_SET_PIN_DIRECTIONS()` function which is used at boot up to do a preliminary initialization of the pin direction registers of the microcontroller. It first initializes all registers with zeros, which sets all pins to the input mode and then calles the above defined functions to make the appropirate ones to output pins.

Most of the other files in the platform definition are coppied from either the Mica2 or TinyBT definitions and adjusted to the small differences of the BTnode revision 3.

```
 [...]
// ChipCon control assignments
TOSH_ASSIGN_PIN(CC_CHP_OUT, E, 7); // chipcon CHP_OUT
TOSH_ASSIGN_PIN(CC_PDATA , D, 7); // chipcon PDATA
TOSH_ASSIGN_PIN(CC_PCLK , D, 6);   // chipcon PCLK
TOSH_ASSIGN_PIN(CC_PALE , D, 5);   // chipcon PALE

TOSH_ASSIGN_PIN(LATCH_SELECT, B, 5);
 [...]
/* set leds and power functions */
unsigned char latch_status;
/* set function, not irq safe */
void set_latch(int pos, int bit)
{
        unsigned char mask = 1;
        mask <<= pos;
        latch_status = (latch_status & ~mask) | (bit ? mask : 0);
        outb(PORTC, latch_status);
// latch is always enabled due to hardware problems
//      TOSH_SET_LATCH_SELECT_PIN();
//      TOSH_wait(); // wait a short while until the latch is updated
//      TOSH_CLR_LATCH_SELECT_PIN();
}

static inline void TOSH_SET_YELLOW_LED_PIN() {set_latch(2,1);}
static inline void TOSH_CLR_YELLOW_LED_PIN() {set_latch(2,0);}
#define TOSH_MAKE_YELLOW_LED_OUTPUT()
static inline void TOSH_SET_GREEN_LED_PIN() {set_latch(3,1);}
static inline void TOSH_CLR_GREEN_LED_PIN() {set_latch(3,0);}
#define TOSH_MAKE_GREEN_LED_OUTPUT()
static inline void TOSH_SET_RED_LED_PIN() {set_latch(1,1);}
static inline void TOSH_CLR_RED_LED_PIN() {set_latch(1,0);}
#define TOSH_MAKE_RED_LED_OUTPUT()
static inline void TOSH_SET_EXTRA_LED_PIN() {set_latch(0,1);}
static inline void TOSH_CLR_EXTRA_LED_PIN() {set_latch(0,0);}
#define TOSH_MAKE_EXTRA_LED_OUTPUT()
// CC1000 power control
static inline void TOSH_SET_CC_PWR_PIN() {set_latch(5,1);}
static inline void TOSH_CLR_CC_PWR_PIN() {set_latch(5,0);}
void TOSH_SET_PIN_DIRECTIONS(void)
{
  outp(0x00, DDRA);
  outp(0x00, DDRB);
  outp(0x00, DDRD);
  outp(0x02, DDRE);
  outp(0x02, PORTE);
  TOSH_MAKE_LATCH_SELECT_OUTPUT();
  TOSH_SET_LATCH_SELECT_PIN();
  TOSH_SET_CC_PWR_PIN();
}
```

Listing 6.1
tos/platform/btnode3_2/hardware.h

7

The TinyOS Standard Applications

Before any applications can be compiled for TinyOS a few settings have to be done. Foremost, the `TOSDIR` enviroment variable has to be set to the location of the ”`tos`” directory (e.g. `TOSDIR=$HOME/src/tinyos-1.x/tos`).

The compilation of one of the standard applications is done by switching into the directory in `apps` directory, eg `apps/Blink` and executing the `make` command with the desired platform as a parameter (e.g. `make btnode3_2`). This will create a file `main.exe` in a platform specific subdirectory of the `build` directory.

For the next step, the programming of the BTnode, the `AVRISP` enviroment variable has to be set to the serial interface to which the programmer is connected to (e.g. `AVRISP=/dev/ttyS0`). The targed is programmed by calling `make install` with the platform as parameter (e.g. `make install btnode3_2`). Note that the application is compiled once again when executing this command.

7.1 Blink

The `Blink` (Listing 3.4) application is a version of the well known ”Hello World” program for a hardware architecture that does not have any character output. It is a simple application that only sets up a timer to toggle a LED on and off. Thus it is not much more than an example application to show that TinyOS really runs on a given platform. The *configuration* (Listing 3.4) connects the four used components, the `Main`, `BlinkM` (Listing 7.1), `SingleTimer` and `LedsC` together. Listing 7.1 shows that the *module* provides the `StdControl` *interface* and uses the *interfaces* `Timer` and `Leds`. Upon boot up the command `StdControl.init()` will be called which initializes the `Leds` module. The later called command `StdControl.start()` sets the timer to trigger an event every second. This timer will then call the event `Timer.fired` which toggles the red LED.

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

Listing 7.1
apps/Blink/BlinkM.nc

```
configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;

  Main.StdControl −> Counter.StdControl;
  Main.StdControl −> IntToLeds.StdControl;
  Main.StdControl −> IntToRfm.StdControl;
  Main.StdControl −> TimerC.StdControl;
  Counter.Timer −> TimerC.Timer[unique("Timer")];
  IntToLeds <− Counter.IntOutput;
  Counter.IntOutput −> IntToRfm;
}
```

Listing 7.2
apps / CntToLedsAndRfm / CntToLedsAndRfm.nc

```
configuration RfmToLeds {
}
implementation {
  components Main, RfmToInt, IntToLeds;

  Main.StdControl −> IntToLeds.StdControl;
  Main.StdControl −> RfmToInt.StdControl;
  RfmToInt.IntOutput −> IntToLeds.IntOutput;
}
```

Listing 7.3
apps / RfmToLeds / RfmToLeds.nc

7.2 CntToLedsAndRfm and RfmToLeds

`CntToLedsAndRfm` (Listing 7.2) and `RfmToLeds` (Listing 7.3) together form an application to test the radio subsystem of TinyOS. `CntToLedsAndRfm` counts upwards and shows the number in binary form on the LEDs. Additionaly the number is send over the radio interface out. `RfmToLeds` recieves the packets, extracts the number and displays it the same way on the LEDs. Although these applications are still very simple, they use a substantial part of TinyOS over the instanciated components.

7.3 Surge

The `Surge` application builds up an ad hoc network of motes to transmit measurment data from the nodes to the root node which is connected to a PC that displays the data (Figure 7-1). In this case it was used to demonstrate that it is possible to communicate between Mica2 motes and BTnodes when using the ChipCon1000 module.

Because the BTnode platform does not have any available sensor boards yet, every

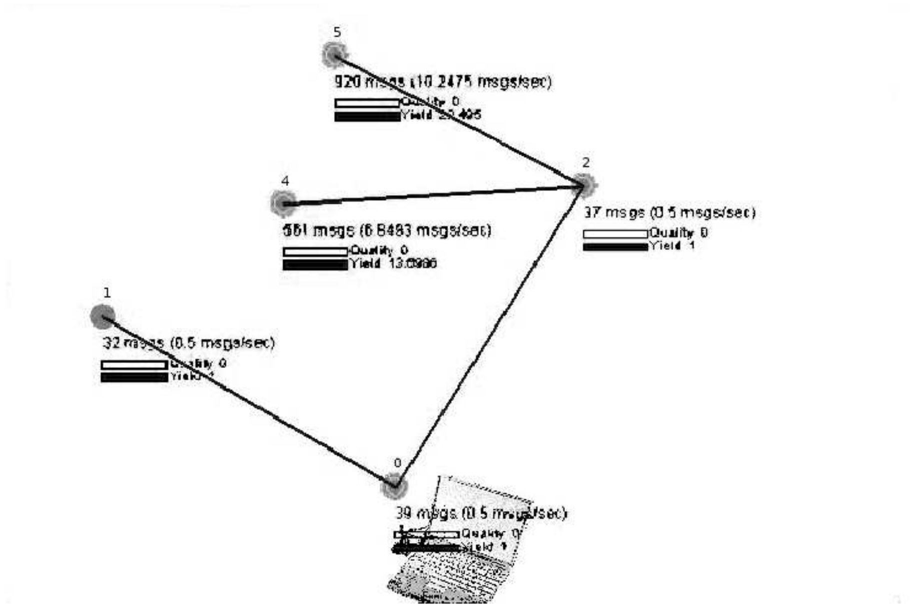

Figure 7-1
The Surge demo application running on a mixed network of Mica2 Motes (nodes 0, 1 and 2) and BTnodes (nodes 4 and 5).

component using it had to be removed from the `Surge` application. In partuicular the `Sounder` component had to be removed completely removed. The `Photo` component, which acesses a photo sensor on the sensorboard was replaced by the RSSI ADC, which measures the signal strength of the ChipCon1000 module. Therefor the `ADCC` component was used and the `SurgeM.ADC` was connected to `ADCC.ADC[TOS_ADC_CC_RSSI_PORT]` instead of `Photo`.

8 Conclusion

Porting TinyOS on to the BTnode platform was conducted in several steps. During this work a few of the standard applications could be successfully run. Although this port is still incomplete because only the subsystems controlling the LEDs and the ChipCon1000 were ported, alread quite a few applications can be run just by compiling them for the `btnode3_2` platform.

Because sensorboards are not yet available for the BTnodes it is not possible to use applications that require one of them. For those either a special sensorboard for the BTnodes or an adapter to an existing one has to be build. Same goes for applications that use the on board EEProm of the ”Mica2 Motes” which was replaced by an SRAM module. Although it should be straight forward to write an interfacing component to use the SRAM as storage this was not done yet. Due the volatile nature of SRAM applications requiring a permanent storage of data have to deal with additional difficulties. These can be either solved by providing a fall back power supply to the BTnodes or attaching a non-volatile memory as storage. The additional 60 kB of memory of the BTnodes could allow to test applications that need a data such as dynamic routing algorithms and the like. The dual radio nature of the BTnodes could also lead to interesting applications where communication between BlueTooth enabled consumer electronics is required or can simplifiy the data aquisition as PCs and laptops with BlueTooth become more and more common.

Bibliography

[1] *The BTnode website*
`http://www.btnode.ethz.ch/`

[2] *The "Mica2 Mote" hardware description*
http://www.tinyos.net/scoop/special/hardware/#mica2

[3] *The Robomote website*
`http://www-robotics.usc.edu/~robomote/`

[4] *OSKit Project website*
`http://www.cs.utah.edu/flux/oskit/`

[5] *The TinyOS website*
`http://www.tinyos.net/`

[6] *TinyOS Installation*
`http://www.tinyos.net/tinyos-1.x/doc/install.html`

[7] *TinyOS Tutorial*
`http://www.tinyos.net/tinyos-1.x/doc/tutorial`

[8] *TinyOS on BTnotes*
`http://www.distlab.dk/madsdyd/work/tinyos`

[9] *The uClinux website*
`http://www.uclinux.org/`

[10] *The QNX website*
`http://www.qnx.com/`

[11] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, Eric Eide *Knit: Component Composition for Systems Software*, Proceedings of the Fourth Symposium on Operating Systems Design and Implementation 2000

[12] J. Beutel, O. Kasten, F. Mattern, K. Rmer, F. Siegemund and L. Thiele, *Prototyping Wireless Sensor Networks with BTnodes*, 1st European Workshop on Wireless Sensor Networks 2004

[13] M. Flatt, M. Felleisen, *Units: Cool Units for HOT Languages*, In Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation

[14] David Gay, Phil Levis, Eric Brewer, and David Culler. *nesC 1.1 Language Reference Manual*, 2003

[15] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler, *The nesC Language: A Holistic Approach to Networked Embedded Systems*, Proceedings of Programming Language Design and Implementation (PLDI) 2003,

[16] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister, *System architecture directions for network sensors*, ASPLOS 2000

[17] Konrad Lorincz, Matt Welsh, *A Robust, Decentralized Approach to RF-Based Location Tracking*, Technical Report TR-19-04, Harvard University, 2004