Chapter 11

BTnodes and Sensors

While the BTnode has been designed for conducting research in *Wireless Sensor Networks* (WSNs), it does not carry any onboard sensors. This is in contrast to other WSN-platforms, such as the Tmote Sky, which (optionally) comes with three onboard sensors (temperature, light, and humidity). The reason for not including a fixed set of sensors lies in its added flexibility: depending on the particular application, BTnodes can be equipped with seperate "sensor-boards" that contain only the required set of sensors and which can be directly connected to one of the external connector sockets on the BTnode.

Working with sensors on the BTnode thus requires us to choose either a pre-made sensor board, or to connect our own set of sensors directly to one of the BTnode's connectors. In this tutorial, we will use the BTsense sensor board, developed as part of the 2006 Wireless Sensor Network lecture at the ETH Zurich's Inst. of Pervasive Computing. It has been specifically designed to contain both analog (light) and digital (temperature and motion) sensors, as well as an actuator (buzzer). It is connected through the BTnode's "Debug Connector" (called J2) on the side, and is designed to be attached (e.g., with some plaster material) to the side of the BTnode. Figure 11.1 shows the top of the board. In particular, the BTsense board features the following sensor and actuators (rev 1.1):

1. Microchip TC74 digital (I^2C) temperature sensor [9]
2. Taos TSL252R analog light sensor [13]
3. Napion AMN1 digital (binary) motion sensor (passive IR) [15]
4. muRata 7BB-12-9 piezoelectric buzzer [10]

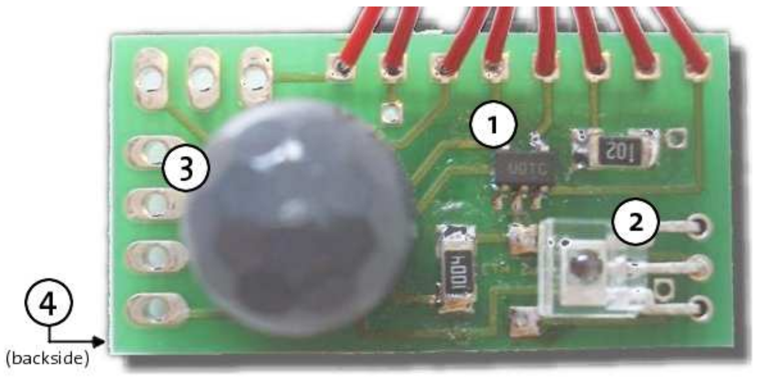

Figure 11.1: The BTsense rev1.1 sensor board.

Another popular alternative for connecting sensors to the BTnode is the set of boards developed at Teco in Karlsruhe. These boards have an extension connector that fits directly into the USB programming board. They come in several sizes and differ in the number of sensors they offer. The largest board, *spart*, additionally featured a separate microcontroller that would relieve the BTnode ATmega of any sensor related management tasks, though making its usage somewhat more difficult. The *ssmall* boards (without microcontroller) are available in a “medium” and “large” size and feature:

- Microchip TC74 digital (I^2C) temperature sensor
- Taos TSL2500 analog light sensor
- MAX8261 OP capacitive microphone
- ADXL210 2-axis acceleration sensor
- Second ADXL210 for combined 3-axis acceleration sensor (only on “full” version of the board)
- solder plates for optional pressure sensor, humidity sensor, second temp. sensor, speaker, etc.
- Two LEDs

In order to be able to gather sensor data on our BTnode, we first need to understand how its processor, the ATmega128L, receives and processes external data, and then how we can use BTnut to use this information in our program.

11.1 ATmega128L I/O-Ports and Registers

The ATmega128L microprocessor features 53 programmable I/O lines. It is through these lines that all communication to and from the processor takes place. While all 53 lines can be used in a totally generic fashion (i.e., they can both be used to output a bit, as well as reading input bits), all of them also have at least one so-called “alternate function”, i.e., they are connected to a specific on-chip feature such as the analog-digital converter, the UART, a hardware timer, or an external interrupt signal. It is up to the programmer – either from within the OS, or as part of the application – to properly choose how a particular I/O line should be used: as a generic output line, as part of an ADC conversion, to monitor an input line and throw an interrupt whenever it changes, to control a set of digital sensors via a sensor-bus such as the I^2C-interface, etc.

Figure 11.2 shows all 64 pins of the ATmega128L. PA though PG are the seven available I/O ports, with ports A through F having 8 pins each, while port G has only five pins. Each port is represented through three registers each, which together provide – for each pin of each port – access to its I/O functionality: the *Data Direcion Register* (`DDRx`) (where x stands for A through G) defines whether a particular pin on a port will be used for input or output, while the *Data Register* (`PORTx`) and *Port Input Pin* (`PINx`) register (among other features) provide access to output and input values of each pin, respectively. Page 84 of the ATmega128L manual [1] gives an overview of all I/O port registers.

Additionally, more than one hundered registers can be used to enable or disable a certain “alternate function” of each pin. For example, the `ADCSRA` register controls the analog-digital converting unit – like turning it on and off, and starting a conversion – while the `ADMUX` register controls which of the potential input pins (pins 0 through 7 of port F) are to be used during the conversion.

Each of these registers (see the ATmega128L manual [1] on page 364 for a complete list) is provided to the programmer as a so-called *hardware register*. While programmers typically understand the term “register” to be a *processor register* – a small amount of very fast on-chip memory that is used to hold intermediate values during a computation in a very efficient manner – hardware registers are much more common in embedded systems programming. They often look and feel like being just another memory value, yet they physically control access to various devices. The avr-libc defines mnemonics[1] identical to the ones used in the ATmega128L manual as shorthands, in order to allow statements of the form “enable `ADEN` in the `ADCSRA` register” instead of “set bit 7 of register `0x0026` to 1”.

[1]Actually: these are precompiler definitions, which can be found in `avr/io.h`.

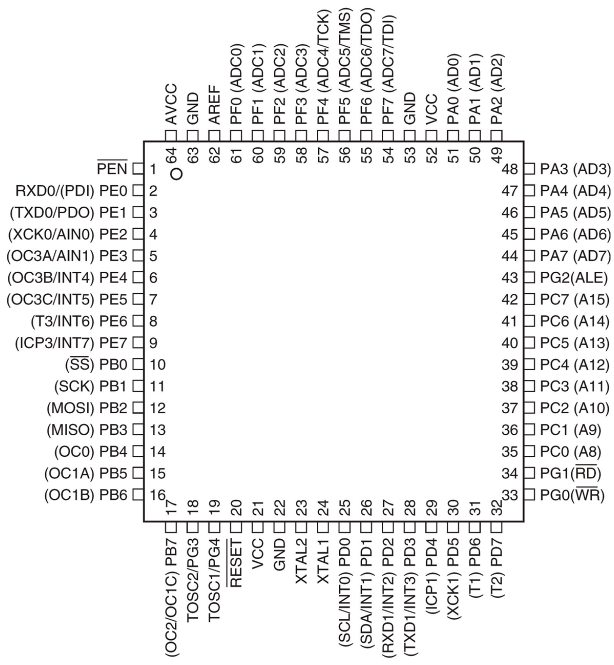

Figure 11.2: Ports of the ATmega128L [1].

Exercise 11.1. *Use the ATmega128L user manual [1] to find out which pins have the alternate function of serving as the data (SDA) and clock (SCL) line of the so-called "two-wire interface" (TWI). Find out what bit needs to be set in which hardware register in order to enable TWI support in the ATmega128L.*

Exercise 11.2. *Locate the schematic for the BTsense sensor board at www.btnode.ethz.ch (search under Hardware Reference) and find out to what ports and pins each of the three sensors – the light, temperature, and motion sensor – are connected.*

When using any kind of sensor platform with a BTnode, we thus first need to know how an individual sensor is connected to our microprocessor. Analog sensors will most certainly be connected (or have to be connected) to one of the ADC input pins, while digital ones either use TWI or are connected to a generic input pin. Knowing how and where a sensor is connected, we then need to understand the sensor's output, i.e. what information is delivered from the sensor to the input pin. This can be found in each sensor's *datasheet*. Last not least, we must then properly query these sensor values in our program: either by simply polling a sensor value repeatedly; using a timer to do this repeated polling for us; or by setting an interrupt to occur whenever a sensor value changes.

The following sections will describe each sensor type in turn, before outlining two possible ways of querying sensor values: polling and interrupts.

11.2 Sensor Types

BTnodes support three types of sensors: digital bus (I^2C) sensors, digital logic-level sensors, and analog sensors. Each type needs to be connected to different pins, each type needs a different way to read out a value. The sections below describe in detail how a generic sensor would need to be read out. However, for certain sensor platforms such as the BTsense sensor board, higher-level support is available in form of dedicated functions, alleviating the need for direct manipulations of the individual ATmega128L ports. Nevertheless, knowing the general principle of sensor read out should certainly foster overall understanding.

11.2.1 Digital I^2C-Bus Sensors

The I^2C-bus[2] was originally developed by Philips Semiconductors in the 1980s to simplify communication among various chips within TV-sets. It is a simple Master-Slave-bus, with a 7-bit address space that supports up to 112 slave devices (16 addresses of the possible 128 are reserved).[3] The biggest advantage of the I^2C-protocol is its abilitiy to allow a single microcontroler the control of more than hundred devices with only two I/O-pins. The ATmega128L used in the BTnode supports the I^2C-protocol in hardware, which greatly simplifies control of I^2C-compatible devices. However, as I^2C is a registered trademark of Philips, Atmel calls this TWI ("two wire interface").

The two wires of the I^2C-bus are called SDA (data) and SCL (clock). Communication is always initiated by the master and is only between the master and a single slave. The clock is controlled by the master (this is handled by the ATmega128L for us) – it tells the slave when it should read a value from SDA (i.e., when SCL is high).[4] This allows the use of the I^2C-protocol also without fixed hardware or real-time clocks.

In order to poll an I^2C-sensor in BTnut, we need to know its address on the I^2C-bus. Addresses are defined in the corresponding header file (hopefully conflict-free) – in `btsense/btsense.h` for the BTsense board, and under `extras/teco_ssmall` for the *ssmall* board (both of which use the I^2C-compatible TC74 temperature sensor [9]).

Also, we need to know the corresponding I^2C-command that needs to be issued through the `TwMasterTransact`-function. This information can be found in the sensor's datasheet – for the TC74, the datasheet lists `0x00` as the command code for reading a temperature value.

Explanation *TWI-Communication in BTnut*:
NutOS comes with a simple two wire interface (TWI) library that works also on our ATmega128L. The most important commands are `TwInit` to initialize the interface, and `TwMasterTransact` to send commands to, and receive data from, the individual sensors.
`TwInit` takes a sole argument a 7-bit slave address, in order to allow (in theory) our master to also act as a slave to other masters on the bus. However, as the current implementation does not support slave mode for the ATmega128L, the parameter can safely be ignored (set it to 0, for example).
`TwMasterTransact` takes as a first argument the (slave) device address, followed by two variables each for sending and receiving data: (the address of) the variable where the command can be found, followed by its length, and the (address of the) variable where the received data should be put, followed by the maximum number of bytes to receieve. A final argument indicates a timeout value, which is currently not supported (will be ignored). It returns the number of bytes received, or -1 in case of error.

```
#include <dev/twif.h>
#define BTSENSE_I2C_TC74 0x48

void main(void) {
    . . .
    // set TWI pins (Port D Pins 0 and 1) as Input w/ Pull-Up
    cbi(DDRD, DDD0); cbi(DDRD, DDD1); sbi(PORTD, PD0); sbi(PORTD, PD1);
    TwInit(0); // parameter currently ignored
    . . .
    u_char tw_cmd = 0x00; // "read temperature"-command
    u_char t;             // holds return value (temperature)

    if (TwMasterTransact(BTSENSE_I2C_TC74, &tw_cmd, 1, &t, 1, 0) == -1) {
       printf ("Error while reading sensor: %i\n", TwMasterError()); }
    . . .
}
```

[2]I^2C is pronounced "i-square-c", sometimes also "i-two-c".

[3]The address space can optionally be extended to 10 bit and 1008 devices (1024-16), though this is not supported on the BTnode.

[4]During a high SCL level, SDA levels must be stable. Level changes on SDA during a high SCL indicate special START and STOP commands that a master uses to initiate or end a command.

The `TwInit` function simply initializes the software stack – it does not configure the corresponding ATmega128L ports for us. We therefor need to make sure that both of our TWI ports (i.e., pins 0 and 1 of Port D) are both configured for input (using the `DDRD` register) and have *pull-ups* enabled (using the `PORTD`).[5]

Explanation *Pull-ups*:
Pull-ups are resistors in an electronic circuit that ensure that, given no other input, a circuit assumes a default value. The I^2C-protocol requires that when IDLE (i.e., when no devices use it), the bus remains in a logic HIGH state. This is achieved by inserting so-called *pull-up resistors* into the circuitry, which have the effect that as soon as at least one device puts a LOW value onto the bus, the whole circuit will be pulled to a logic LOW state. This allows other devices to detect communication on the bus.
Each of the 53 I/O-pins of the ATmega128L can have pull-ups enabled or disabled, using the `PORTx` register.

Also notice that the `TwMasterTransact` function references both the command variable *and* the result variable, i.e., it is not just for sending a command to a TWI-compliant device, but also for receiving its result.

Exercise 11.3. *Locate the TC74 datasheet off the BTsense documentation page on the BTnode Web site and find all supported I^2C-commands (with their corresponding command codes).*

11.2.2 Digital Logic-Level Sensors

Another type of digital sensor is that of the *logic-level* sensor. While also digital, it simply responds in a binary fashion: logical 1 and 0 (`VCC` and `GND`) represent "on" or "off", "detected" or "not detected", "critical" or "not critical". Such sensors do not need (and do not support) special communication protocols such as I^2C. Instead, we can directly connect them to one of the available I/O pins of the ATmega128L, configure the corresponding port-pin as "input" (using the `DDRx` register, cf. section 11.1) and read its value from the `PINx` register.

Explanation *Reading logic-level data in BTnut*:
Knowing to which pin a particular binary input is connected, we can easily define this pin as an input pin and read out its value. BTnut offers the *setbit* and *clearbit* functions – `sbi` and `cbi` – that set and clear individual bits of a selected register, respectively.
BTnut contains macros for all ATmega128L ports and pins, allowing for a convenient way of setting or clearing individual bits in a register. These macros are identical to the identifiers given in the ATmega128L reference manual [1] – see page 364 for an overview of all registers and pins.

```
// define pin 5 of port B as an input pin in port B's DDR register
cbi (DDRB, DDB5); // '0' means input pin
. . .
// read out all 8 pins of port B
u_char current_value_port_b = PORTB;
if (PORTB & (1<<PB5)) {
    // pin 5 is set
}
```

In many situations, it is important not simply to know a logic-level sensor's current value, but instead to know *when* it changes. The ATmega128L offers various *interrupts* that can be configured to observe an input pin for change, and trigger a program interruption whenever the output of such a logic-level sensor changes. More about such interrupts can be found in section 11.3.2 below.

Exercise 11.4. *Write a program that continously reads out (and prints) the value from the BTsense (logic-level) motion sensor. If you do not know to which port it is connected, see ex. 11.2 above.*

[5] See page 65 of the ATmega128L user manual [1] for an overview of I/O-Port configuration.

11.2.3 Analog Sensors

Analog sensors do not simply deliver an “off”/“on” value, but output a different voltage level for each possible sensor reading. In order to use this information in a program, this voltage level needs to be *sampled* into a binary value, typically between 0-255 (i.e., 8 bit resolution), though up to 10 bits resolution are supported on the ATmega128L.

Digitizing analog data on the ATmega128L is generally simple: its built-in Analog-Digital Converter (ADC) supports up to 8 different analog input channels (two of which optionally amplify the signal 10 or even 200 times), noise cancellation, and either single or continuous conversion modes. One only needs to properly setup the various needed parameters, trigger a conversion, and subsequently read out the resulting digital values. Each of these steps can be controlled through one or more ATmega128L registers.

Explanation *Using the ATmega128L ADC*:
ADC setup is performed through the *ADC Control and Status Register* (`ADCSRA`), where for example the ADC can be enabled and disabled (bit 7, `ADEN`), and single conversions can be triggered (bit 6, `ADSC`). The *ADC Multiplexer Selection Register* (`ADMUX`) allows the selection of input pins, as well as voltage reference and input gain setup. Note that before configuring the ADC, it should be turned off (i.e., the `ADEN` bit in `ADCSRA` should be cleared).
After starting a single conversion, the result is written to two registers, `ADCL` and `ADCH`. In order to know when the conversion is finished and these values can be read, one can simply check the value of the `ADSC` bit in the `ADCSRA` register: as soon as it is cleared, the result of the single conversion can be read. Alternatively, one can setup an ISR for the ADC interrupt (`sig_ADC`, see table 11.2 below) or wait for its corresponding flag (`ADIF` in the `ADCSRA` register) to be set. The default setup will put the LSB into `ADCL` and bits 8 and 9 into `ADCH`.

```
cbi(ADCSRA, ADEN); // disable ADC
ADCSRA = 0; // stop ADC & conv., no free-runn, no irq, def. prescaler
ADMUX  = 0; // AREF, ADLAR cleared, ADC0 input
sbi(ADCSRA, ADEN); // enable ADC
sbi(ADCSRA, ADSC); // start single conversion
// wait until conversion is finished
while (bit_is_set(ADCSRA, ADSC));
// find result in ADCL and ADCH
NutEnterCritical();
result = ADCL | (ADCH << 8);
NutExitCritical();
```

Note that its is important that reading out the final value is not temporarily suspended by a system interrupt (see more on interrupts in section 11.3.2 below), otherwise we might get a skewed result.

Exercise 11.5. *Use the above skeleton-code to write a program reading out the BTsense board’s light sensor (if you do not know to which port it is connected, see ex. 11.2 above).*

In BTnut, the above ADC functions are encapsulated in the `dev/adc.h` library. All of the above mentioned functions – disabling, enabling, and configuring the ADC, as well as reading out converted values – can be achieved with a set of dedicated functions and corresponding constants, thus increasing code legibility and portability. However, the `adc.h` library is unable to cope with concurrent use, making it practically unusable. This is because other threads might concurrently use the ADC for other purposes (e.g., to measure the current battery level), thus reconfiguring the ADC repeatedly. In order to get reliable measurements, it is imperative to assert that the current ADC configuration still matches the desired one, or change it if otherwise. The `dev/adc2.h`-library offers the `adc2_init` and `adc2_read` functions, which are much more robust than their `dev/adc.h` counterparts and also support configuration validation.

Explanation *Using ADC in BTnut*:
The functions `adc2_init` and `adc2_read` are defined in the library `dev/adc2.h`. In contrast to the regular `dev/adc.h`-library, these functions also support ADC context switches, i.e., the concurrent use of the ADC by other threads.

```
#include <dev/adc2.h>

static u_short my_adc_handle;  // saves ADC context
int main (void) {
    my_adc_handle = adc2_init(ADC2_MODE_SINGLE_CONVERSION, ADC2_PRESCALE_DIV2,
                              ADC2_CHANNEL0, ADC2_REF_AREF);

    for (;;) {
        val = adc2_read(my_adc_handle); // read from prev. saved context
        printf("%d\n",val);
        NutSleep(1000);
    }
}
```

Optional Exercise 11.6. *Reimplement ex. 11.5 using the* `adc2.h` *library referenced above. How does the library support ADC-context switches? Locate the source code of* `adc2.c` *and look up.*

11.3 Reading Sensor Data

Depending on the type of sensor that we want to read out, different reading strategies might be appropriate.

11.3.1 Polling

Polling is the simplest yet least efficient way of reading sensor values. The simplest way would be to wrap the reading in a loop, potentially in a separate thread in order to allow the main program to continue executing other tasks. However, this approach ties up a lot of processing power and uses up precious energy when running under battery power. In most cases, one would want to at least include a `NutSleep` statement within the loop, to ensure that sensor readings only happen seconds or minutes apart (not milliseconds), e.g., for recording light or temperature values across several hours.

Instead of looping and repeatedly calling `NutSleep`, we can also let BTnut do the work for us, by using the `NutTimerStart` function described in section 4.4 above. By utilizing BTnut timers, repeated requests for sensor data can be scheduled over the course of hours, days, or even weeks. Given the 32-bit resolution of the `NutTimerStart` function, both one-shot and periodic timers of up to 49 days can be installed.

11.3.2 Interrupts

BTnut timers are a less resource intensive way of "manually" polling (e.g., in a loop) a sensor value. They are well suited for periodic measurement tasks, e.g., for documenting the temperature every 10 minutes over the course of a day. Sometimes, however, it is necessary to quickly react to a change in the measured data. Instead of increasing the polling frequency (and thus tying up CPU cycles), we can use an *interrupt* to get automatically notified of changing values.

The ATmega128L offers eight *external* interrupt request lines (i.e., pins that can automatically trigger the execution of a particular code snippet) and several *internal* interrupts (i.e., for monitoring internal processes, such as the above-mentioned counter overflows). Table 11.2 lists selected signals in BTnut – for a complete list of interrupts, see section *Interrupt Vectors* in the ATmega128L manual [1]. When interrupts are enabled, the processor will automatically interrupt the normal program flow and execute a previously registered *interrupt service routine* (ISR). As ATmega128L interrupts always have a higher priority than regular program code,

Mode	Description
NUT_IRQMODE_LOWLEVEL	Signal as long as level is low
NUT_IRQMODE_FALLINGEDGE	Signal when level changes to low
NUT_IRQMODE_RISINGEDGE	Signal when level changes to high
NUT_IRQMODE_EDGE	Signal whenever level changes

Table 11.1: BTnut external interrupt modes

they will be executed almost immediately when their interrupt condition holds true, allowing for almost real-time handling of events.

Explanation *Using interrupts in BTnut*:
In order to activate a particular interrupt in BTnut, we simply need to register an interrupt handler, a so-called *interrupt service routine*, with the corresponding *interrupt signal*. The `NutRegisterIrqHandler` function takes a signal, an ISR, and an optional argument to be passed to the ISR. Before assigning a new ISR, the interrupt in question should be turned off (using `NutIrqDisable`); *afterwards* it should of course be turned on (usign `NutIrqEnable`).
For external interrupts, which allow monitoring the logical level of up to eight input pins (`INT0` through `INT7`), we need to additionally clear its `DDRx` port bit (to define the pin as an input pin), and define for what kind of levels or level changes we want an interrupt. This is done with the *NutIrqSetMode* function, which takes an external interrupt signal and a trigger mode as input (again, this should be set *before* the input is enabled). Table 11.1 summarizes the various ways the pin level can be monitored.

```
#include <dev/irqreg.h>
. . .
void my_interrupt6_handler (void* arg) {
   static u_char my_variable = 0; // static variables for persistance
   . . .
}
. . .
NutIrqDisable( &sig_INTERRUPT6 );
cbi(DDRE, DDE6); // define pin E6 as input
NutRegisterIrqHandler(&sig_INTERRUPT6, my_interrupt6_handler, NULL);
NutIrqSetMode( &sig_INTERRUPT6, NUT_IRQMODE_EDGE );
NutIrqEnable( &sig_INTERRUPT6 );
```

What happens if an interrupt occurs during such an ISR? BTnut does not support stacked interrupts, so the current ISR will first be finished. When the system exists an ISR and finds another interrupt waiting (i.e., its corresponding interrupt bit is set) it will continue with executing the ISR of the next interrupt. However, while the current ISR was still running, there might have actually been multiple identical interrupts – e.g. a certain value crossed a threshold not only once (and threw an interrupt), but twice, or more often. As there is only one flag to indicate whether an interrupt has fired, there is no way to know how many interrupts have been missed during the execution of the current ISR. It is therefor important to keep the code inside an ISR as short as possible, in order to minimize the chances of missing out on important other interrupts, e.g., incoming packets on the Chipcon or Bluetooth radio. Another factor that should not be underestimated is the time it takes the system to switch between the main program and an ISR – typically tens or hundreds of CPU cycles, in order to save the current system state and switch to an ISR (and again back to the main program).

This uninterruptability of ISRs is sometimes also needed within the main program. For example, certain 16-bit registers of the ATmega128L need to be written to in an *atomic* fashion, e.g., either both bytes get written or none. If an interrupt occurs in the middle of such an assignment, the already written first byte might not be the same anymore by the time program control returns to the main program. BTnut offers the `NutEnterCritical` and `NutExitCritical` functions (in `sys/atom.h`) to allow main program code to run uninterrupted. As with ISR code, these parts of code should be as short as possible, in order not to loose any interrupt signals. Note that `NutExitCritical` does not simply re-enable interrupts. Instead,

Signal	**Description**
sig_ADC	ADC conversion complete
sig_COMPARATOR	Analog comparator
sig_INTERRUPT0	External interrupt 0
sig_INTERRUPT1	External interrupt 1
sig_INTERRUPT2	External interrupt 2
sig_INTERRUPT3	External interrupt 3
sig_INTERRUPT4	External interrupt 4
sig_INTERRUPT5	External interrupt 5
sig_INTERRUPT6	External interrupt 6
sig_INTERRUPT7	External interrupt 7
sig_SPI	SPI interrupt entry
sig_INPUT_CAPTURE1	Timer 1 input capture
sig_INPUT_CAPTURE3	Timer 3 input capture
sig_OUTPUT_COMPARE0	Timer 0 output compare
sig_OUTPUT_COMPARE1A	Timer 1A output compare
sig_OUTPUT_COMPARE1B	Timer 1B output compare
sig_OUTPUT_COMPARE1C	Timer 1C output compare
sig_OUTPUT_COMPARE2	Timer 2 output compare
sig_OUTPUT_COMPARE3A	Timer 3A output compare
sig_OUTPUT_COMPARE3B	Timer 3B output compare
sig_OUTPUT_COMPARE3C	Timer 3C output compare
sig_OVERFLOW0	Timer 0 overflow
sig_OVERFLOW1	Timer 1 overflow
sig_OVERFLOW2	Timer 2 overflow
sig_OVERFLOW3	Timer 3 overflow
sig_UART0_RECV	UART0 receive complete
sig_UART1_RECV	UART1 receive complete
sig_UART0_TRANS	UART0 transmit complete
sig_UART1_TRANS	UART1 transmit complete
sig_UART0_DATA	UART0 data register empty
sig_UART1_DATA	UART1 data register empty

Table 11.2: Selected `BTnut` interrupt signals

`NutEnterCritical` saves the current interrupt state before disabling them, so that `NutExitCritical` can restore whatever state previously existed. If interrupts were disabled before calling `NutEnterCritical`, they still stay disabled even after calling `NutExitCritical`.

11.4 Hardware Timers and Actuators

While adequate for issuing periodic sensor readings, the use of `NutTimerStart` has two important drawbacks: it only has a resolution of milliseconds, and actual code execution is thread-based, i.e., it might be delayed (potentially indefinitely) due to higher priority threads or non-yielding threads. In order to use more real-time and fine-grained timers, the integrated hardware timers of the ATmega128L can be used directly. This becomes important when driving actuators, e.g., the buzzer of the BTsense sensor board, or controlling motors based on pulse-width modulation (PWM).

The ATmega128L processor features two 8-bit and two 16-bit timers (Timer0 and Timer2, and Timer1 and Timer3, respectively). These simply work as *counters*, i.e., they continuously count from 0 to 255 (or 65535) and begin again from 0 afterwards. Whenever the counter overflows (i.e., starts again at 0), an *overflow interrupt* can be triggered, which allows a program to periodically execute a certain command. While

Timer0 is already in use in BTnut to drive its timer functions (e.g., `NutTimerStart`, but also `NutSleep`), the remaining timers are available for use in your BTnut program.

Using a number of processor registers, one can customize the behavior of these counters. For example, by writing to the `OCRx` register (x being 0, 1, 2, or 3), we can set the so-called TOP value, i.e., the value at which an interrupt should be triggered. One can also switch a timer to the so-called *Clear Timer on Compare* (CTC) mode, where it restarts counting at zero whenever the counter reaches the TOP value (otherwise it continues to the 8-bit or 16-bit maximum).[6] Counters run at most with the speed of the main CPU – which runs at about 7.37MHz in the case of the ATmega128L on the BTnode.[7] Using a so-called *prescaler*, the counter can be slowed by factors of 8, 64, 256, or 1024. This can be set in the *Timer/Counter Control Register* `TCCRxn` (with x being 0, 1, 2, or 3, and n being A, B, or C for the two 16-bit counters only).

Explanation *Using a hardware timer in BTnut*:
The ATmega128L hardware timer/counter must be accessed directly through the corresponding hardware registers. The four timer are started by setting their corresponding prescaler value to non-zero value (see above). Also, one needs to set the count at which an interrupt and/or a reset to 0 should be triggered, as well as indicate what counter mode should be used (normal, CTC, etc.). The example below uses the 8-bit timer *Timer2*.

```
#include <sys/atom.h> // for NutEnterCritical and NutExitCritical
. . .
u_char max = 128;

void my_timer_handler(void* arg) {
   static u_char active = 0;

   if ( active == 0) { . . . } else { . . . }
}

void setup_counter0(void* arg) {
   // registers my_timer_handler to be called for 8-bit counter0 timer

   // set counter mode to CTC (see ATmega128L manual p.156) in
   // Timer/Counter2 Control Register TCCR2
   sbi (TCCR2, WGM21); cbi (TCCR2, WGM20);

   // set prescaler to 1024 (slowest timer possible), see p.157
   sbi (TCCR2, CS22); cbi (TCCR2, CS21); sbi (TCCR2, CS20);

   // make sure interrupts are turned off
   NutEnterCritical();
   // register interrupt handler to be called for 8-bit counter0
   NutRegisterIrqHandler(&sig_OUTPUT_COMPARE2, my_timer_handler, NULL);
   // set TOP value (i.e., when interrupt should be triggered)
   OCR2 = max;
   // reset current counter value to zero
   TCNT2 = 0x00;  // no need to start anything - counter runs continuosly
   // enable interrupts again
   NutExitCritical();
}
. . .
```

Exercise 11.7. *What type of counter do you need to generate waveforms for the 7BB-12-9 buzzer of the BTsense sensor board? You will have to take into account not only the desired signal frequency, but also the speed of the processor and the possible values of the prescaler. Hint: The corresponding chapters in the ATmega128L manual [1] contain a formula for computing a timer/counter's frequency.*

[6]Further counter modes can be found in the corresponding chapter of the ATmega128L manual [1].

[7]The exact processor speed can be obtained by calling `u_long NutGetCpuClock(void)`.

When setting the `OCx` pins as output pins (using the corresponding port's `DDRx` register), one can easily connect a waveform output to a peripheral device, such as a buzzer or a motor.[8] The `OCx` pins can be used in three different modes: *CTC*, *Fast PWM*, and *Phase Correct PWM*. In the already mentioned CTC mode, the `OCx` pin can be set to simply alternate (toggle) between 0 and 1 whenever the TOP values is reached (see figure 11.3 below). In Fast PWM mode[9], the counter always counts from BOTTOM to MAX. The `OCx` pin is cleared whenever the TOP value is reached, and set when the counter begins again at BOTTOM. This ensures PWM-signals with constant periods (i.e., from BOTTOM to TOP) that have a pulse width of exactly TOP (see figure 11.4 below). Phase-correct PWM finally creates the high pulse of the PWM signal always in the center of the period, not at its beginning flank, by counting from BOTTOM to TOP and back again, and inverting the signal when reaching TOP (both upwards and downwards, see figure 11.5 below).

When using a timer to drive an actuator connected to one of these pins, this has the advantage of not needing a separate interrupt service routine to explicitly set a pin output to 1 or 0: the timer/counter's corresponding `OCx`/`OCxn` will automatically alternate between 0 and 1 whenever the counter reaches its TOP and/or BOTTOM value.

Explanation *Putting a Waveform onto an I/O pin in BTnut*:
By connecting a device to the output pin of a 8-bit or 16-bit counter, we can directly modulate a corresponding signal onto the pin. This only requires that we set the data direction register of this pin (i.e., to define it as an "output" pin):

```
u_short max = 57535; // example

// set counter mode to CTC (see ATmega128L manual p. 133) in
// Timer/Counter1 Control Registers TCCR1A/TCCR1B
cbi (TCCR1B, WGM13); sbi (TCCR1B, WGM12);
cbi (TCCR1A, WGM11); cbi (TCCR1A, WGM10);

// set up output pin OCR1A to be toggled by counter (p.131)
cbi (TCCR1A, COM1A1); sbi (TCCR1A, COM1A0);

// start counter with prescaler to 8 (example), see manual p. 135
cbi (TCCR1B, CS12); sbi (TCCR1B, CS11); cbi (TCCR1B, CS10);

// make sure interrupts are turned off
NutEnterCritical();
// no need for interrupt handler! Simply set TOP value. (Note: this is a 16-bit
// register, see 'accessing 16-bit registers' in the ATmega128L manual on p.112)
OCR1A = max;
// reset current counter value to zero
TCNT1 = 0x0000;
// enable interrupts again
NutExitCritical();

// enable pin toggle on OC1A (i.e., PB5) by setting bit 5 of DDRB to 1
sbi (DDRB, DDB5);  // needs something connected to PB5 to show some effect!
```

Note again that each counter can only toggle a selected number of pins (the `OCxn` pins), i.e., Timer/Counter 0 can only toggle PB4 (also called OC0), Timer/Counter1 only PB5, PB6, and PB7 (also called OC1A, OC1B, and OC1C), Timer/Counter2 to PB7 (called OC2), and Timer/Counter3 pins PE3, PE4, and PE5 (called OC3A, OC3B, and OC3C). See the section on I/O Ports in the ATmega128L manual (p. 63ff) and figure 11.2 in this tutorial.

Exercise 11.8. *Describe the steps necessary to put a 440Hz signal onto an* `OCxn` *pin.*

[8]The two 8-bit counters 0 and 2 have only one such pin – `OC0` and `OC2`, respectively – while the 16-bit counters 1 and 3 feature three such pins: `OC1A`, `OC1B`, `OC1C` and `OC3A`, `OC3B`, `OC3C`.

[9]PWM stands for *Pulse Width Modulation*.

Optional Exercise 11.9. *Describe the steps necessary to put the same 440Hz signal onto an* arbitrary *I/O pin of the ATmega128L. What is the difference to the solution in ex. 11.8?*

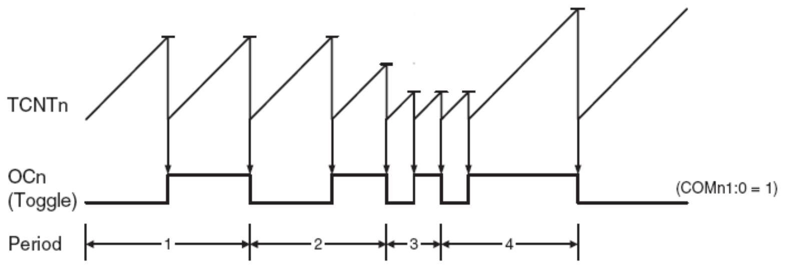

Figure 11.3: Waveform generation in CTC mode [1]. Notice the variable signal periods due to varying TOP values (horizontal bars).

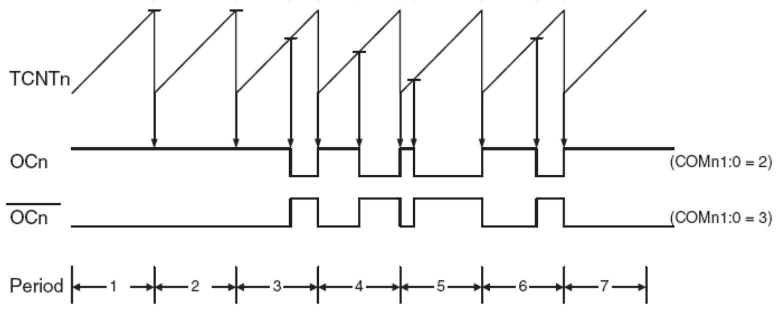

Figure 11.4: Waveform generation in Fast-PWM mode [1]. Periods are constant, signal width is according to TOP value (horizontal bars).

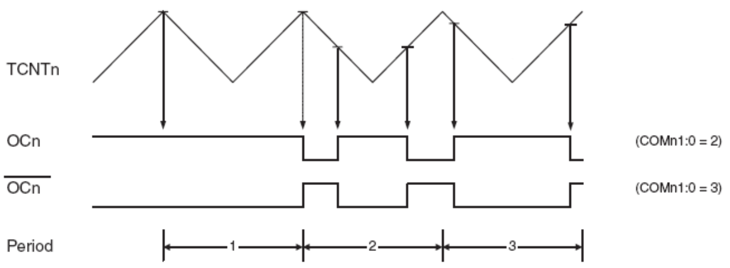

Figure 11.5: Waveform generation in Phase-Correct-PWM mode [1]. Notice how signals are centered within constant periods.

11.5 The `btsense`-library

Many of the low-level details for querying sensor data off the BTsense sensor board have already been encapsulated in dedicated functions as part of the `btsense`-library. While it might still be necessary to use BTnut timers, interrupts, or hardware timers to read out sensor values periodically and/or automatically, these functions should greatly simplify the act of reading out each of the three sensors, as well as driving the connected buzzer.

Explanation *Using the library for BTsense*:
The btsense-library offers three functions for reading out each of the three sensors, as well as a function for driving the buzzer with a particular frequency. It takes care to properly set all required hardware registers, as well as configure both the ADC and any necessary timer/counters. Note that the `btsense_init` function requires a board revision identifier – typically this should be `BTSENSE_REVSION_1_1A`.

```
#include <btsense/btsense.h>

int main(void* arg) {
   btnode_init(); // init hardware, uart, network
   btsense_init( BTSENSE_REVISION_1_1A );

   u_short light = btsense_sample_light();
   printf ("Light Level: %d\n", light);
   u_char motion = btsense_sample_motion();
   printf ("Motion Level: %d\n", motion);
   u_char temp; int err;
   int err = btsense_sample_temp(&temp);
   if (err != 0) {
      // TWI error
      printf ("TWI Error: %d\n", err);
   } else {
      printf ("Temperature Level: %d\n", temp);
   }
   // make a beep
   btsense_sound (440); NutSleep (1000); btsense_sound (0);

   for (;;) { NutSleep (5000); }
}
```

Below is a detailed description of the functions available in the btsense-library. The `adc2/adc2.h` additionally offers helper functions for using the ADC – see the source code for details.

1. `void btsense_init( btsense_revision_t rev );`

   Initialises the BTense library. This function has to be called before any other function of this library. Note that this function also enables the interrupt 6 (for the PIR) and installs an appropriate interrupt handler.

   rev The revision of the BTsense board, typically `BTSENSE_REVSION_1_1A`.

2. `u_short btsense_sample_light(void);`

   Samples a value from the light sensor. Note: This functions makes use of the ADC.

retval	*	The current (digitized) light value (0-1023)
	0xFFFF	if called without calling `btsense_init()` before

3. `u_char btsense_sample_motion(void);`

   Samples a value from the motion sensor.

retval	1	if a motion is currently detected
	0	if no motion is currently detected
	0xFF	if called without calling `btsense_init()` before

4. `int btsense_sample_temp(char* temp);`

   Samples a value from the temperature sensor.

temp (out) The current temperature in degrees Celsius

retval	`0`	if no error occured
	`>0`	the result of `TwMasterError()` if an I2C error occured
	`0xFFFF`	if called without calling `btsense_init()` before

5. `void (*btsense_sound) (u_short);`

   Outputs a sound on the beeper. Note: This is a pointer to a function. Its declaration is `void btsense_sound (u_short freq)`

   freq The frequency in Hz to output. Use *0* to turn beeper off.

Exercise 11.10. *Write a program that converts the light levels detected by the light sensor into a corresponding LED-meter, i.e., the brighter it is, the more LEDs light up.*

Optional Exercise 11.11. *Extend the program from ex. 11.10 to also sound the buzzer at different frequency levels, according to the detected light level.*

Exercise 11.12. *Write a program that indicates motion detection through LED or buzzer signaling. Instead of repeatedly polling the current value of the motion sensor, you should use set up an interrupt service routine to get triggered whenever the motion sensor's signal changes. Describe the output you observe.*

Exercise 11.13. *Extend the program from ex. 11.12 above to send motion events via the radio to a receiving node, which then prints out a corresponding line to the terminal.*

Optional Exercise 11.14. *Combine the motion sensors of several BTnodes in order to be able to detect the direction of motion, e.g., along a corridor or on both sides of a door. Both nodes should send motion events to a sink node, which then determines the direction of the motion and keeps an on-screen statistic (e.g., how many people entered and exited a certain room). Tip: In order to limit the area that the motion sensor covers, you can simply build a small paper cone and put it around the sensor.*

Optional Exercise 11.15. *Write a program that periodically reads out all available sensor values and sends them wirelessly to a sink node, which is connected to a laptop or PC via USB. Try to save power by grouping several measurements into a single transmission. The sink node should print out the comma-separated list of values to STDOUT, which can then be easily captured into a file by using the terminal's capture-to-file function and then displayed graphically in Excel, OpenOffice Spreadsheet, or GNUplot.*

Optional Exercise 11.16. *Extend the program from ex. 11.15 to work with multiple BTnodes, i.e., prefix each nodes measurements with a node ID. Try to minimize packet loss, e.g., by sending packets repeatedly. Use this setup to record one or two rooms over the course of an entire day. Prepare corresponding graphical plots.*