# Chapter 9

# The Chipcon Radio Module

## 9.1 Introduction

The BTnode features a Chipcon CC1000 radio module – the same radio that is used in the popular MICA mote platform, allowing those two platforms to communicate over a common radio channel. In contrast to the Bluetooth radio module (which was covered in one of the previous sections), the CC1000 is very simple: you can either send a radio signal, or listen for incoming signals from other nodes. As there is no automatic frequency hopping as in Bluetooth, we neither have discovery phases nor master-slave relationships. There is no default packet format or standardized access interface (like HCI or L2CAP) – using simple commands like "turn radio on" and "send this data", we can pretty much send out anything we please. However, this newfound freedom also comes at a price: Without the complex Bluetooth synchronization, we will need to take care of limiting access to the shared broadcast medium (i.e., the radio channel) ourselves. Otherwise, if two or more nodes in range of each other decide to send at the same time, their signals will interfere with each other (this is called a "collission") and none of the sent data can be received.[1]

Regulating access to a shared communication medium is done by a "medium access control" (MAC) layer.[2] The MAC layer is responsible for deciding who gets access to the physical layer at any one time. It also detects transmission errors and provides addressing capabilities, i.e., it verifies whether a received packet was actually intended for the receiving station. BTnut comes with one particular MAC-layer implementation for its Chipcon radio, based on Berkeley's B-MAC protocol [7]. The B-MAC protocol offers a very energy efficient way of regulating medium access, which is especially suited for sensor networks, called *clear channel assignment* (CCA). It also offers an equally low-power oriented approach to listening for incoming data, called *low power listening* (LPL). Just as any other MAC protocol, B-MAC detects transmission errors for us, handles acknowledgements, and provides an addressing scheme. Overall, however, B-MAC is a rather simple protocol that minimizes protocol overhead while providing essential support for low-power communication.[3]

## 9.2 Accessing the CC1000

Three main modules (and a number of helper modules)[4] implement control of the CC1000 radio on our BTnode. The low-level access to the radio (i.e., the physical layer) resides in *cc1000.c*, the B-MAC protocol (the data-link layer) is implemented in *bmac.c*, and the high-level routines for sending and receiving data are in *ccc.c*. This modular setup allows the use of multiple MAC protocols, though so far only a single one is available. Figure 9.1 gives an overview of the dependencies. Unless you want to program your own MAC-

[1] Note that they don't even have to be in range of each other, if a third, receiving node "sees" both of them. This is known as the *hidden terminal problem.*

[2] In the ISO/OSI network reference model, the physical layer (layer one) would be our Chipcon radio, while the MAC would be situated in layer two, the data link layer.

[3] Its authors explicitly encourage the implementation of more sophisticated MAC protocols on top of B-MAC [7].

[4] Specifically, the B-MAC protocol uses *cca.c* to implement the clear channel assignment, while *crc.c* provides CRC error checking.

layer, you will only need to include both *ccc.h* and *bmac.h*. The next three sections will explain initialization the radio, sending data, and receiving data.

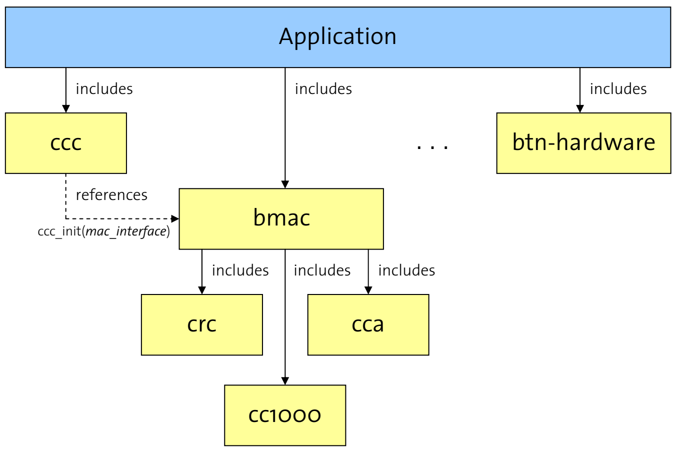

Figure 9.1: CC1000 Modules.

### 9.2.1 Initialization

Initializing the CC1000 radio is done in the `ccc_init` function, which takes as its single argument a *mac_interface* structure, i.e., a reference to a MAC protocol to be used for communication. Consequently, we will first need to initialize our MAC library, which will create a matching instance of such a *mac_interface* structure for us. The relevant code thus looks like this:

```
#include <cc/bmac.h>
#include <cc/ccc.h>

#define NODE_ADDRESS 0x0001;

static void init_radio (void) {
    int res;
    /* initialize bmac */
    res = bmac_init(NODE_ADDRESS);
    if (res != 0) { /* bmac initialization failed - halt system */ }
    bmac_enable_led(1);
    /* bmac_interface defined in bmac.h */
    res = ccc_init(&bmac_interface);
    if (res != 0) { /* cc1000 initialization failed - halt system */ }
}
```

Notice that we need to supply a node address for B-MAC initialization. This address will be used by the MAC layer to filter out packets addressed to other nodes, i.e., we will only receive packets addressed either directly to this node, or those sent to a broadcast address. More details about addresses can be found below.

The `bmac_enable_led` command activates LED feedback for sending and receiving, i.e., the BMAC layer will light the green (outermost) LED when listening, the blue (innermost) LED when sending or receiving, and the red LED in case of CRC errors.

**Exercise 67** *Write a program that activates the CC1000 as described above, including the BMAC LED activation, before going in an endless `NutSleep`. What do you observe? Add terminal access to your application and integrate the Nut/OS command set (using `nut_cmds_register_cmds`). Check the output of the `nut threads` command.*

### 9.2.2 Sending Data

Once we have initialized the radio, we can use the `ccc_send` command (part of *ccc.h*) to send out data.

```
#define MAX_PACKET_SIZE 8
#define PACKET_TYPE 0x01        /* application-specific, 0-255 */

pkt = new_ccc_packet(MAX_PACKET_SIZE);

void _cmd_send_ushort(char* arg) {
    int val;
    pkt->length = 4;

    if (sscanf(arg,"%u",&val)==1) {
        sprintf(pkt->data,"%u", val));
        if (ccc_send(BROADCAST_ADDR, PACKET_TYPE, pkt)) {
            /* send failed (<> 0 indicates error) */
        }
    }
}
```

`ccc_send` takes as input the intended receiver's address, the type of packet that should be sent, and the packet itself. Packets not only contain payload, but also source and destination information, an explicit size (`length`), as well as a *packet type*. We can use the `new_ccc_packet` function to obtain a pointer to an empty packet struct, with memory allocated up to the given size (`PACKET_SIZE` in our example code above). However, we still need to explicit specify the actual length of each packet that gets sent, by setting the `length` attribute accordingly.

**Exercise 68** *Lookup the source code of the `ccc_send` in the BTnut sources. How is sending data implemented? Why is `ccc_send` not doing the actual data transmission? Lookup the corresponding `*_send` function in `bmac.c` and explain.*

**Explanation *Addressing in B-MAC*:**
For each packet sent using `ccc_send`, a *destination address* must be given. The B-MAC implementation uses a two-tiered 16-bit address structure, composed of $2^{11}$ (i.e., 2048) *clusters* with $2^5 - 1$ (i.e., 31) individual addresses each. A reserved *broadcast address*, `BROADCAST_ADDR` (`0xFFFF`), can be used to address all nodes in all clusters. Each cluster (except for cluster 2047) also has a *multicast* address, which is simply the "highest" address in the cluster. Table 9.1 gives an overview.
In practice, the cluster address of a particular node does not matter much: As long as nodes are in range of each other, nodes from any cluster can send and receive data from nodes from any other cluster. Clusters are simply a means to form subgroups of nodes that can easily communicate among each other using a cluster-specific broadcast (called a "cluster-multicast"). Special care must be taken with such multicast addresses (i.e., addresses that are multiple of 32 minus one: 31, 63, 95, ...), as data sent to such an address will be received by all other nodes in this particular cluster. When accidentially assigning such an address to a node (e.g., using `bmac_init(63)`), all packets sent to it will also be delivered (by the B-MAC layer) to all other nodes in this particular cluster (e.g., 32 through 62 in this case). Also note that cluster 2047 does not have an individual multicast address, as `0xFFFF` is actually used as a broadcast address for *all* nodes.
In order not to accidentially assign multicast addresses to nodes, use the following macro to compose an address from separate node and cluster IDs:

```
#define address (node, cluster) (((cluster) << 5) | (node))
```

When using `ccc_send`, we will need to take care of properly packaging our data. In case of binary data, this means making sure that multi-byte data (e.g., 16-bit shorts) are put in a well-defined order, otherwise

| Address | Node ID | Cluster ID |
|---|---|---|
| 0x0000 | 0 | 0 |
| 0x001E | 30 | 0 |
| 0x001F | *ALL* | 0 |
| 0x0020 | 0 | 1 |
| 0x002E | 30 | 1 |
| 0x002F | *ALL* | 1 |
| ... | ... | ... |
| 0xFFC0 | 0 | 2046 |
| 0xFFDE | 30 | 2046 |
| 0xFFDF | *ALL* | 2046 |
| 0xFFE0 | 0 | 2047 |
| 0xFFFE | 30 | 2047 |
| 0xFFFF | *ALL* | *ALL* |

Table 9.1: *Cluster Addresses.* The B-MAC layer divides the 16-bit address space into clusters with 31 nodes each. One address per cluster is reserved for so-called *cluster-multicast*, while the highest address (`0xFFFF`) broadcasts to all nodes in all clusters.

the receiver might accidentially reverse those bytes during decoding. This is because not all microprocessors (nor compilers, for that matter) represent multi-byte values in the same order. Intel chips have traditionally arranged multi-byte values in memory by beginning with the *least significant byte* (LSB) first, i.e., the value `0x1234` stored at, say, memory address 0x3201, would have the value `34` at 0x3201 and value `12` at 0x3202. This is called "little-endian" order. Consequently, beginning with the *most significant byte* (MSB) first would store value `12` at 0x3201 and value `34` at 0x3202. This is called "big-endian" order. This "endianness" becomes crucial when exchanging multi-byte data (e.g., integers) between platforms, e.g., through binary files (an image) or over the network.[5]

**Explanation *Network Byte Order*:**
As long as the data we send is picked up by identical hardware running identical software built using the same compiler, we can ignore byte order, as both sender and receiver will use the same representation. However, for exchanging data between different platforms, or between software from different generations, vendors, or compilers, agreeing on a common byte order is crucial. For network exchanges (e.g., over Ethernet, but also wirelessly), the commonly agreed upon *network byte order* uses big-endianness. There are standard C-functions, `htons` (*host-to-network-short*) and `ntohs` (*network-to-host-short*), to convert between this network byte order (where the most significant byte is put first) and the "host byte order", i.e., whatever the current host's and/or used compiler's order is.

```
void _cmd_send_ushort(char* arg) {
    int val;
    pkt->length = 2;

    if (sscanf(arg,"%u",&val)==1) {
        // put two-byte value (in network order) into packet
        *((u_short*) &pkt->data[0]) = htons((u_short) val);
        // /* alternatively, do this manually: */
        // pkt->data[0] = val>>8;   // high byte
        // pkt->data[1] = val&0xFF; // low byte
        if (ccc_send(BROADCAST_ADDR, PACKET_TYPE, pkt)) {
            /* send failed (<> 0 indicates error) */
        }
    }
}
```

[5]Notice that the concept of endianness is less important with regards to the individual *bits*, as access to bits is usually not given directly, but through well defined logical operators that work independant of the actual representation.

The second argument to `ccc_send` is a *packet types.* Packet types allow us to simplify packet receiption, as each different type can trigger a different reception function, so-called *packet handlers.* This is explained in the following section.

### 9.2.3 Receiving Data – The `ccc_rec` Receiver Thread

As we have seen in exercise 67 above, calling `ccc_init` automatically activates a `ccc_rec` thread that will repeatedly listen for incoming packets on the CC1000 radio. The `ccc_rec` thread is started with the relatively high priority of 16, in order to prevent delaying packet reception. This thread listens on a specific event handler for incoming data packets (as signaled by the B-MAC *low power listening* implementation), and in turn calls type-specific *packet handlers* for each received packet.

Packet handlers are registered using the `ccc_register_packet_handler` function and must implement the `void pkt_handler(ccc_packet_t *pkt)` interface. An example is shown below:

```
void pkt_handler(ccc_packet_t *pkt) {
    u_short i;
    if (sscanf(pkt->data,"%u",&i) == 1) {
        printf ("Received Value: '%u'\n", i);
    } else { /* error parsing stringified data packet */ }
}

#define PACKET_TYPE  0x01

int main (void) {
    ...
    ccc_register_packet_handler(PACKET_TYPE, pkt_handler);
    ...
}
```

**Explanation *B-MAC Packet Handlers*:**
A packet handler is always assigned to a single packet type, and will thus only be called when the `ccc_rec` thread not only received a properly *addressed* packet, but also one with a matching *type.* These types are (currently) application specificy, i.e., you need to define the necessary type IDs (from 0–255) yourself. For example, an application might decide to define several such types in order to differentiate between status messages, sensory data, and routing information:

```
#define SENSOR_DATA  0x01
#define ECHO_REQUEST 0x04
#define ECHO_REPLY   0x05
#define ROUTING_TBL  0x09
```

**Exercise 69** *Write a small chat program, consisting of a terminal command "say", which simply sends off its arguments via broadcast, and a corresponding packet handler that listens for such packets and writes their source and contents to stdout in a chat-program fashion (e.g., "[45] says: Hello world").*

**Optional Exercise 70** *Extend the program from ex. 69 to take an address for the "say" command (e.g., "say 345 hello world"). Use "say all" or an additional "shout" command to initiate broadcasts.*

**Exercise 71** *Write a program that periodically (e.g., every 2-4 seconds) sends out* `PING_TYPE` *packets to the broadcast address. A specific packet-handler for these packets should print out a brief message everytime it receives such a packet. Install your program on two BTnodes and observe them on two separate terminals.*

**Explanation *The B-MAC Packet struct*:**
A Chipcon packet is defined as shown below. It not only contains the actual packet payload, but also information about the packet's sender (`pkt->src`).

```
struct ccc_packet_t {
    /** source of the packet */
    u_short src;
    /** destination of the packet */
    u_short dst;
    /** payload length */
    u_short length;
    /** packet type */
    u_char type;
    /** payload data */
    u_char data[0];
}
```

**Exercise 72** *Change your program from ex. 71 so that* `PING_TYPE` *packets are only sent out after no packet has been received for some time (use a timer). Upon reception of a* `PING_TYPE` *package, a* `PONG_TYPE` *package should be sent out, and vice versa (make sure that the timer is reset after a packet has been received). Print corresponding "ping" and "pong" messages upon sending each packet type. Watch the output of both nodes over two separate terminals, occasionally reseting one node to see whether your program works in both directions. Don't forget to reset the timer upon packet reception.*

**Attention:** CC1000 reception[6] using *battery power* is extremely unreliable in the current BTnut release (1.8). This is a software problem and should hopefully be fixed in future releases. Until then, we recommend using either USB power when trying to receive data of the CC1000, or explicitly disabling the sleep mode using `NutThreadSetSleepMode(SLEEP_MODE_NONE);`.

## 9.3 Advanced Topics

Two interesting features of the CC1000 radio are that both its frequency and its power output can easily be adjusted, allowing for example frequency-hopping schemes or minimal-power transmissions.

### 9.3.1 Power Control

Transmission power can be set using the `cc1000_set_RF_power` function, which can be found in *cc1000.h*. It accepts a value from 0 to 255, with 0 being no power, 1 being the minimal power, and 255 representing maximum transmission power.

```
#include <cc/cc1000.h>

void rfpower_cmd(char *arg)
{
        u_short num;
        u_char num2;

    if (( sscanf(arg, "%u", &num) != 1 ) || num > 255 )
    {
        printf("usage: rfpower <0..255>\n");
        cc1000_get_RF_power( &num2 );
        printf( "Current RF power level is %u.\n", num2 );
        return;
```

[6] *Sending* data, however, works fine both under battery and USB power.

```
    }

    printf( "Setting RF power to %u...\n", num );
    cc1000_set_RF_power( num );
}
```

**Exercise 73** *Write a program to measure the transmission distance for different power levels, i.e., find out how far away a signal sent with tranmission power 1, 2, or 3 can be still received, or how much power is necessary to contact a node at, say, 5 meter distance, or in another room.*

**Optional Exercise 74** *Change your program from ex. 72 so that `PING_TYPE` packets include as payload the sender's current power level, initially set to its maximum of 255. Upon receiving such a packet, the receiver should print this information to STDOUT and acknowledge it with a `PONG_TYPE` packet. Receiving a `PONG_TYPE` packet should lower a sender's transmission power before sending out another `PING_TYPE` packet. Take two nodes and measure various distances that certain power levels can achieve.*

**Optional Exercise 75** *Extend your program from ex. 73 so that it will build a neighborhood table of the closest $n$ neighbors and their "power-level" distances.*

**Optional Exercise 76** *Implement a multi-hop flooding protocol on the BTnodes. You will need to set the power level to a reasonably small number, e.g., 2-3. All packets will be sent to the broadcast address, and contain a packet ID that allows nodes to detect packets they already sent (in order to avoide reduplicating packets). Test your protocol by flooding your network with a certain LED pattern, i.e., use a terminal to initiate a certain LED pattern, which will be set on each receiving node (before sending the packet on to other nodes).*

### 9.3.2 Frequency Control

The CC1000 radio supports a wide variety of frequencies, primarily in the ISM-bands[7] at 315, 433, 868, and 915 MHz. However, it can be also tuned to almost any frequency between 300 and 1000 Mhz [2].

During B-MAC initialization, the radio is set to 868.5 MHz. However, if desired, one can use the `cc1000_set_frequency` function (in `cc1000.h`) to set it to pretty much any frequency within the 915 and 868 MHz bands.[8] Note that the `cc1000_set_frequency` function takes the desired frequency in Hz (to avoid fractional values) and returns the actual frequency that has been set (as not all frequencies can be achieved on the CC1000):

```
#include <cc/bmac.h>
#include <cc/ccc.h>
#include <cc/cc1000_defs.h>
#include <cc/cc1000.h>

#define address (node, cluster) (((cluster) << 5) | (node))
#define NODE_ADDRESS address (27, 34); /* node #27, cluster #34 */

static void init_radio (uint32_t desiredFrq) {
    uint32_t actualFrq;
    /* This does *not* yet work with B-MAC in BTnut 1.8!! */
    bmac_init(NODE_ADDRESS);   /* inits radio to 868.5 MHz */
    actualFrq = cc1000_set_frequency(desiredFrq); /* reset radio frequency */
    printf ("[init_radio] set cc1000 to %lu Hz\n", actualFrq);
    ccc_init(&bmac_interface);
}
```

[7]ISM stands for "Industrial, Scientific, Medial" and denotes frequency spectrums that can be used without acquiring a license first.

[8]This does not yet work together with the B-MAC protocol in the current BTnut release (1.8).

This is still considered experimental, compatibility with `bmac.h` will hopefully soon be established – you might want to try using the latest version from CVS to see if it has already been fixed. If it is, you could try the following exercises (these will most likely *not* work with release 1.8!):

**Exercise 77** *Manually set the frequency of one of your nodes to 868.5 Mhz, the B-MAC default frequency. Observe the actual frequency that the CC1000 gets tuned to (as returned by* `cc1000_set_frequency`*) and compare. Can you still receive packets sent from this node on a node that is not manually tuned? Explain why this works or does not work.*

**Optional Exercise 78** *Extend your program from ex. 72 so that each packet also contains the frequency on which the next packet should be sent (use* `cc1000_set_frequency` *together with a small set of predefined frequencies that both programs share). Take packet-loss into account, i.e., make sure that a lost packet will not put the two nodes permanently out of synch.*

### 9.3.3 Measuring Signal Strength

The CC1000 additionally offers access to RSSI (*Receive Signal Strength Indication*) information. However, as this data is available only in analog form, we will need to use on of the available ADCs (digital/analog converter) on the Atmega128 in order to obtain a digital readout. Access and usage of the ADCs is covered in the sensor chapter of this tutorial (see chapter 10). The many layers between our main program and the BTnut CC1000 modules further complicate matters: by the time one of our packet handlers gets called, packet reception has already finished, so reading out RSSI data at this point will most likely only measure the channel's background noise.[9] Even if noise levels are all you want, measuring RSSI in your own program will most certainly interfere with B-MAC's CCA routines, requiring careful coordination of ADC registers in order not to mix up different RSSI readings.

**Optional Exercise 79** *Where would we need to measure RSSI in order to obtain the signal strength with which a particular packet was received? Look trough the three modules* ccc.c*,* bmac.c*, and* cc1000.c *and speculate on the best place to add RSSI measurements to a data packet's struct.*

[9]B-MAC's *clear channel assignment* (CCA) feature actually requires measuring the current noise level on the channel, which is implemented by averaging a number of RSSI measurement. See the corresponding BTnut source code in `btnut/cc/cca.c`.