TAISC hands-on tutorial

27/01/2015

Introduction

1 Hardware

The hardware we will be using for this tutorial is the RM090 sensor node. This sensor node is equipped with an MSP430 microcontroller and a CC2520 radio. Additionally the node also contains external memory, two push-buttons and a temperature/humidity sensor. During this tutorial we will only be using the buttons to trigger the transmission of a predefined packet. In a real life application we would transmit more interesting data such as sensor measurements but that is outside the scope of this tutorial.

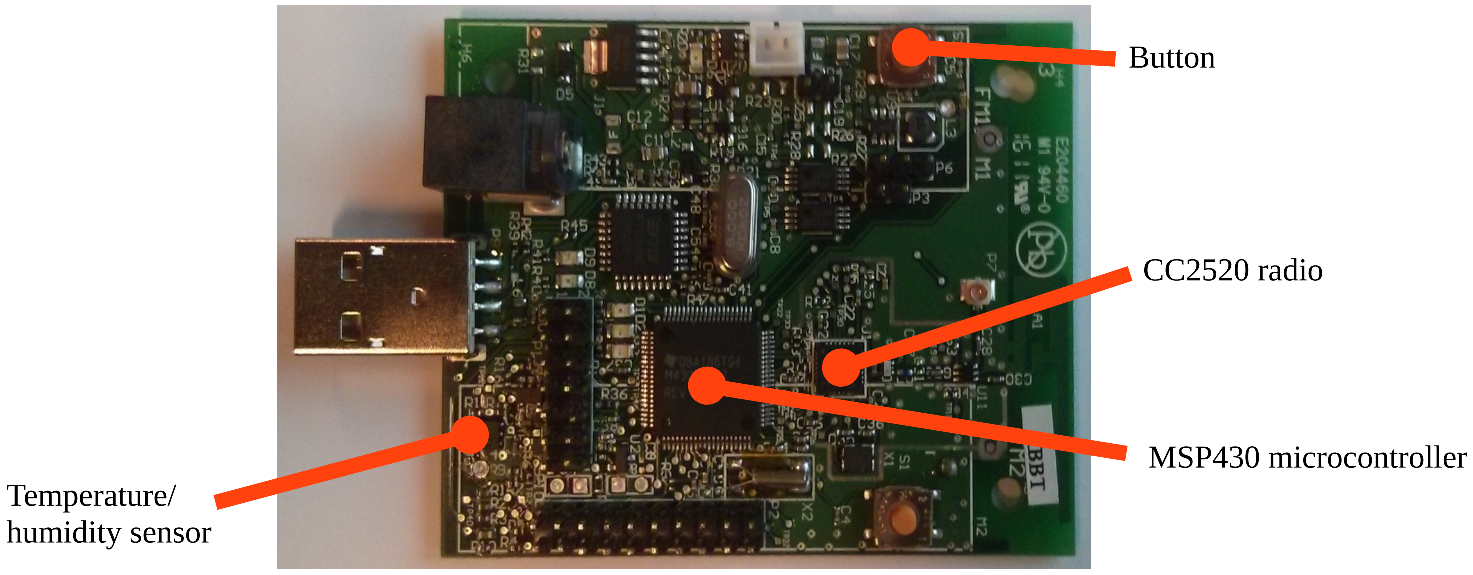


Figure 1: RM090 sensor node

The node will be powered by USB power and a built-in USB to serial converter will allow programming and communicating with the microcontroller.

2 Setup

Each group will have at its disposal 2 RM090 sensornodes. One node will be used to transmit packets, while the other node can be used as a sniffer to monitor all wireless traffic.

The transmitting node will run an application we have prepared that generates 100 packets with an interval of 100 ms when you push on of the buttons. We will start with a very simple MAC but during the exercises you will be able to change this MAC.

In the front of the room we will have a receiving node that will receive the packets sent by the sending node of each group. This RX node forwards this information to a host PC that displays traffic statistics on the projector screen.

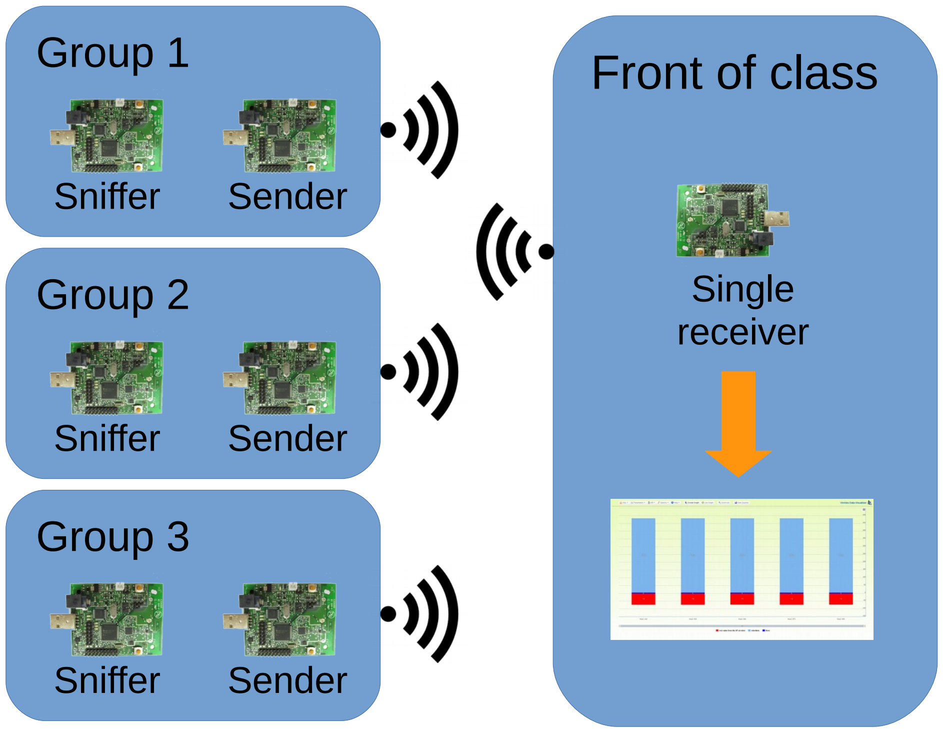


Figure 2: Schematic overview of the setup

3 Software

Most of the software used in this tutorial will be called from the available makefiles and thus you should not need to call them directly. However to provide some insight the most important utilities are listed here:

- **msp430-gcc**: C compiler for the MSP430 microcontroller. Will generate a full binary that can be loaded and executed on the senornode.

- **ibcn-f5x-tos-bsl**: Python script that takes the binary generated by msp430-gcc and programs the microcontroller using the USB to serial link.

- **parser.py**: Python script that will convert the human-readable chain description into bytecode that can be understood by TAISC

- **snifferproxy**: This program will forward the packets received by the sniffer node to a UDP socket on the host

- **wireshark**: GUI program to dissect the packets forwarded by the snifferproxy

GUI tools like geany or qtcreator can be used to edit the source code of the application that we will modify during this tutorial

Tutorial

Before you get started you will have to check-out a copy of the code we will be using during the tutorial:

```
git clone https://pdvalck@bitbucket.org/pdvalck/taisc-tutorial.git
```

During the tutorial it is important to know which node you are using when executing commands. You can obtain a list of all nodes connected to your PC by executing:

```
ls -la /dev/ttyUSB*
```

This should display a list containing '/dev/ttyUSB0' and '/dev/ttyUSB1'. If your devices have another number use that name instead of the ones used throughout the tutorial. We will assume node 0 to be the sniffing node while node 1 will be the sender.

1 Test your sensornode

In this exercise you will gain some familiarity with the hardware and software we will be using.

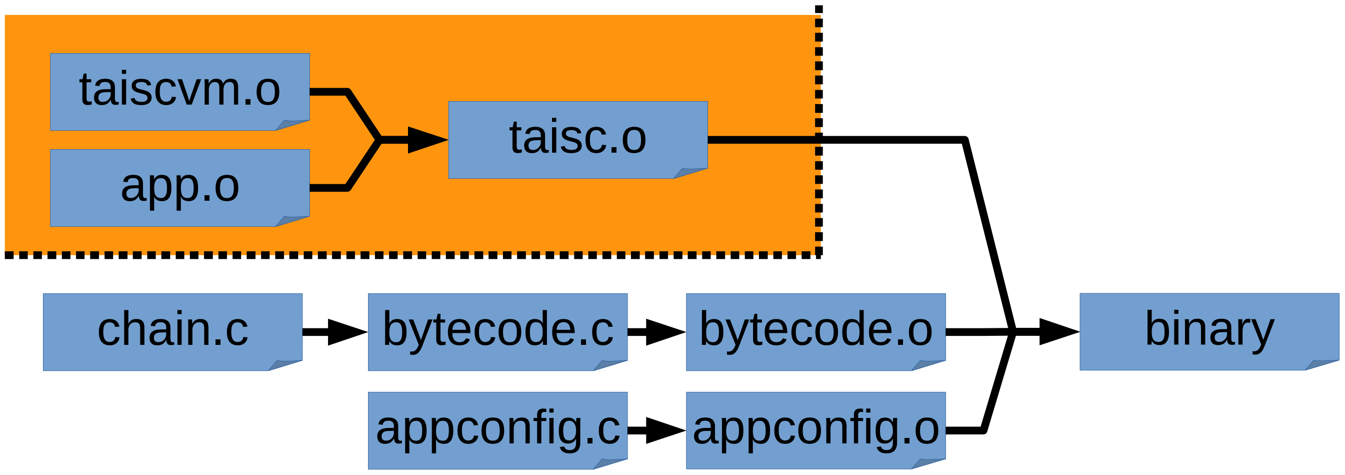


Figure 3: Build steps of the final binary (the orange steps were done beforehand)

The files for this exercise can be found in the folder 'exercise-01'. There is an empty 'chain.c' and an object file 'taisc.o' containing the application and TAISC implementation. The 'appconfig.c' file is used to specify your group number in the transmitted packets so make sure to change this accordingly.

When compiling, the 'chain.c' will be parsed into a file containing the binary representation of the chains. This file is then compiled by the msp430 compiler and linked with the taisc library. The result is a single binary that can be loaded and executed on the sensornode.

All this can be done by executing:

```
make
```

If this produces no errors you can load your program on the sensornode by executing:

```
make install DEV=/dev/ttyUSB0
```

Once this command has finished you can test your program by pressing one of the buttons. You will see that the LEDs start blinking, indicating that the application is trying to transmit packets. However, since no MAC was specified, no packets are actually transmitted or received.

2 Send some packets

In this exercise you will use the same files as for the previous one, but instead of an empty 'chains.c' a simple 'chains.c' is provided in 'exercise-02'. This chain will simply transmit a packet once it is delivered by the application.

Have a look at this file and try to compile it:

```
make
```

Loading the program is again:

```
make install DEV=/dev/ttyUSB0
```

If you now press the button the application will start generating packets and the simple MAC will transmit them. After a few seconds you should see the statistics changing on the projected screen.

3 Use the sniffer

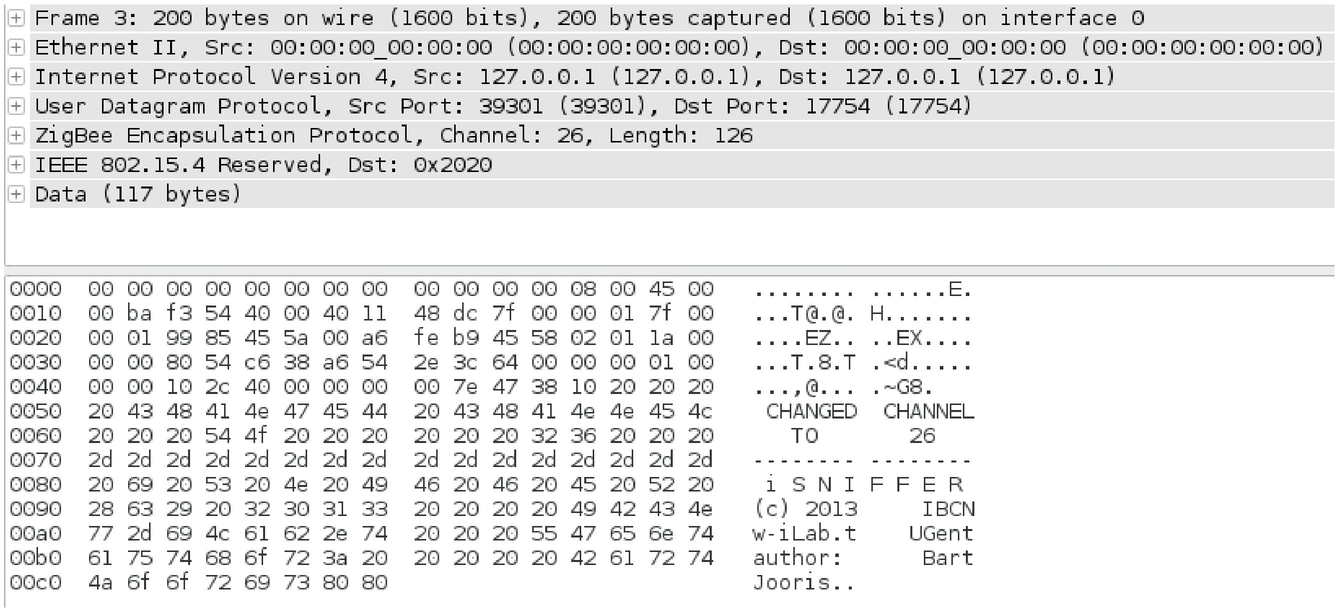
While the statistics in front of the class provide some feedback when your solution is working, it only provides a very limited amount of information. It would be much more interesting to be able to see each packet as it is transmitted. On a physical level this could be done by doing spectrum sensing, but since we have an additional sensor node on hand we might as well use it.

In the folder 'sniffer' you will find a script called 'run_sniffer.sh' along with a hex file and another executable. Run this script with:

```
./run_sniffer.sh /dev/ttyUSB1
```

This will program the sensornode with the sniffing code and start a proxy that forwards all packets received by the sensornode to your PC.

To inspect these packets you can start **wireshark** and capture on the loopback interface. If there is too much other traffic on this interface you can filter on UDP port 17754.



```
⊞ Frame 3: 200 bytes on wire (1600 bits), 200 bytes captured (1600 bits) on interface 0
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
⊞ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ User Datagram Protocol, Src Port: 39301 (39301), Dst Port: 17754 (17754)
⊞ ZigBee Encapsulation Protocol, Channel: 26, Length: 126
⊞ IEEE 802.15.4 Reserved, Dst: 0x2020
⊞ Data (117 bytes)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 ba f3 54 40 00 40 11  48 dc 7f 00 00 01 7f 00   ...T@.@. H.......
0020  00 01 99 85 45 5a 00 a6  fe b9 45 58 02 01 1a 00   ....EZ.. ..EX....
0030  00 00 80 54 c6 38 a6 54  2e 3c 64 00 00 00 01 00   ...T.8.T .<d.....
0040  00 00 10 2c 40 00 00 00  00 7e 47 38 10 20 20 20   ...,@... .~G8.
0050  20 43 48 41 4e 47 45 44  20 43 48 41 4e 4e 45 4c    CHANGED  CHANNEL
0060  20 20 20 54 4f 20 20 20  20 20 20 32 36 20 20 20      TO       26
0070  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d   -------- --------
0080  20 69 20 53 20 4e 20 49  46 20 46 20 45 20 52 20    i S N I F F E R
0090  28 63 29 20 32 30 31 33  20 20 20 20 49 42 43 4e   (c) 2013     IBCN
00a0  77 2d 69 4c 61 62 2e 74  20 20 20 55 47 65 6e 74   w-iLab.t    UGent
00b0  61 75 74 68 6f 72 3a 20  20 20 20 20 42 61 72 74   author:      Bart
00c0  4a 6f 6f 72 69 73 80 80                            Jooris..
```

Figure 4: Wireshark displaying the initial sniffing packet

You can change the channel the sniffer is listening on by pressing the buttons of the node. If the sniffer seems to hang, try to change the channel and back.

4 Change the channel

In the previous exercise the sensornode was programmed to transmit on the default channel of 27. Now we have another receiver that is listening on channel 28. In order to address this receiver, you will thus have to change the channel before you transmit a packet.

To do this you can copy the files from exercise 2 and modify the transmitting chain so that it will set the correct channel when transmitting a packet. To get an idea how to get started have a look at the 'Instructions' section at the end of this document.

Running is again:

```
make
make install DEV=/dev/ttyUSB0
```

Don't forget that you can change the channel of your sniffer by pressing the buttons.

5 Deal with an interferer

For this exercise we will use channel 29. The setup is the same as before, but now we have introduced an interferer on this channel: another node close to the receiving node will be transmitting data and thus be interfering with any packets you try to transmit. To see this interferer in action, move your sniffer to channel 29.

When two packets are sent at the same time, the signal strength of the packets as well as their relative timing will influence the reception of these packets:

- If a packet with a sufficiently high signal strength is interrupted by a low signal strength packet, there is a chance that the strong packet will still be correctly received.

- On the other hand, a low signal strength packet interrupted by a high signal strength packet will never be correctly received. Because the receiver is already synchronized with the weak packet, the strong packet will also be lost.

As our interferer is located right next to the receiving node, it is not possible for your transmit node to overpower our interferer.

To solve this problem you can introduce Clear Channel Assessment in your transmit chain: before transmitting a packet you have to listen if the medium is free. Only when that is true can you start sending. While this is only a very rudimentary collision avoidance technique, it should improve your results significantly.

6 Wait for your turn

In this final exercise we switch from an opportunistic channel access method to a coordinated system on channel 30. The central receiver will now transmit beacons with a group ID and then accept one message from that group.

Examine the behavior of this beacon by using the sniffer and try to adapt your transmitting chain to conform to this behavior. If you transmit outside of your turn you will lose some points.

Instructions

In this section the TAISC commands needed to complete this tutorial are explained. This is only a subset of the available commands but they should be sufficient.

While the language used to describe a TAISC chain is very similar to C, there are a few limitations to keep in mind due to the current implementation restrictions:

- const parameters must be provided as integer literals. So

```
function(10);
```

instead of

```
const uint8_t myliteral = 10;
function(myliteral);
```

- Non-const parameters must be provided by reference

```
uint8_t myliteral = 10;
function(myliteral);
```

instead of

```
function(10);
```

setChannel(uint8_t channel)

This command changes the channel the radio is currently operating on. Valid channels range from 7 to 31.

waitForTrigger(const uint16_t timeout, TAISC_eventT event)

Wait at most *timeout* µs for *event* to happen. This instruction should be used in an if-clause.

```
if(waitForTrigger(500, TAISC_EVENT_dataplane_txFrameAvailable)){

     // Executed if event was triggered

}

// If not skip here
```

The events used in this tutorial are:

TAISC_EVENT_dataplane_txFrameAvailable: a frame is available for sending

TAISC_EVENT_dataplane_rxFrameAvailable: a frame was received

TAISC_EVENT_cc2520_stateOfMediumChanged: the medium became idle according to the CCA mechanism (see next instruction)

controlCCA(CC2520_CCA mode, CC2520_CCA_HYST hyst, int8_t threshold)

Configures the CCA settings of the radio.

Mode indicates the CCA method:

CC2520_CCA_DISABLED: CCA is disabled

CC2520_CCA_THRESHOLD_HYST: RSSI based CCA

CC2520_CCA_NO_FRAME_RECEPTION: CCA is low when receiving a packet

CC2520_CCA_HYBRID: A combination of the previous 2 CCA methods

Hyst is the hysteresis value used when doing RSSI based CCA (to avoid constant state changes). Valid values are: CC2520_CCA_HYST_[0 to 7]

Threshold is the threshold value used when doing RSSI based CCA

For this tutorial, RSSI based CCA with a hysteresis of 3 and a threshold value of -85 should be sufficient.

check(uint16_t buffer1, uint8_t offset1,
uint16_t buffer2, uint8_t offset2,
uint16_t maskbuffer, uint8_t maskoffset,
uint16_t size)

This instruction will compare *size* bytes from *buffer1* and *buffer2* while applying a mask indicated by *maskbuffer*.

The different buffers are specified by a buffer address and an offset within this buffer. So to access the 10th byte of the last receive packet you would use (TAISC_RAM.currentRxFrame, 10). To access variables you have defined locally you can use:

```
uint8_t myChar = 20;

uint8_t myMask = 255;

check(TAISC_RAM.currentRxFrame, 10,heap, &myChar, heap, &myMask,1)
```

This will compare the 10th byte of the last received frame with 20 (after masking the bytes with 255).