Sheffield Hallam University

Department of Engineering

BEng (Hons) Computer and Network Engineering

BEng (Hons) Electrical and Electronic Engineering

Activity ID		Activity Title			Laboratory Room No.	Level
Lab 104		Basics of WPANs with XBees			4302	6
Term	**Duration [hrs]**	**Group Size**	**Max Total Students**	**Date of approval/review**	**Lead Academic**	
1	8	2	25	09-18	Alex Shenfield	

Equipment (per student/group)

Number	Item
1	STM32F7 kit
3	XBee radio modules
3	XBee USB / regulator adaptors
2	XBee sensor nodes

Learning Outcomes

	Learning Outcome
3	Design, implement and test embedded networked systems, written in a high level programming language such as C/C++/ Java using appropriate interface devices from an initial specification through to validation
4	Demonstrate knowledge of the various tools and technologies available to develop and test an embedded networked system

Basics of Wireless Personal Area Networks with XBees

Introduction

Computing is about more than the PC on your desktop! Embedded devices are everywhere – from wireless telecommunications infrastructure points to electronic point of sale terminals. One definition of an embedded system is:

> "An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints."
>
> (http://en.wikipedia.org/wiki/Embedded_system)

In the laboratory sessions for this module you are going to be introduced to the STM32F7 discovery board – a powerful ARM Cortex M7 based microcontroller platform capable of prototyping advanced embedded systems designs. The STM32F7 discovery board includes advanced functionality such as Ethernet connectivity, UART over the USB connection, an LCD screen, and a micro-SD slot. Appendix B shows the various pins that are broken out from the STM32F7 discovery board (onto the Arduino form factor header), and Appendix C provides a schematic showing how these map to the headers on the SHU base board.

More and more embedded devices and applications now require the ability to talk to other devices and computers that are connected over a network. Whilst historically this may have been done at a local level using serial bus protocols (such as CAN bus) and running appropriate wires between devices, more and more developers are choosing to utilise existing network infrastructure and wireless protocols to achieve this communication. Not only does this reduce the amount of installation effort required for distributed embedded applications, but it also potentially allows users of those applications to manage them remotely via the internet.

In this lab session we are going to explore the use of small, low power, low data rate radios based around the 802.15.4 / Zigbee standard to enable both point-to-point and multicast communication. These sort of devices enable intelligent routing and mesh networking strategies over a short range to provide a simple way of deploying wireless sensor nodes. One example application (shown in Figure 1) is in home automation.

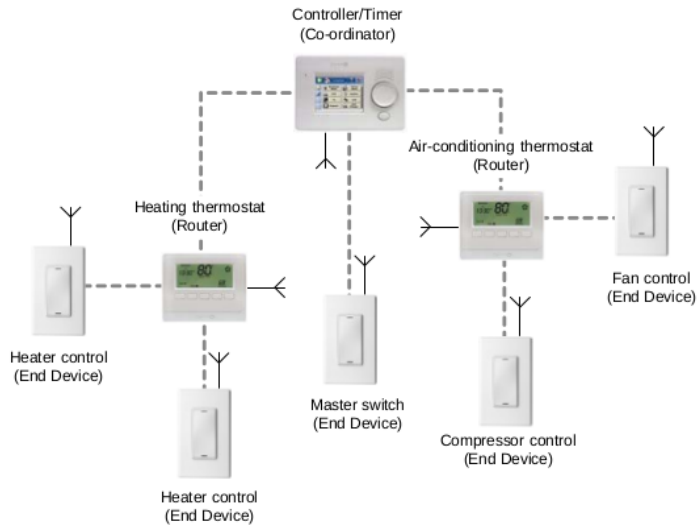


Figure 1 – A distributed home automation system

Bibliography

There are no essential bibliographic resources for this laboratory session aside from this tutorial sheet. However the following websites and tutorials may be of help (especially if you haven't done much electronics previously or your digital logic and/or programming is a bit rusty):

- The hands-on xbee lab manual – J. A. Titus[1]
- http://ftp1.digi.com/support/documentation/90000976_V.pdf
- http://www.cs.indiana.edu/~geobrown/book.pdf[2]
- https://visualgdb.com/tutorials/arm/stm32/
- http://www.keil.com/appnotes/files/apnt_280.pdf
- https://developer.mbed.org/platforms/ST-Discovery-F746NG/

1 An electronic version of this is available from the SHU library gateway
2 Note: this book is for a slightly different board – however, much of the material is relevant to the STM32F7 discovery

Methodology

Check that you have all the necessary equipment (see Figure 2)!

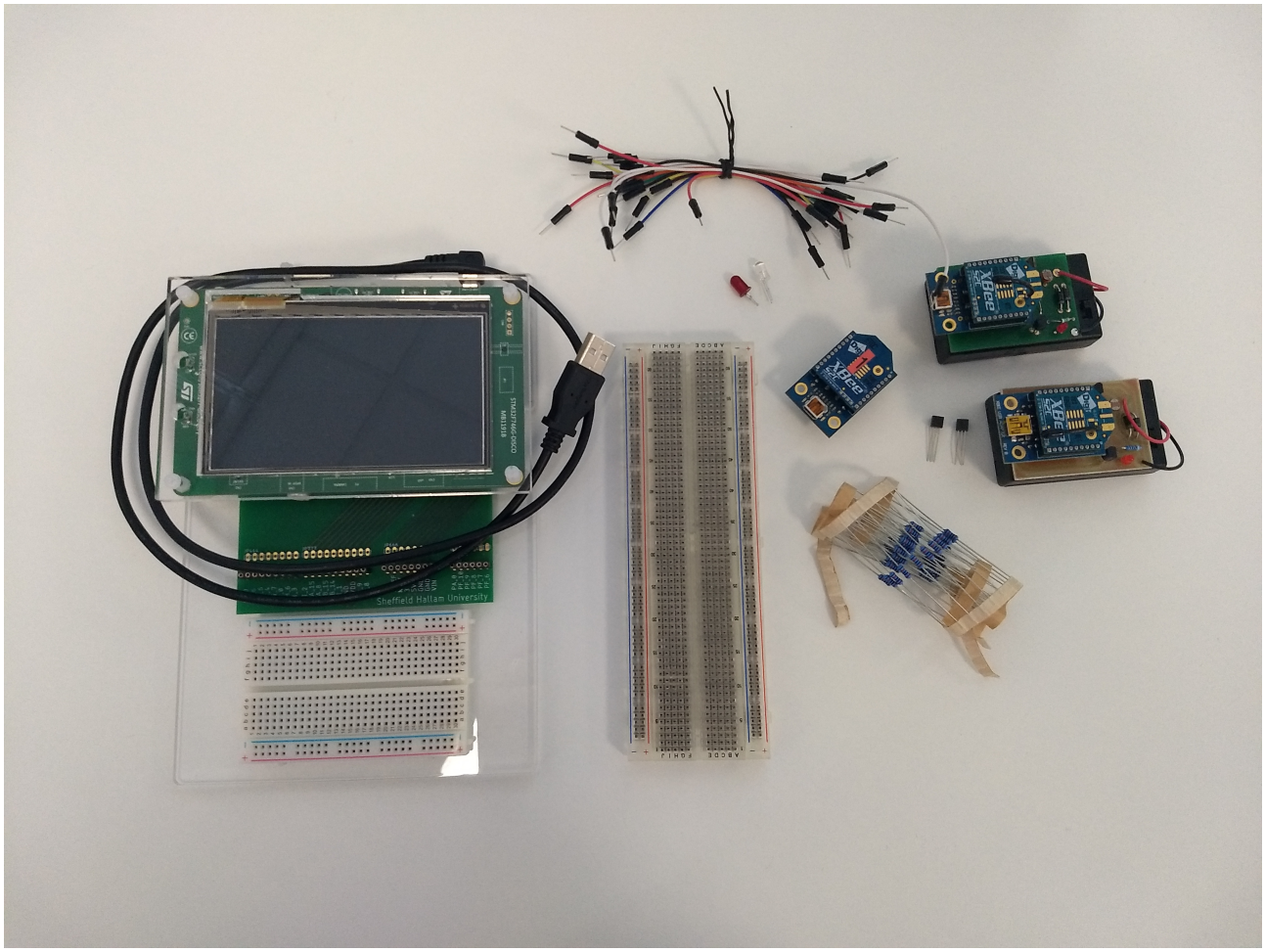


Figure 2 – A selection of the equipment for this lab

Task 1 – An introduction to XBee radios and the XCTU software

This task aims to familiarise you with the XBee radios and the XCTU software used to configure them. XBee radios are low power, low data rate radios from Digi International that conform to 802.15.4 and Zigbee standards. They are connected to a PC through a simple built-in Universal Asynchronous Receiver Transmitter (UART) so are easy to connect to hardware such as embedded microcontrollers.

Connect the XBee radio to the USB adapter and cable (as shown in Figure 3) and plug into the computer (if possible it is best to use the USB ports that are physically located on the computer rather than extension ports on monitors as they are a potential source of trouble).

Figure 3 – The XBee radio attached to a USB adapter

Ensure that the XBee module is firmly inserted into the socket strips on the adapter board and that the placement of the module is correct. Please be very careful when inserting and removing the XBee modules as it easy to bend / break the legs. Common mistakes when inserting the XBee module into the adapter are shown in Figure 4.

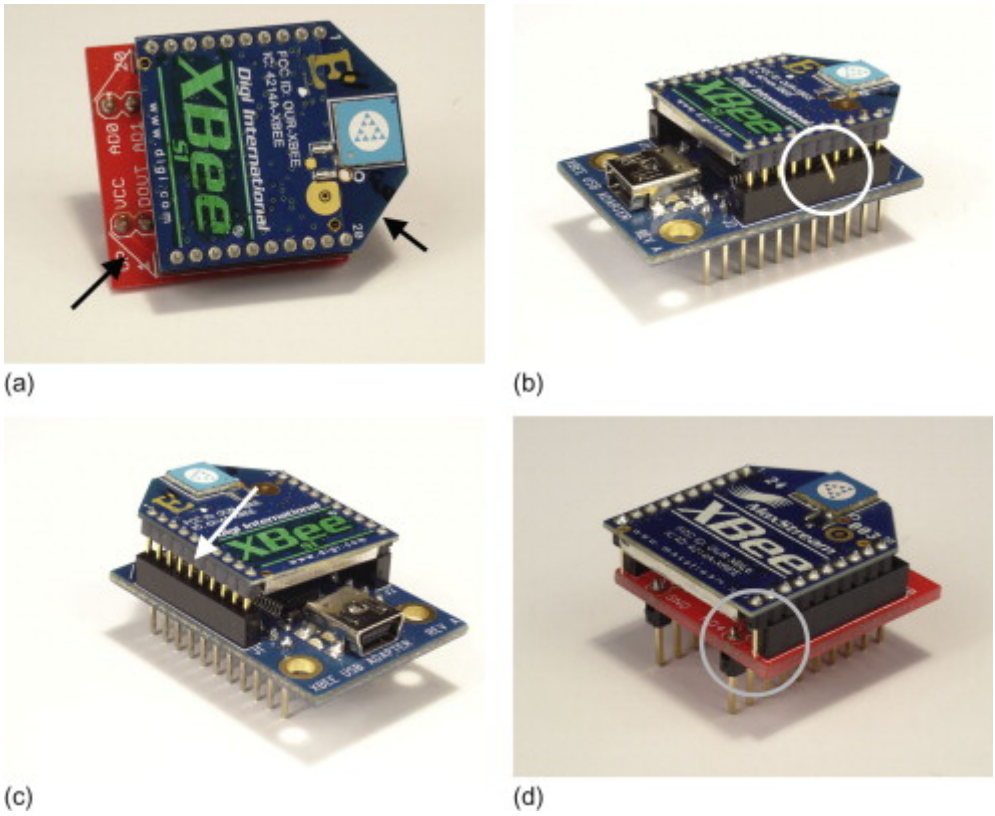


Figure 4 – Don't do this ...[3]

We can then fire up the XCTU software (see Figure 5) and connect to the XBee radio.

3 Taken from “The hands-on xbee lab manual”

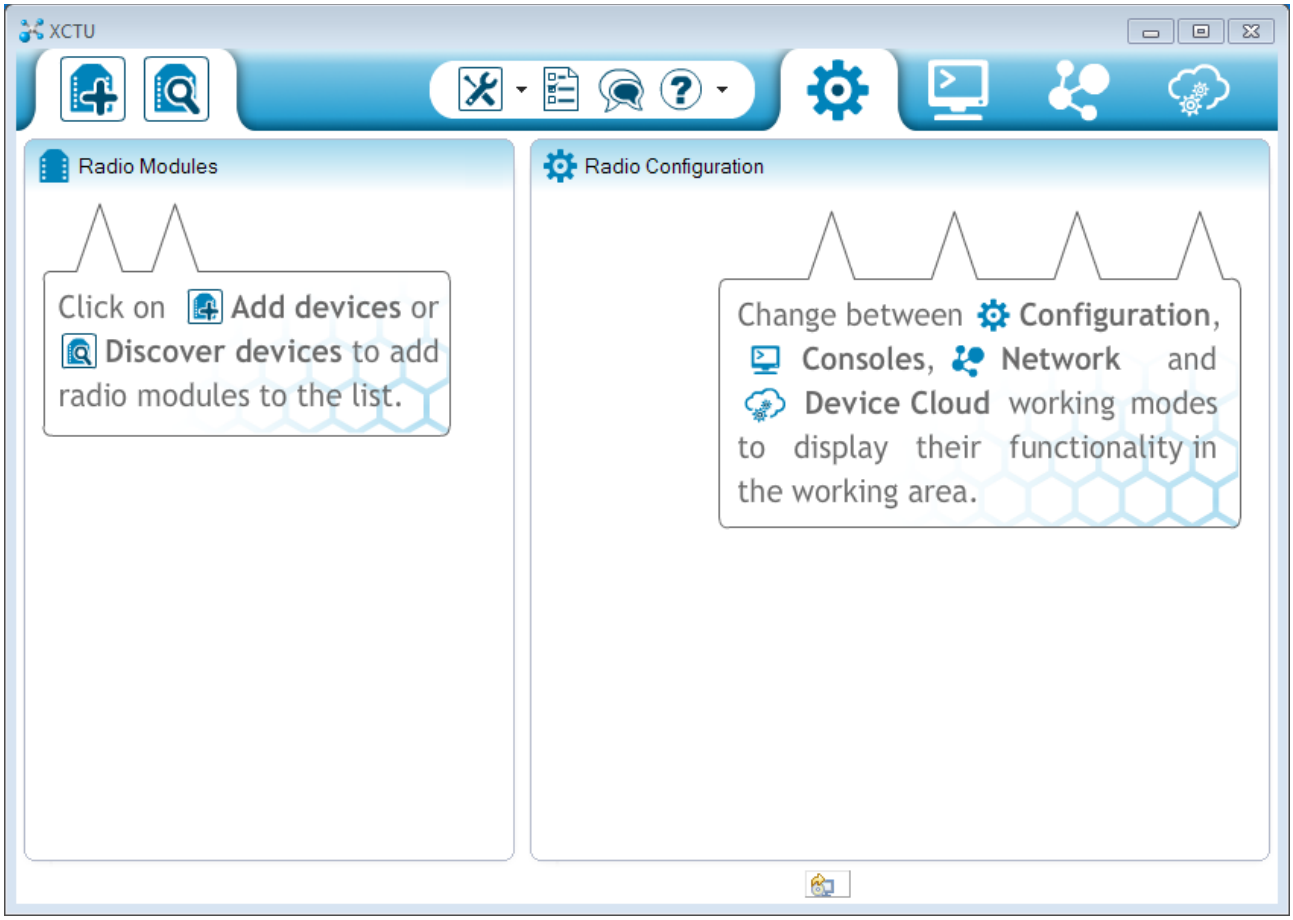


Figure 5 – The XCTU splash screen

The latest version of XCTU can scan multiple serial ports and configuration settings to find your XBee radio (using the “Discover devices” option in Figure 5, above) – be aware that the more of these you select, the longer it will take to find your radio! Figure 6 shows the configuration parameter selection process for searching for your XBee module. By default the XBee radios use the configuration settings shown in Table 1.

Baudrate	9600
Flow Control	None
Data Bits	8
Parity	None
Stop Bits	1

Table 1 – Default XBee configuration values

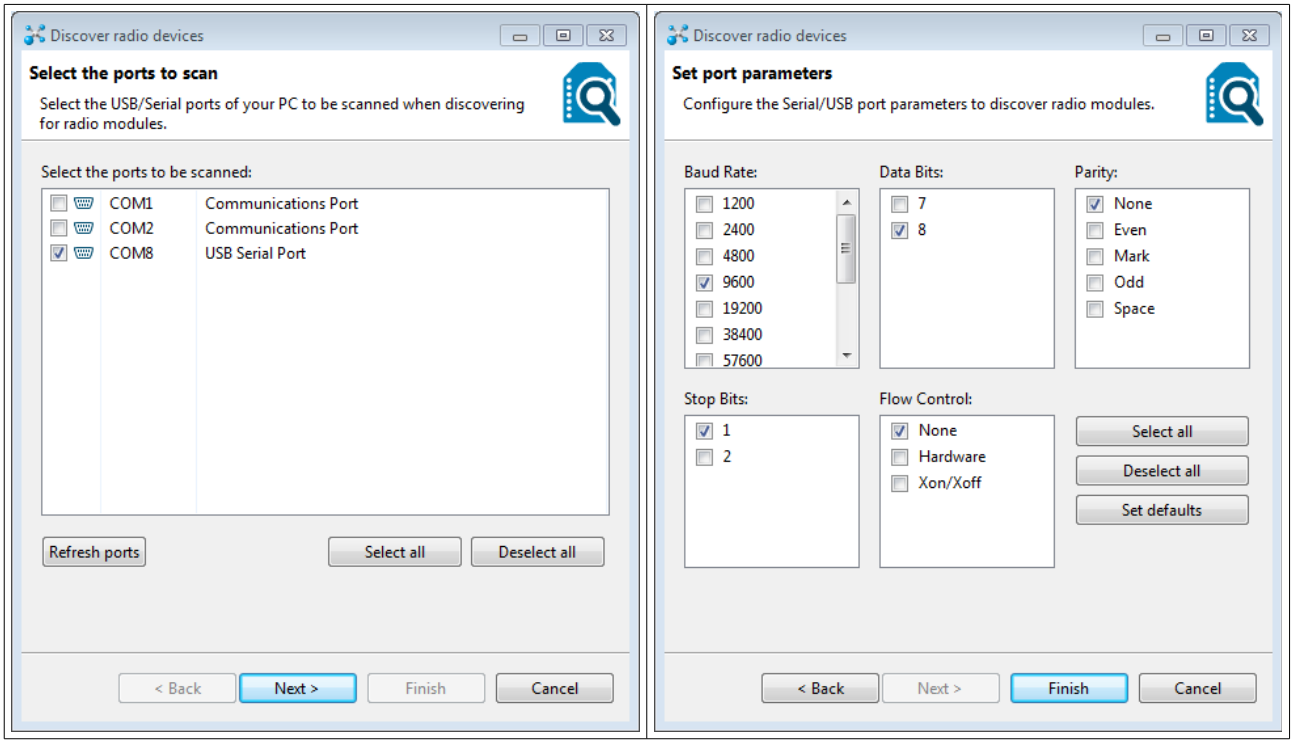


Figure 6 – Selecting configuration parameters to search in XCTU

Once you have found your XBee radio module you should add the selected device and choose it in the left-hand pane – this will bring up the radio configuration details (as shown in Figure 7). From this pane we can graphically change the radio settings such as the PAN ID, destination address, and even the firmware version that the radio is running.

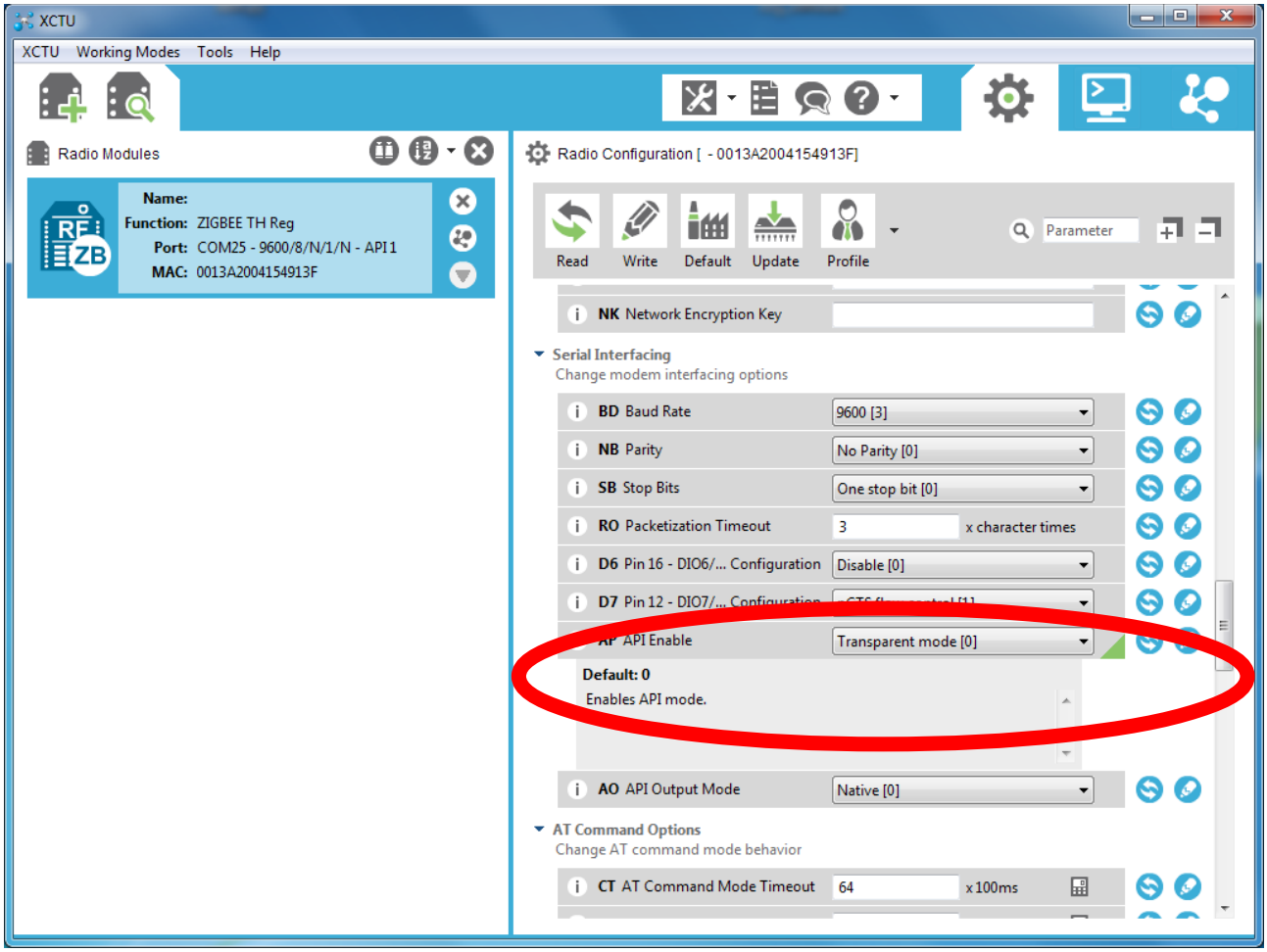


Figure 7 – XCTU radio configuration pane

The XBee radio modules can work in two modes AT (transparent) and API mode. In AT mode, data sent to an XBee module is forwarded on to the destination address specified in the modules memory. This mode is particularly useful in simple networks and for point-to-point communications where the destination addresses don't change very often (in larger networks, or networks where the destination addresses change more frequently, API mode allows much more flexibility).

When using the XBee S2C radio modules there is not separate firmware for API and AT modes (unlike the standard S2 modules), instead we have to set the API enable mode to “transparent” (see highlight option in Figure 7).

When the module is in AT mode, there is a command mode that can be triggered to alter it's configuration (e.g. to change the destination address). To enter this command mode a sequence of three plus signs, '+++' is sent over a terminal connection and is acknowledged by an 'OK' message. A full list of AT commands can be found in the XBee / XBee PRO product manual[4].

4 Available from http://ftp1.digi.com/support/documentation/90000976_V.pdf and on blackboard

Figure 8 shows an example of using this AT command mode to read the high and low addresses of the current XBee module using TeraTerm. Note that the command strings are all preceded by 'AT'.

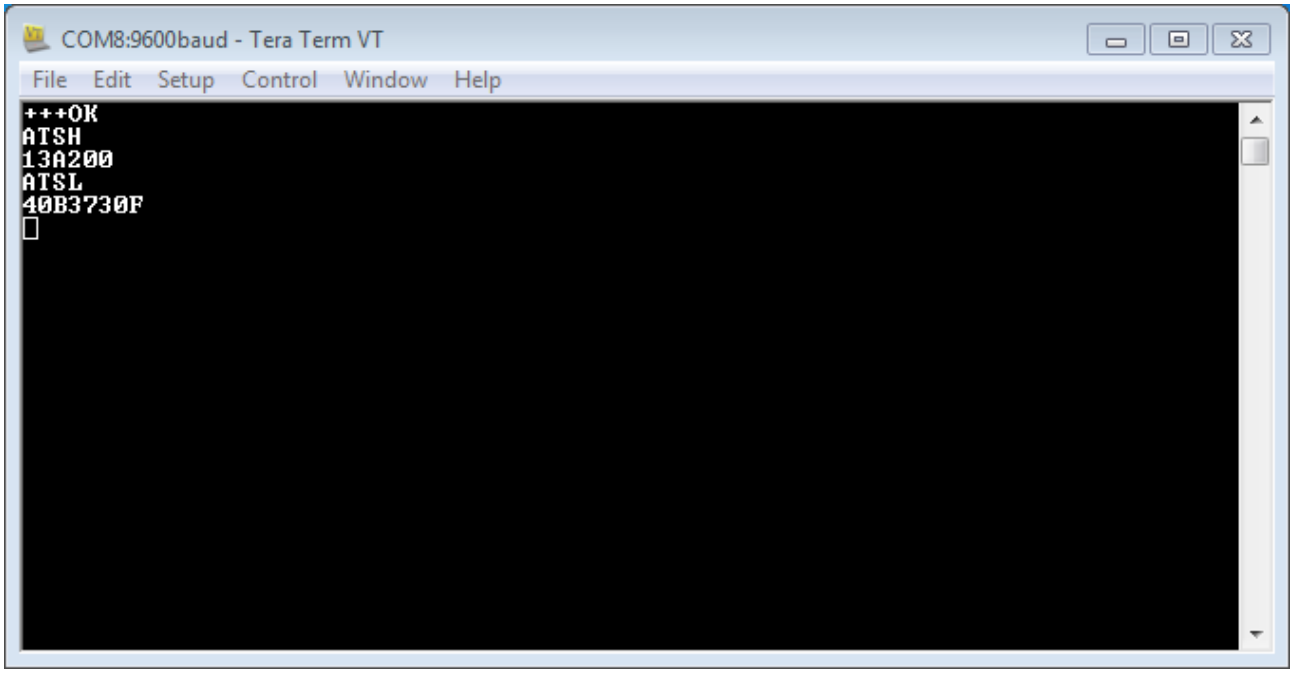


Figure 8 – AT command mode

Now use the information in the AT command table to read the following information:

1. The firmware version that the radio is running
2. The operating PAN ID
3. The current supply voltage to the radio

Task 2 – A simple chat connection

This task shows you how to get basic wireless chat functionality going between two computers (or two terminal windows on the same PC) using the Digi XBee ZB (series 2C) wireless modules. Using a serial connection from one computer, the text you type can be wirelessly sent to another computer (and vice versa). The basic topology is shown in Figure 9.

Figure 9 – XBee wireless chat topology

You will need:

- 1 x XBee radio configured as a **Zigbee Coordinator AT**
- 1 x XBee radio configured as a **Zigbee Router AT**
- 2 x XBee adapter boards (we are using the Parallax adapter boards)
- 2 x serial terminal instances (either on the same computer or on two different computers)

All XBee radio modules have a 64bit serial number address printed on the back (see Figure 10). The high part of this will be the same for every XBee (0013A200, which corresponds to the manufacturers allocated address space), and the low part is the unique identifier for every radio.

Figure 10 – An XBee radio showing the 64bit address

For the radio in Figure 10, the unique address is 403B9E21. **Write down these numbers from the 2 XBee modules now as you will need them later!**

Coordinator:

Router:

Now configure the coordinator and router XBees. The key parameters are:

Radio 1	**Radio 2**
ZigBee TH Reg	ZigBee TH Router AT enabled
Coordinator Enable = Enabled [1]	Coordinator Enable = Disabled [0]
PAN ID = 2001	PAN ID = 2001
DH = 0013A200	DH = 0013A200
DL = <the router address you wrote down earlier>	DL = <the coordinator address you wrote down earlier>
API Enable = Transparent mode [0]	API Enable = Transparent mode [0]

Figure 11 shows how to set the PAN ID and "Coordinator Enable" option for the coordinator using XCTU[5]. You will also need to make sure that the destination address is set correctly and the API output mode is set to "Transparent mode [0]". Make sure to write these parameters to the radio memory when you are done and then use the masking tape / sticky labels provided to label the radio as coordinator (this will make life easier later!).

5 You can also set the same parameters using AT command mode in a terminal – however, XCTU is my preferred way as it means that you can easily check the correct firmware is running on the radio and change it if necessary.

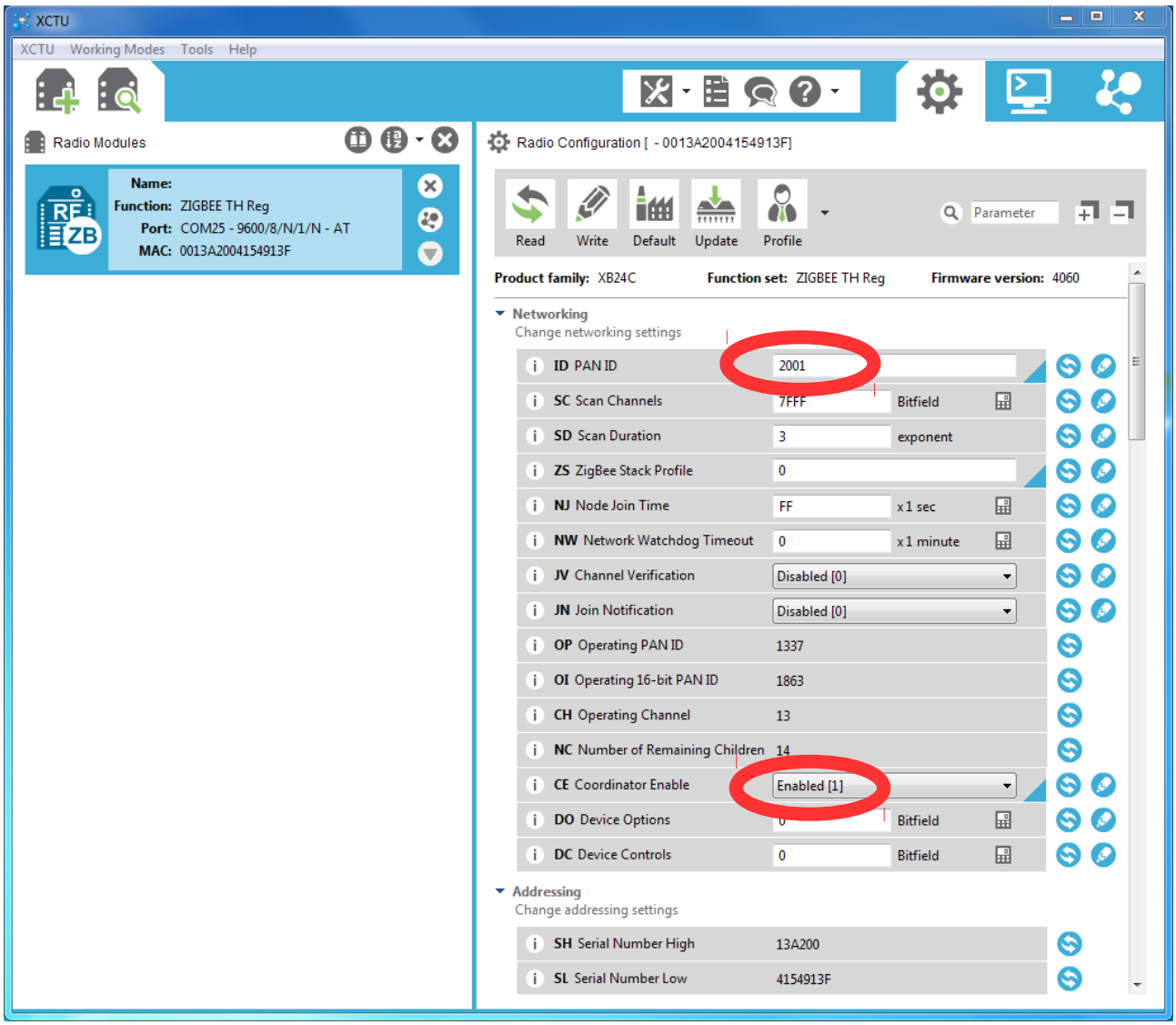


Figure 11 – Setting the PAN ID and “Coordinator Enable” option using XCTU

Now configure the router XBee (see Figure 12). Remember to set the appropriate options (as in the table above). Again, once this is done, write these setting to the firmware and use the masking tape / sticky labels to label it as the router.

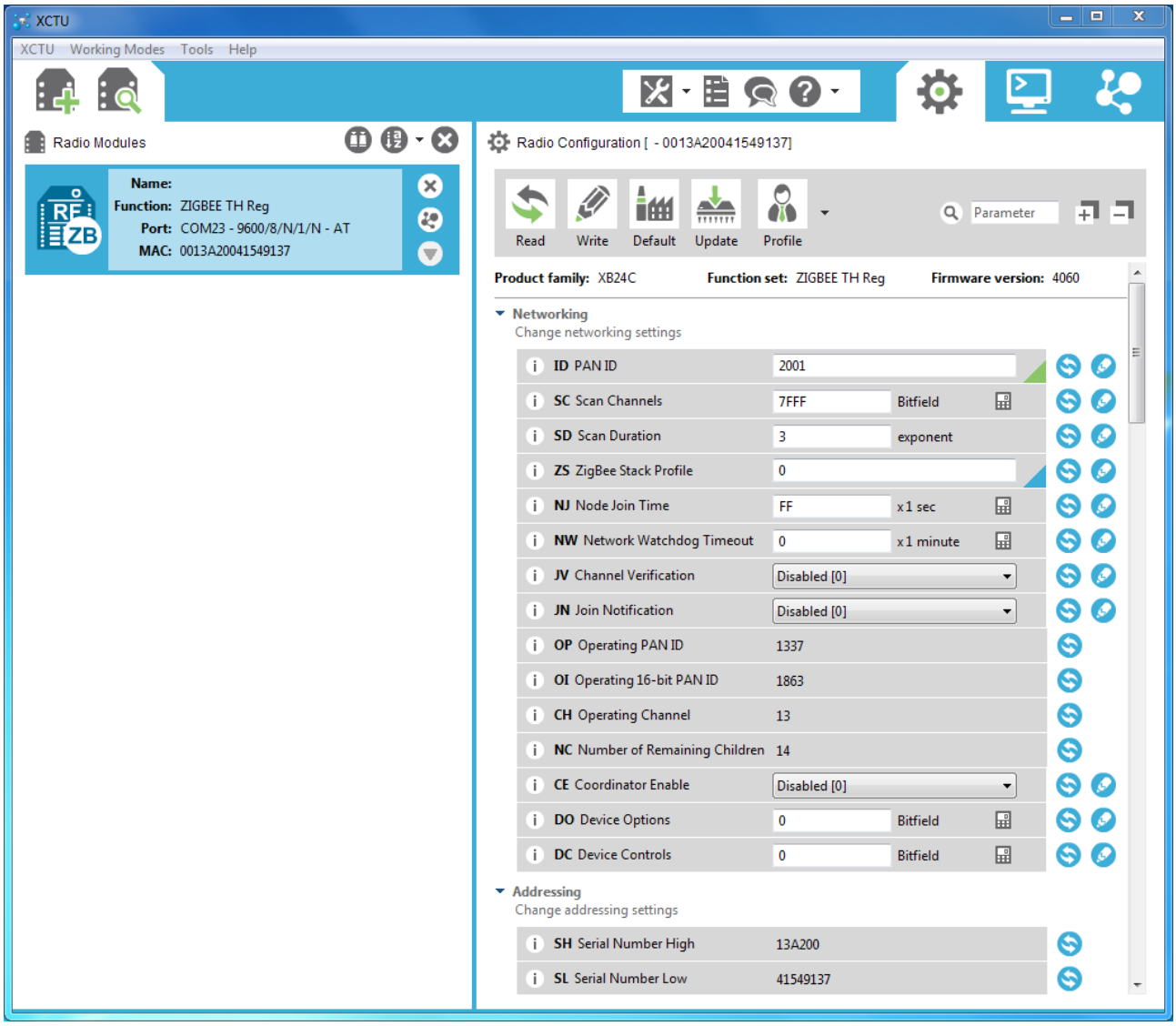


Figure 12 – Configuring the router

Now connect both XBees (either to the same PC or to different PCs) and open up your favourite terminal emulator. You should be able to send text from one terminal to another wirelessly! Figure 13 shows this simple chat connection working.

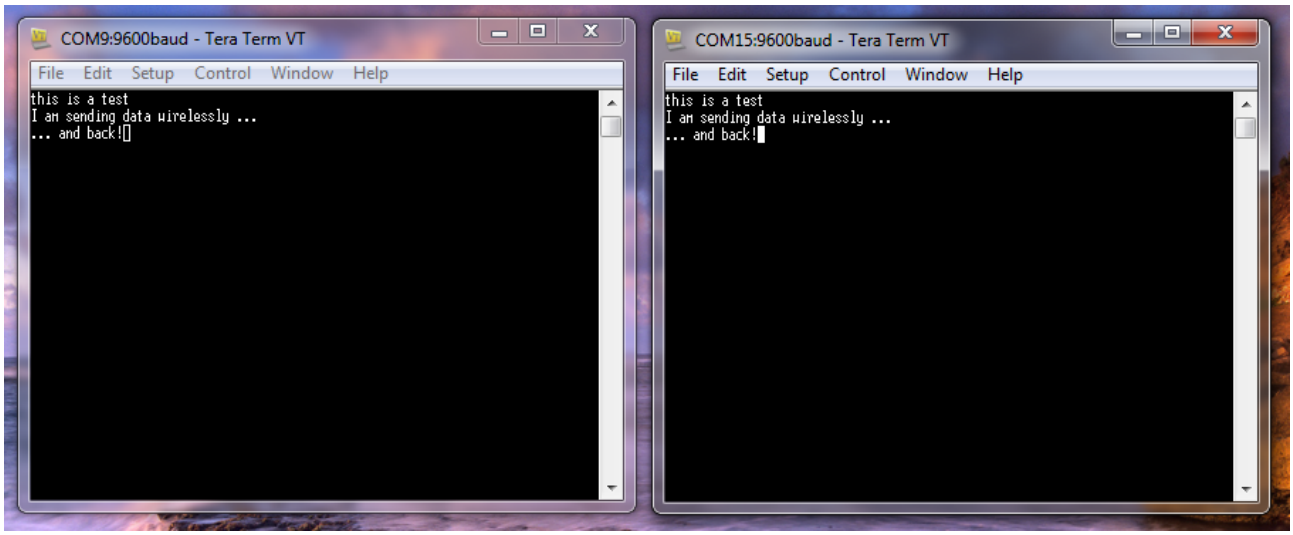


Figure 13 – Simple XBee ZB chat

If it doesn't work first time, don't give up – setting up these XBee modules can be complex and messing up one of the parameter settings often results in the connections failing! If this happens, there are several troubleshooting steps you can try:

- Check the radio is in the adapter board properly
- Check you can connect to each radio (try putting it in command mode using '+++') - if you can't connect to the radio, then try altering the port selection / baud settings / etc.
- If both radios are responding try checking:
  - the PAN ID
  - the destination addresses
- If all else fails, you can use XCTU (or even the AT command mode) to reset the radios back to the factory defaults and try again

Once you have got everything working there are is one other thing to try – if you set the low part of the destination address on the broadcasting XBee to 'FFFF' you can broadcast to every other XBee radio using the same PAN ID.

Task 3 – XBee light switches

In this task we will learn about the XBee API mode by sending API packets to control LEDs and read from analog sensors using the remote AT command capability of the series 2 XBee modules. The general architecture of our system is shown in Figure 14.

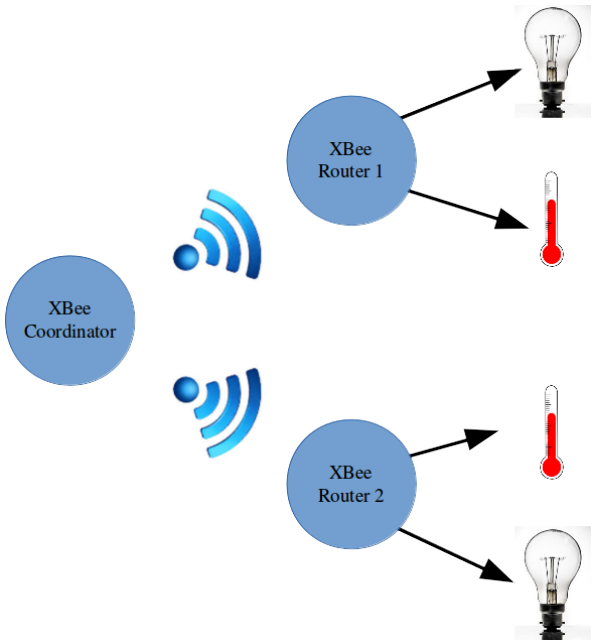


Figure 14 – The wireless sensor and actuator network

The XBee radio modules are capable of driving digital outputs on some of the pins and reading analog inputs on some of the other pins (via the in-built ADC). However, **a word of caution** when reading analog inputs – the maximum capability of the XBee ADC is 1.2V **and if you supply a voltage higher than this you will break it!**

Figure 15 shows the breadboard view of our system (note, you could also build the two XBee sensor nodes on separate breadboards if you wish). Break-out boards (see Figure 16) are used for the XBee modules to ensure that they are breadboard compatible (as you can see, the XBee radio module pin spacing is too small to fit into a breadboard). Also note that these break-out boards have on board voltage regulators to ensure that the XBees receive the correct supply voltage (3.3V).

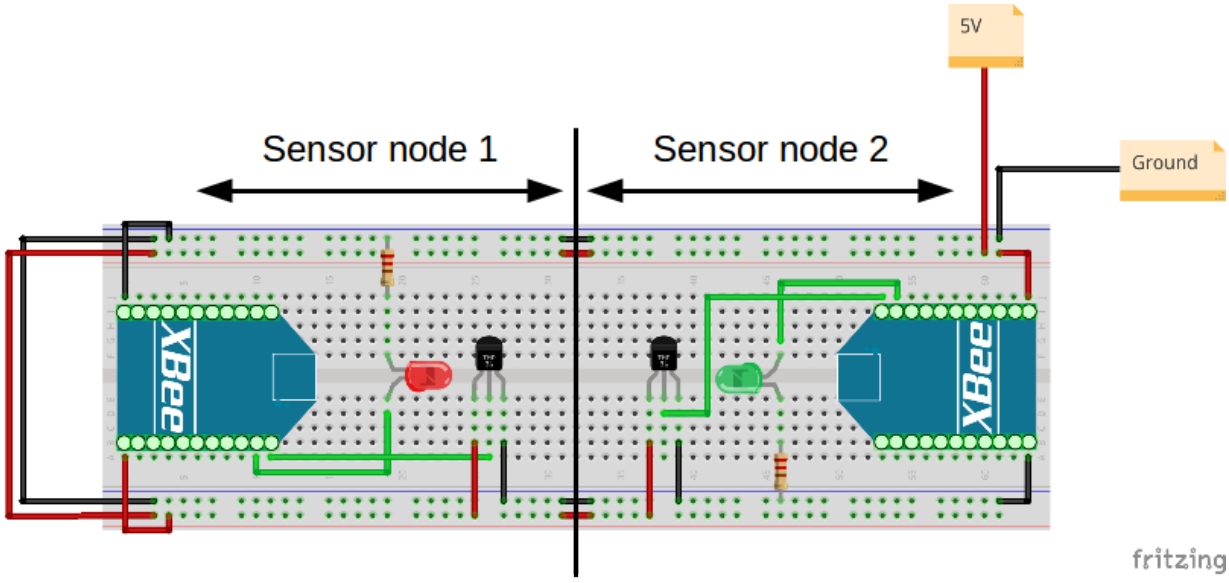


Figure 15 – Wireless sensor and actuator circuit using XBees

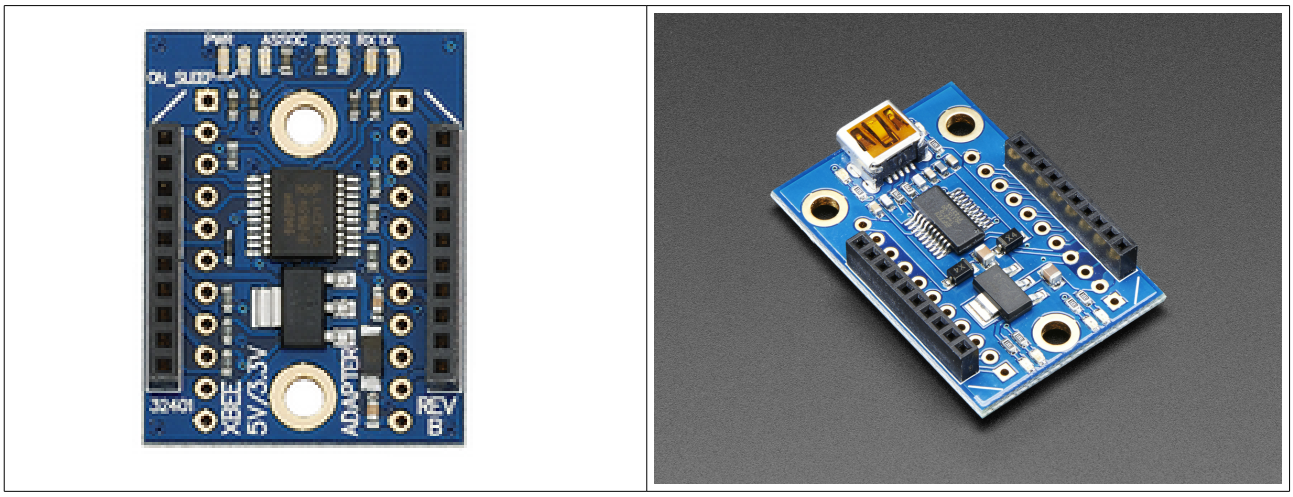


Figure 16 – XBee adapter boards

Now we need to configure the XBee routers to use API mode (see Figure 17).

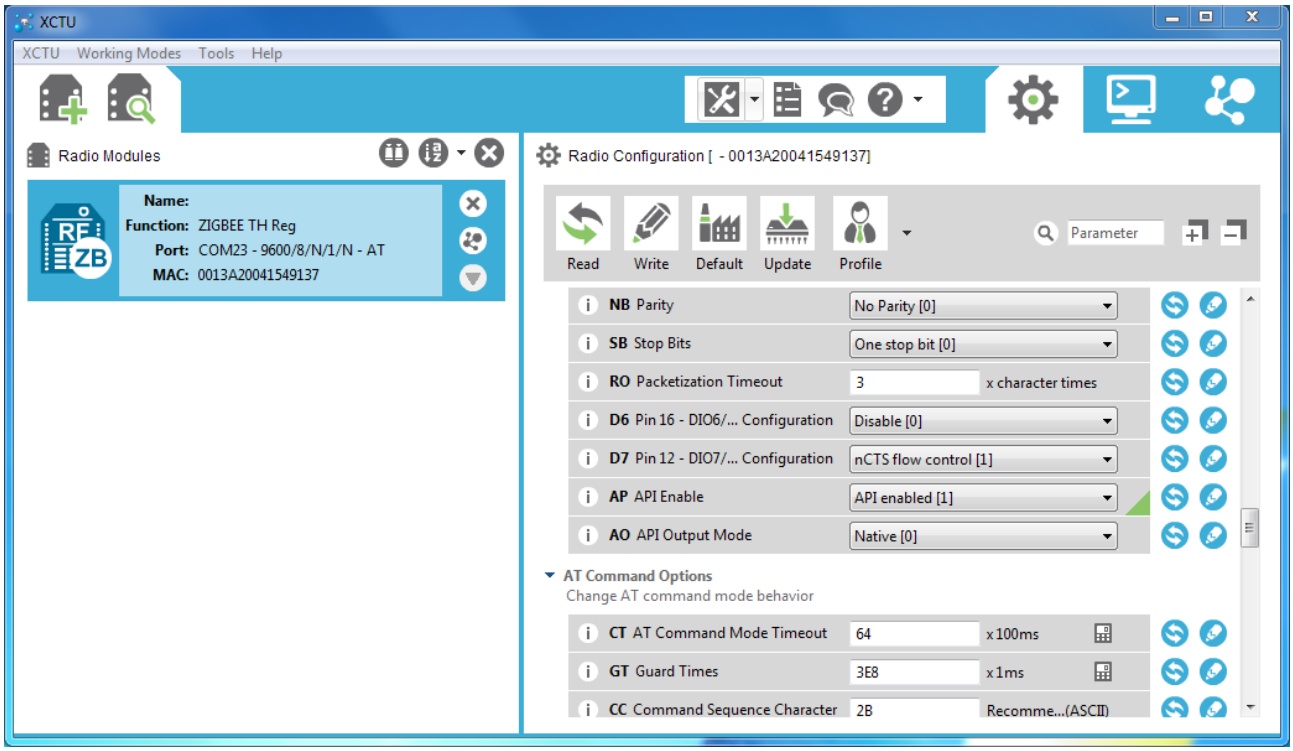


Figure 17 – Setting API mode enabled

Once you have set API mode to “API enabled”, you should set a unique PAN ID to avoid conflicts with the other students around you. Don't forget to write this to the radios memory.

Do this for the other router XBee and then put both XBee modules in the break-out boards in the circuit (making sure that the radios are the correct way round). The final step in setting up our network is to set the “Coordinator Enabled” mode on the last XBee to “Enabled [1]” and make sure its PAN ID is set to the same as the two routers.

As mentioned previously, API mode on the XBee radio modules works by sending packets of data to the radios within the network. A complete description of the format of these packets and the various frame types can be found in the XBee product guide (available on blackboard), but, to make our life easier, XCTU includes a built in frame generator that simplifies the process of creating API frames to control remote XBees (see Figure 18).

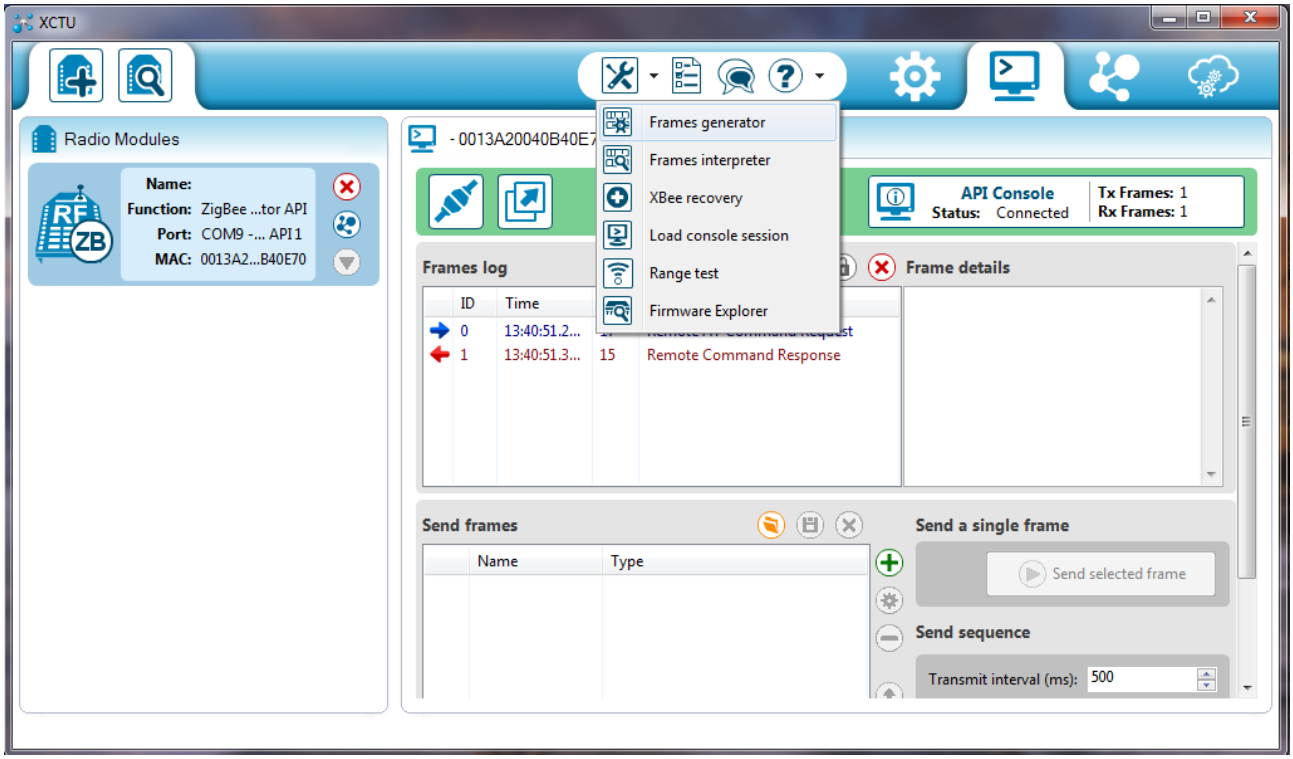


Figure 18 – The XBee frame generator (in the tools menu)

The frame generator allows us to specify the type of frame and the destination address to send the frame to. In the case of the remote AT command frame type used in this task it also allows us to specify the AT command to send (in either hex or ASCII) and the parameters to send with the AT command. The frame generator then calculates the additional parts of the frame such as frame length and checksum. Figure 19 shows the frame generator window creating a frame to turn on the remote LEDs (attached to pin 19 / DIO1 on both radios) in our circuit.

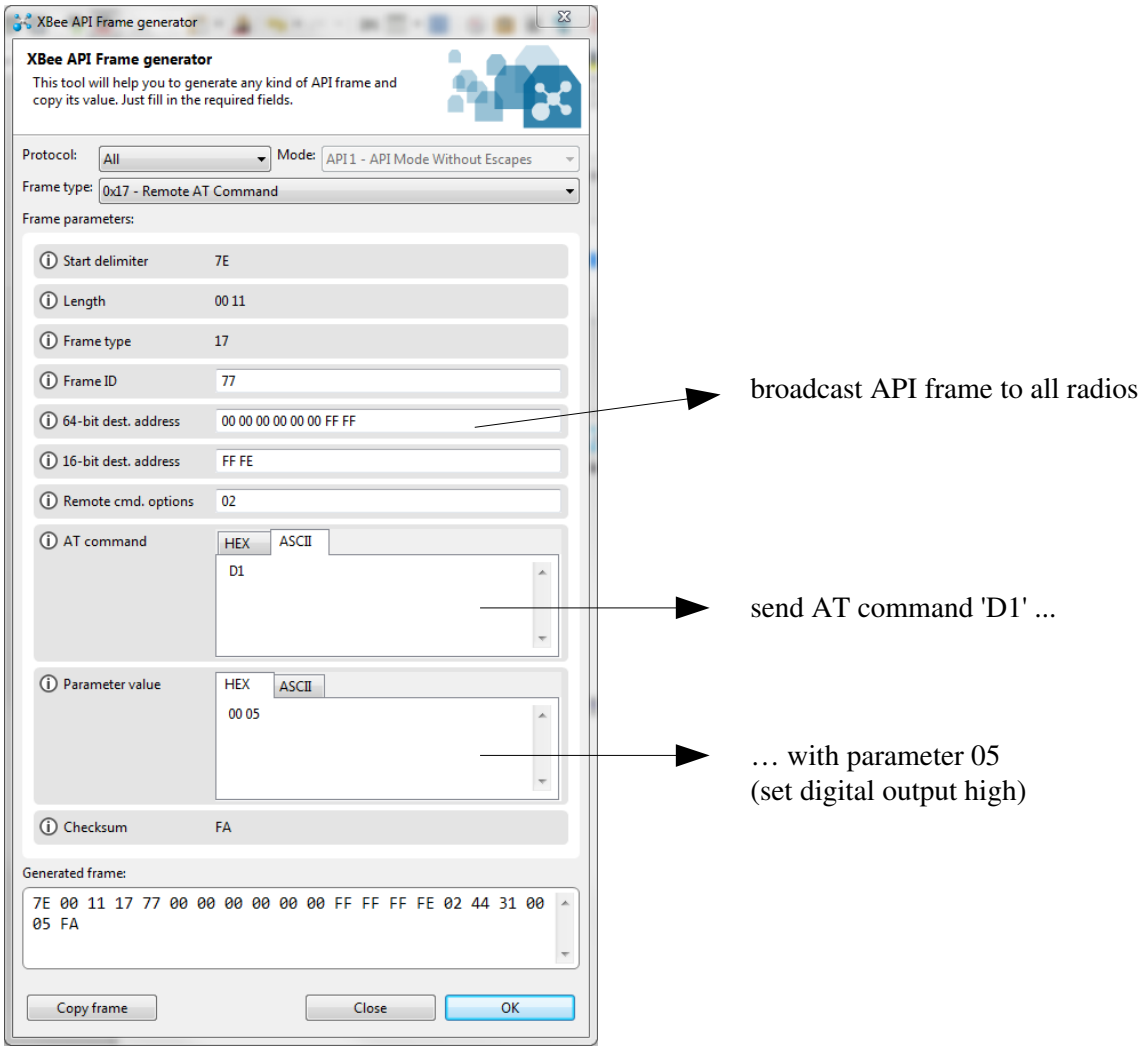


Figure 19 – The XCTU XBee API frame generator[6]

6 Note that we can send the same type of frame but with the parameter value set to 04 (digital output low) to turn the LEDs off

When we send this API packet, we should get a remote command response frame from both the remote radios (see Figure 20) and both the LEDs in the circuit should turn on (see Figure 21).

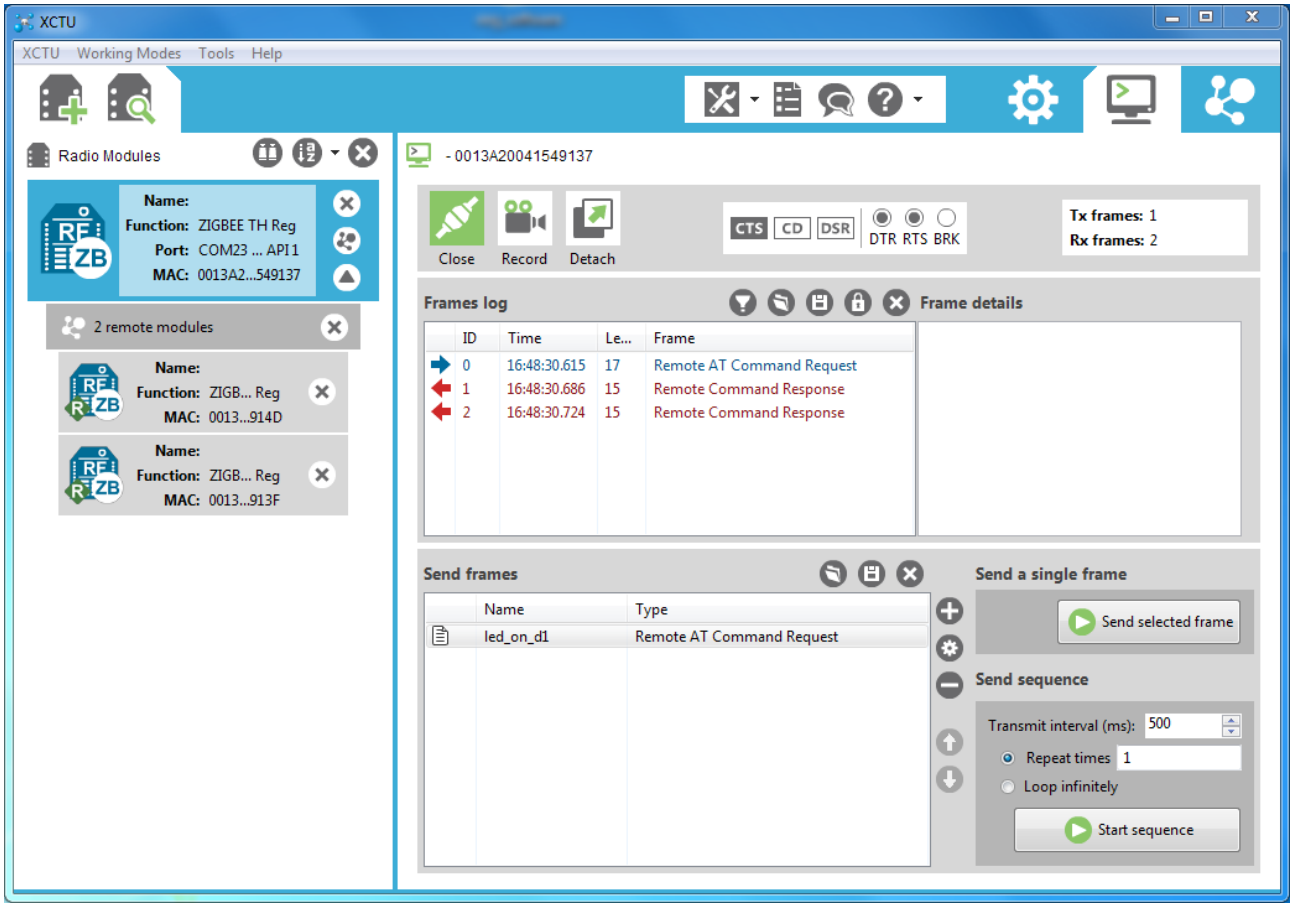


Figure 20 – Remote response frame to acknowledge the sent command

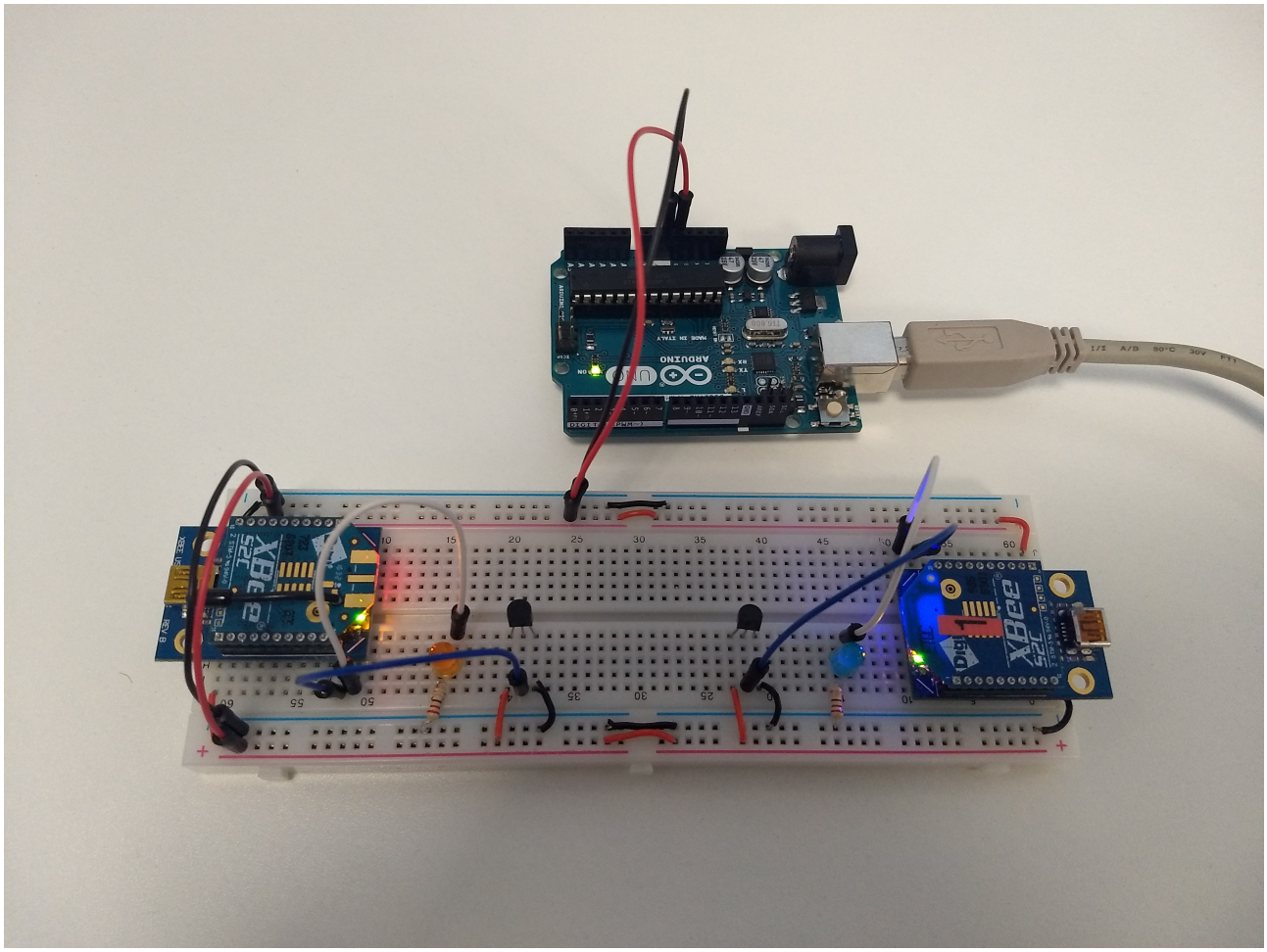


Figure 21 – Wireless control of LEDs (note that the Arduino in this picture is just acting as a power supply)

Instead of turning both LEDs on at once by sending a broadcast message to every radio on the network, we can direct our packet to a specific XBee radio module by setting the 64 bit destination address in the frame generator. In this way we can turn a specific LED on and off. Try doing this by using the XCTU frame creator to modify the frames created above.

We now want to read the values of the analog temperature sensors attached to our XBee radio modules. To do this involves sending two API frames to each radio: one to enable analog input on the pin the sensor is attached to, and one to read the sensor. To enable analog input on pin DIO0 on all radios in the network we can send the API frame:

```
7E 00 10 17 77 00 00 00 00 00 00 FF FF FF FE 02 44 30 02 FE
```

This sends the remote AT command frame to all radios on the network with the command 'D0 02' (see Figure 22 for the appropriate snippet of the frame generator) meaning “enable analog input on pin D0”. Figure 23 shows us the relevant extract from the AT command table in the XBee product manual (available on BlackBoard).

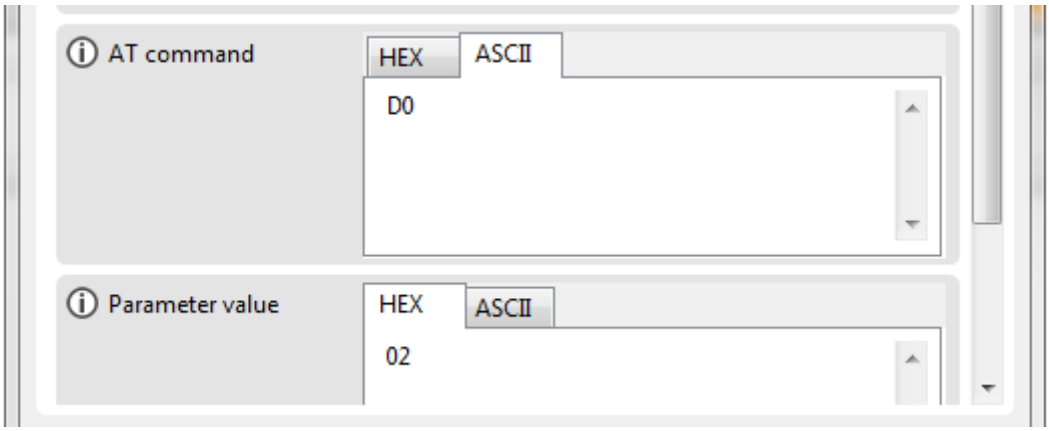


Figure 22 – XCTU API frame generator command

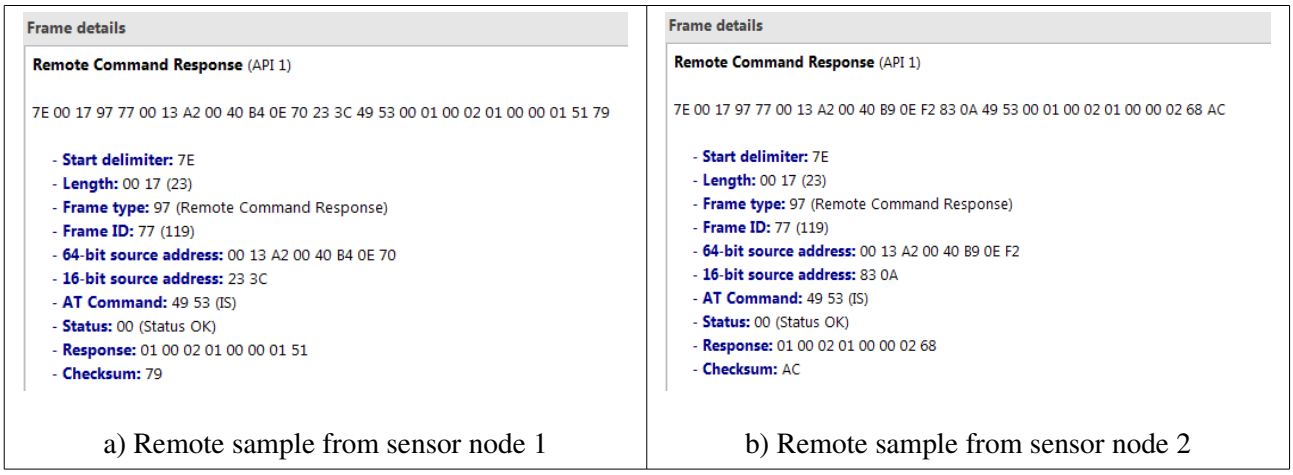
D0	**AD0/DIO0 Configuration**. Select/Read function for AD0/DIO0.	CRE	1 - Commissioning button enabled 2 - Analog input, single ended 3 - Digital input 4 - Digital output, low 5 - Digital output, high	1

Figure 23 – XBee product manual snippet for AT commands

Now we have set up the analog inputs on both the radios in our network, we have to send another API frame asking the XBee modules to read those inputs. This is know as ***queried sampling***. The API frame to read from the enabled inputs is:

```
7E 00 0F 17 55 00 00 00 00 00 00 FF FF FF FE 02 49 53 FA
```

which sends the remote AT command 'IS' to all the radios in the network. We then get remote command responses from the radios containing the analog data they have read. See Figure 24 for the remote responses in our circuit.



a) Remote sample from sensor node 1

b) Remote sample from sensor node 2

Figure 24 – Remote command responses from the XBee radio modules

We then have to parse the response from the API packet to extract the ADC reading. The response data from sensor node 1 is:

```
01 00 02 01 00 00 01 51
```

Table 2 shows the interpretation of this data:

Number of samples	**01**	Indicates how many samples are included per frame. This will always be 01.
Digital channel mask	**0002**	This is a bit mask indicating which channels are set as digital inputs: **0000000000000010** means that DIO1 is enabled.
Analog channel mask	**01**	This is a bit mask indicating which channels are set as analog inputs: **00000001** means that DIO0 is enabled as an analog input.
Digital samples	**0000**	If the digital channel mask is not 0x0000, then this bit field will have the digital sample data.
Analog samples	**0151**	If the analog channel mask is not 0x0000, then this is the value of the ADC[7].

Table 2 – Interpreting the response data in the API frame

We can see from this that the ADC value that we receive from the remote radio is *0x0151*. First we have to convert this hexadecimal value into a decimal (*0x0151* = 337) and then use the equation below to calculate the voltage:

$$\frac{ADC}{1023} \times V_{ref} = Voltage$$

$$\frac{337}{1023} \times 1.2\,V = 0.395\,V$$

7 Each analog sample will have a separate 2-byte field representing the 10 bit ADC values.

Now do this for the other sensor node!

After we calculate the voltage value from the temperature sensors, we can convert that into temperature by using the following equation:

$$\frac{mV_{out} - 500}{10} = Temp\,°C$$

Therefore, the recorded temperatures are:

Note that, as in the case of the LEDs, we can send a remote sample query to a specific XBee by modifying the API frame to include the appropriate destination address.

As well as this queried sampling mode, we can configure the radios to periodically sample the sensors every so often. We do this by setting the IO Sampling Rate (IR) in XCTU (as shown in Figure 25). This sampling rate is in hexadecimal, but XCTU has handily provided a calculator to do the conversion!

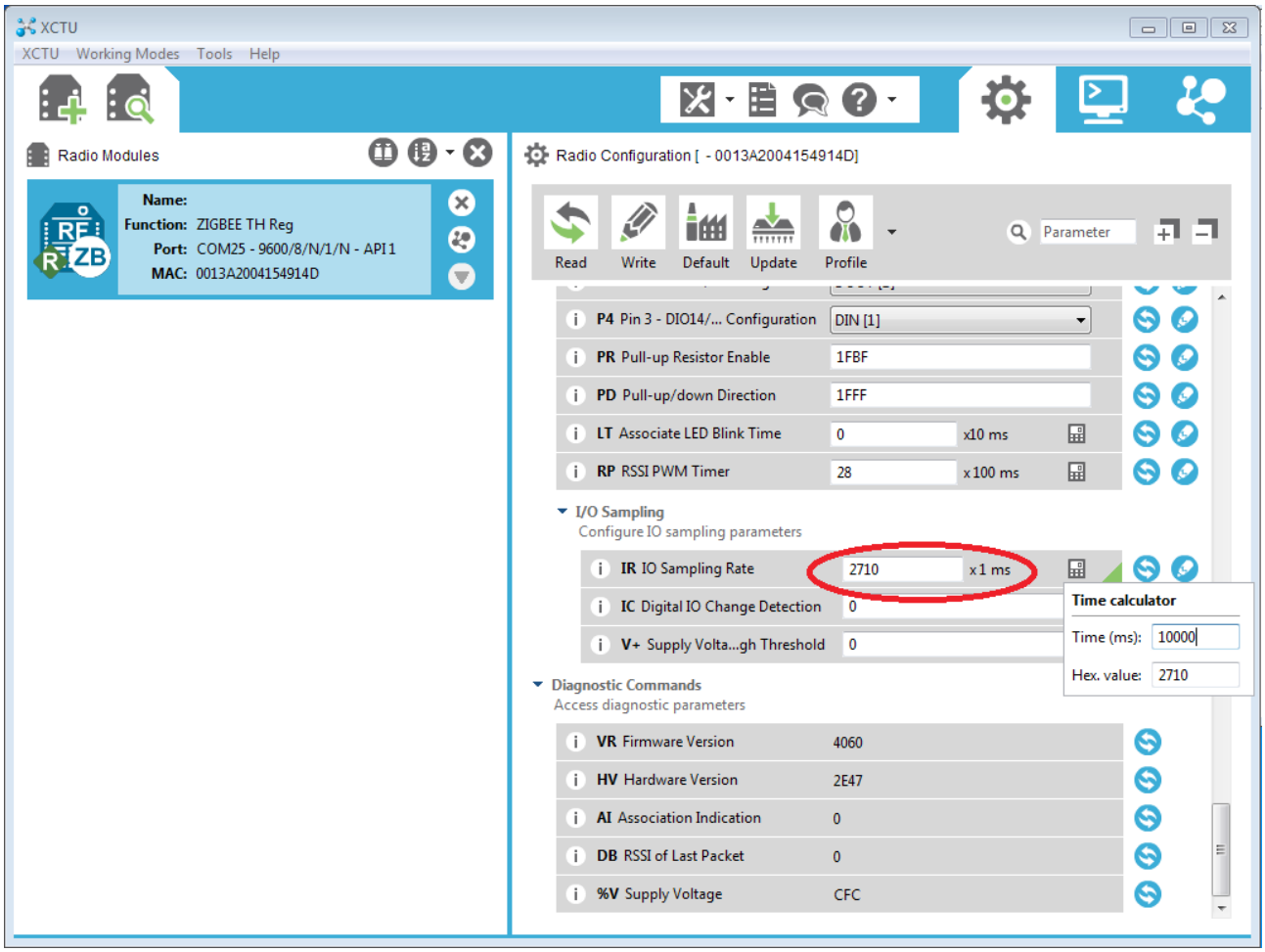


Figure 25 – Using XCTU to configure periodic sampling

This gives us a slightly different packet structure (see Figure 26), but the data payload format is the same as for queried sampling (see highlighted part in Figure 26 and Table 2 for a full breakdown of this).

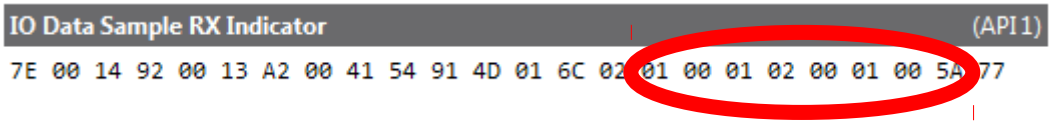


Figure 26 – IO Data Sample RX packet

Task 4 – Integrating XBee radio modules and the STM32F7 discovery board

In this task we are going to build on the understanding of XBee API packets developed in the previous exercise to process periodic data samples returned by XBee sensor units using our real-time operating system. Our system architecture (in terms of threads) is shown in Figure 27.

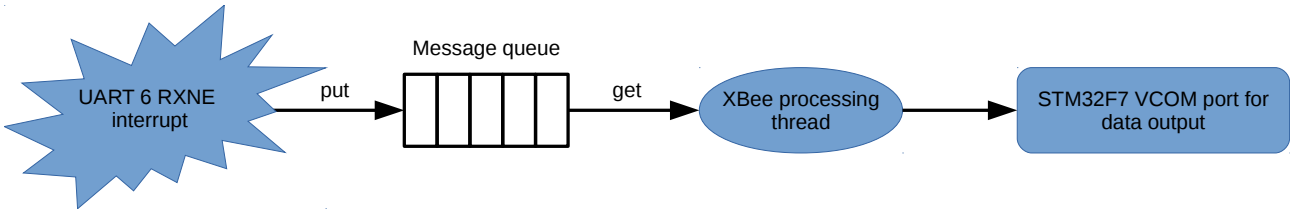


Figure 27 – System architecture for RTOS based queried sampling with XBees

The data from the XBee radios is received via a UART RXNE interrupt (i.e. data received triggers an interrupt) and stuffed into a message queue character by character. The XBee processing thread then pulls these characters out of the message queue and parses them using a state machine based packet parser (see code listing 3 later).

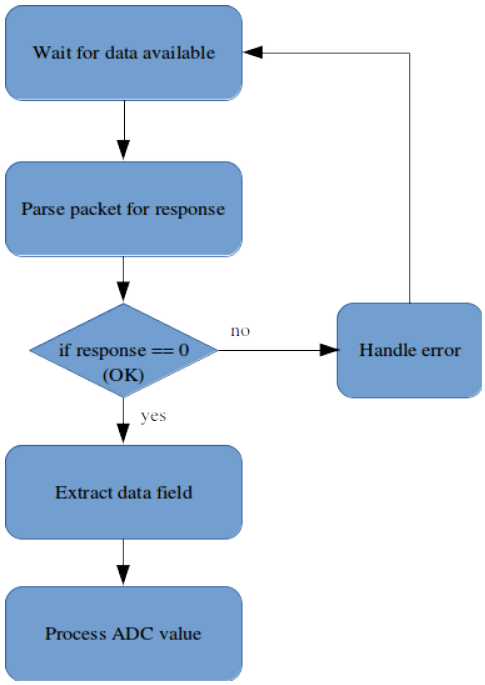


Figure 28 – Flowchart for the XBee data processing thread execution

We are going to build the circuit shown in Figure 29 to connect the XBee coordinator to our STM32F7 discovery board. Figure 30 replicates the schematic for the Arduino header on the STM32F7 discovery board – make sure you note which pins are which for UART6! The SHU break out board has all the pins labelled.

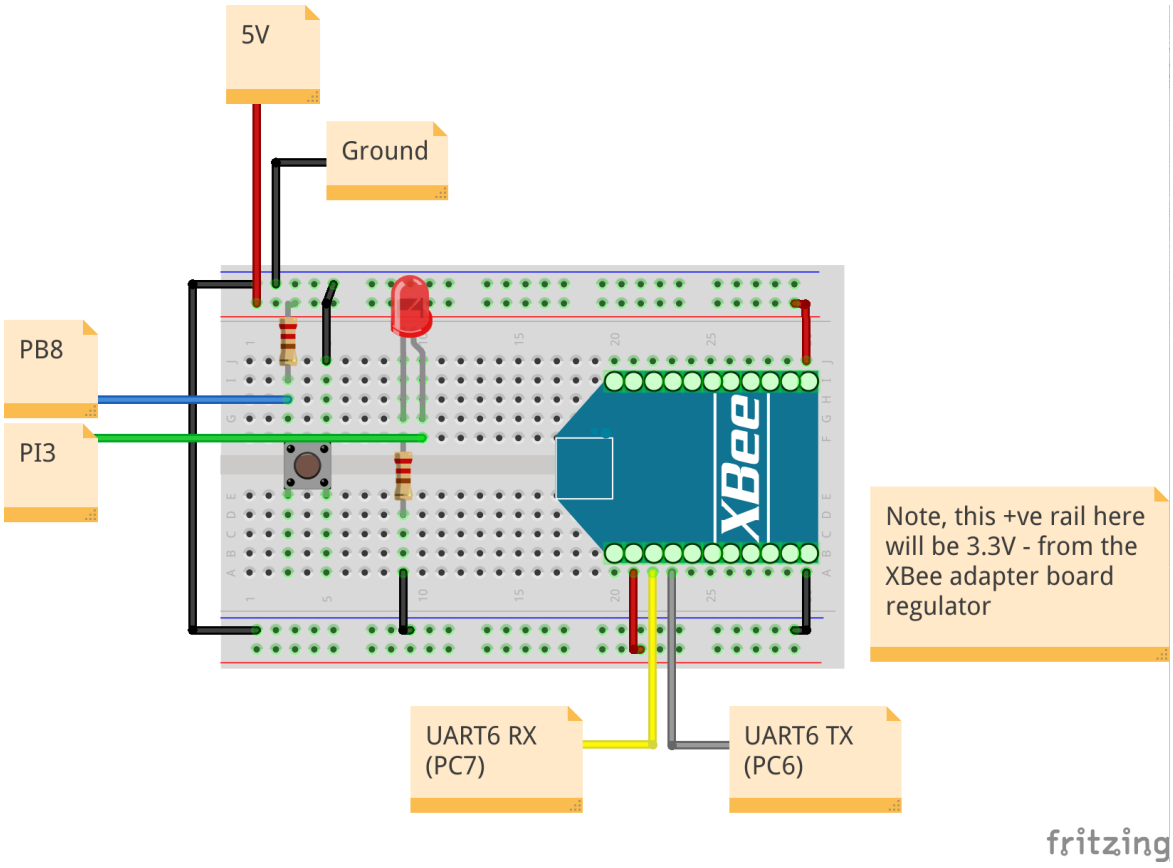


Figure 29 – The XBee coordinator to STM32F7 discovery connections

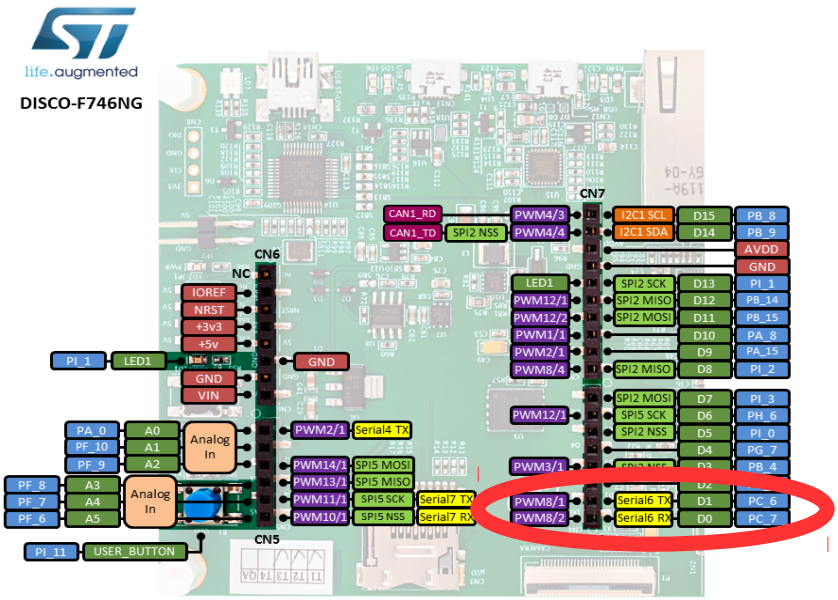


Figure 30 – Arduino header

We will be using environmental sensor boards (as shown in Figure 31) that we have had custom built here at SHU. These contain a TMP36 temperature sensor and an LDR sensor to allow the measurement of temperature and light levels (and include protective circuitry to hopefully isolate the XBee ADC pins from going above their 1.2V maximum).

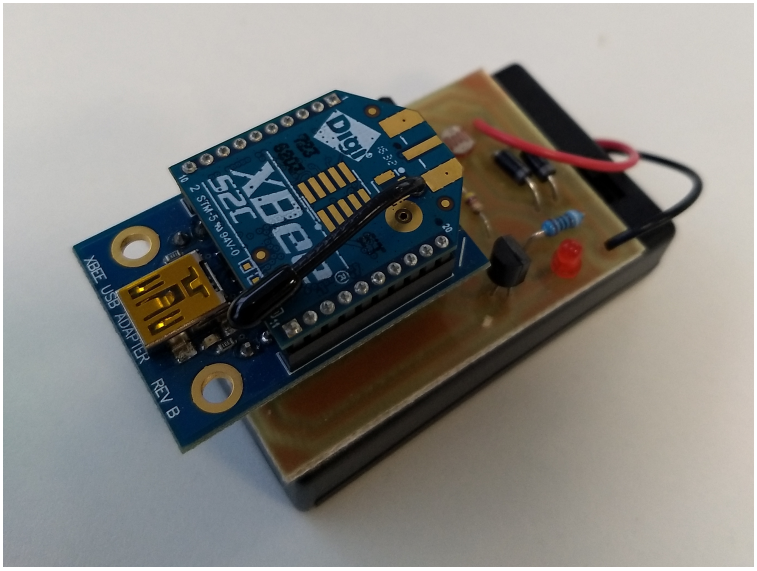


Figure 31 - XBee Sensor Board

You should make sure to configure the XBee radio modules to have the same PAN ID and use the appropriate firmware and API mode (as explained in the previous task). These are shown in Table 3, below.

XBee connected to the STM32F7 discovery	XB-24 ZB Coordinator API
XBee Sensor Node 1	XB-24 ZB Router API
XBee Sensor Node 2	XB-24 ZB Router API

Table 3 – XBee module configuration

The **`1_rtos_xbee`** project structure looks something like that shown in Figure 32, below. It is loosely based around task 4 from the RTOS lab (lab 103), in which we used an interrupt attached to a UART to handle the reception of data and then a separate thread to process that data.

The project follows a similar structure to the ones we used in the rest of our RTOS based applications. You can see that the structure is basically split into:

- application
- configuration
- RTOS
- libraries

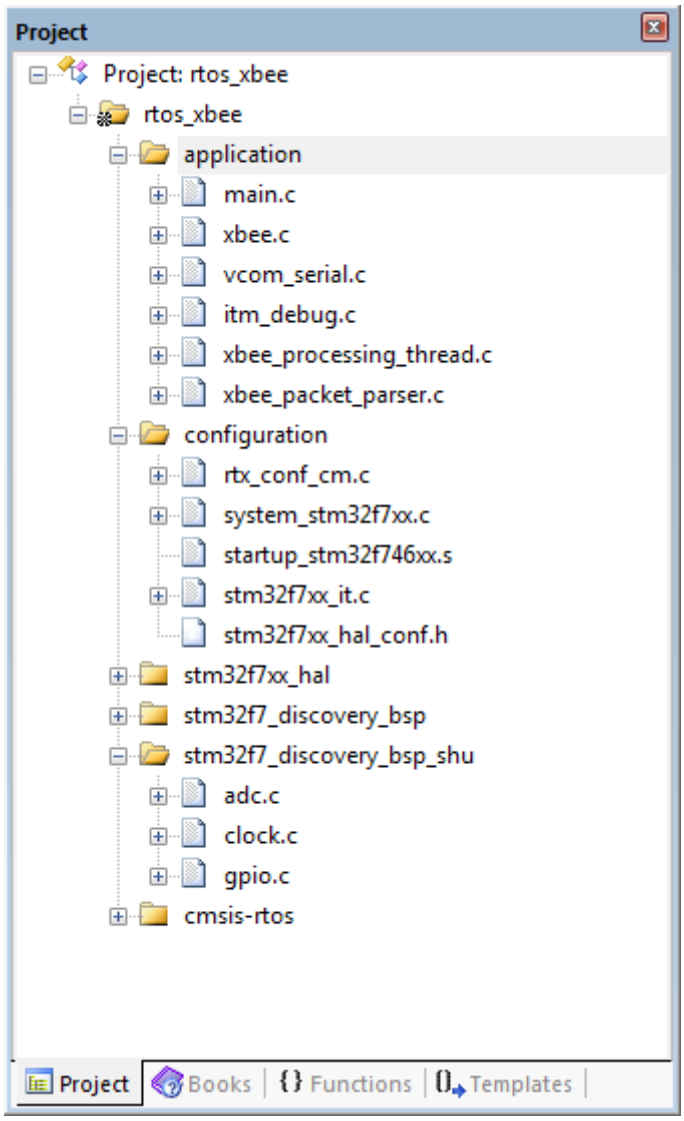


Figure 32 – The project structure

The key files within this project are:

main.c	This is where we initialise everything and create our threads. It mostly follows the same structure as the rest of our RTOS based projects (see lab 103).
vcom_serial.c	This is some configuration and implementation code for transmitting data over the stm32f7 discovery board's virtual com port. This is configured so that we can use 'printf' with the virtual com port for the display of our data.
itm_debug.c	This provides some simple debugging functionality to the itm viewer in the uvision debugger. We use this so that it doesn't interfere with the vcom output. We will use the virtual com port for program output and the itm viewer for debugging information.
xbee.c	This provides configuration and implementation code for the uart that the xbee coordinator is going to use (uart 6). This also handles initialising the uart interrupts and stuffing received characters into a message queue (which is how we are going to get data from the XBee modules).
xbee_processing_thread.c	This is the initialisation and processing code for receiving data from the XBee coordinator that is attached to the STM32F7 discovery board.
xbee_packet_parser.c	This file contains the functions that do the majority of the work in receiving data from the XBee radio over the UART, parsing the data into XBee packets, and validating those packets to ensure they have not been corrupted.

Table 4 – Key project files

The **main.c** program in code listing 1 shows a simple program that initialises the XBee sensor nodes in our WPAN to enable us to read from the on-board ADCs and then configures periodic samples to be sent from those sensor nodes every 10 seconds.

Remember from earlier that, to configure the XBee radio modules to sample the analog input attached to D0, we need to send the remote AT command 'D0 02' which sets up the XBee radio module analog input attached to that pin. See the XBee product manual for more details.

Note that we are sending these commands as broadcasts to all router / end devices in our PAN.

Code listing 1 – **main.c**

```
/*
 * main.c
 *
 * this is the main rtos xbee / uart based application
 *
 * author:    Alex Shenfield
 * date:      08/11/2018
 * purpose:   55-604481 embedded computer networks : lab 104
 */

// include the basic headers for the hal drivers and the rtos library
#include "stm32f7xx_hal.h"
#include "cmsis_os.h"

// include the shu bsp libraries for the stm32f7 discovery board
#include "pinmappings.h"
#include "clock.h"
#include "gpio.h"

// include the xbee tx and rx functionality
#include "xbee.h"

// include the itm debugging
#include "itm_debug.h"

// RTOS

// declare the extern methods that set everything up for us
extern int init_xbee_threads(void);

// OVERRIDE HAL DELAY

// make HAL_Delay point to osDelay (otherwise any use of HAL_Delay breaks things)
void HAL_Delay(__IO uint32_t Delay)
{
        osDelay(Delay);
}
```

```
// XBEE

// xbee configuration packets

// set up adc on dio 0 on all xbees connected to the WPAN - temperature
uint8_t init_adc_0[] = {0x7E, 0x00, 0x10, 0x17, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0xFF, 0xFF, 0xFF, 0xFE, 0x02, 0x44, 0x30, 0x02, 0x74};

// set up adc on dio 1 on all xbees connected to the WPAN - light
uint8_t init_adc_1[] = {0x7E, 0x00, 0x10, 0x17, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0xFF, 0xFF, 0xFF, 0xFE, 0x02, 0x44, 0x31, 0x02, 0x72};

// configure all radios in the WPAN to do automatic sampling every 10 seconds
uint8_t configure_sampling[] = {0x7E, 0x00, 0x11, 0x17, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
                                0x00, 0xFF, 0xFF, 0xFF, 0xFE, 0x02, 0x49, 0x52, 0x27, 0x10, 0x18};

// CODE

// this is the main method
int main()
{
    // initialise the real time kernel
    osKernelInitialize();

    // we need to initialise the hal library and set up the SystemCoreClock
    // properly
    HAL_Init();
    init_sysclk_216MHz();

    // note also that we need to set the correct core clock in the rtx_conf_cm.c
    // file (OS_CLOCK) which we can do using the configuration wizard

    // set up the xbee uart at 9600 baud and enable the rx interrupts
    init_xbee(9600);
    enable_rx_interrupt();

    // print debugging message
    osDelay(50);
    print_debug("initialising xbee thread", 24);

    // initialise our threads
    init_xbee_threads();

    // wait for the coordinator xbee to settle down, and then send the
    // configuration packets
    print_debug("sending configuration packets", 29);
    osDelay(1000);

    // initialise the adc on pin dio0
    send_xbee(init_adc_0, 20);
    osDelay(1000);

    // initialise the adc on pin dio1
    send_xbee(init_adc_1, 20);
    osDelay(1000);

    // set up the periodic sampling of the xbee pins (every 10s)
    send_xbee(configure_sampling, 21);
    print_debug("... done!", 9);

    // start everything running
    osKernelStart();
}
```

Code listing 2 (below) shows the RTOS based XBee processing thread.

```
/*
 * xbee_processing_thread.c
 *
 * xbee data processing thread which pulls the bytes out of the message queue
 * (put in by the uart 1 irq handler) and uses the xbee packet parser state
 * machine to turn them into complete packets
 *
 * author:    Alex Shenfield
 * date:      08/11/2018
 * purpose:   55-604481 embedded computer networks : lab 104
 */

// include the relevant header files (from the c standard libraries)
#include <stdio.h>
#include <string.h>

// include the rtos api
#include "cmsis_os.h"

// include the serial configuration files
#include "vcom_serial.h"
#include "xbee.h"

// include the xbee packet parser
#include "xbee_packet_parser.h"

// include main.h with the mail type declaration
#include "main.h"

// RTOS DEFINES

// declare the thread function prototypes, thread id, and priority
void xbee_rx_thread(void const *argument);
osThreadId tid_xbee_rx_thread;
osThreadDef(xbee_rx_thread, osPriorityAboveNormal, 1, 0);

// setup a message queue to use for receiving characters from the interrupt
// callback
osMessageQDef(message_q, 128, uint8_t);
osMessageQId msg_q;

// set up the mail queues
osMailQDef(mail_box, 16, mail_t);
osMailQId  mail_box;

// FUNCTION PROTOTYPE

// process packet function (note: you should implement this!)
void process_packet(uint8_t* packet, int length);
```

```
// THREAD INITIALISATION

// create the uart thread(s)
int init_xbee_threads(void)
{
    // print a status message to the vcom port
    init_uart(9600);
    printf("we are alive!\r\n");

    // create the message queue
    msg_q = osMessageCreate(osMessageQ(message_q), NULL);

    // create the mailbox
    mail_box = osMailCreate(osMailQ(mail_box), NULL);

    // create the thread and get its task id
    tid_xbee_rx_thread = osThreadCreate(osThread(xbee_rx_thread), NULL);

    // check if everything worked ...
    if(!tid_xbee_rx_thread)
    {
        printf("thread not created!\r\n");
        return(-1);
    }

    return(0);
}
```

```
// ACTUAL THREADS

// xbee receive thread
void xbee_rx_thread(void const *argument)
{
    // print some status message ...
    printf("xbee rx thread running!\r\n");

    // counter to see how long we live for before crashing :(
    int counter = 0;

    // infinite loop ...
    while(1)
    {
        // wait for there to be something in the message queue
        osEvent evt = osMessageGet(msg_q, osWaitForever);

        // process the message queue ...
        if(evt.status == osEventMessage)
        {
            // get the message and increment the counter
            uint8_t byte = evt.value.v;

            // feed the packet 1 byte at a time to the xbee packet parser
            int len = xbee_parse_packet(byte);

            // if len > 0 then we have a complete packet so dump it to the virtual
            // com port
            if(len > 0)
            {
                printf(">> packet received\r\n");

                // get the packet
                uint8_t packet[len];
                get_packet(packet);

                // display the packet
                int i = 0;
                for(i = 0; i < len; i++)
                {
                    printf("%02X ", packet[i]);
                }
                printf("\r\n");

                // process the packet
                // ???

            // keep track of how long we live for ...
            printf("packet %d processed\r\n", counter++);
            }
        }
    }
}
```

Once we have received the data and added it to a queue (as in task 4 of lab 103), we can then process it in a separate thread. To do this we need to parse the received data into packets (as when it arrives at the STM32F7 discovery board it is just a stream of bytes!). We use a state machine[8] based parser to turn this stream of bytes into a valid XBee API packet – see code listing 3.

Code listing 3 – **`xbee_packet_parser.c`** state machine parser

```
// parse an xbee api packet
int xbee_parse_packet(uint8_t c)
{
  // whilst the xbee buffer isn't full ...
  while(xbee_buffer.num_bytes < RING_SIZE)
  {
    // check if it is an api frame header
    if(state == INIT && c == 0x7e)
    {
      xbee_buffer.data[xbee_buffer.ring_head] = c;
      xbee_buffer.ring_head = (xbee_buffer.ring_head + 1) % RING_SIZE;
      xbee_buffer.num_bytes++;

      state = PACKETLENGTH_HI;
      break;
    }

    // read high byte of data field length
    if(state == PACKETLENGTH_HI)
    {
      xbee_buffer.data[xbee_buffer.ring_head] = c;
      xbee_buffer.ring_head = (xbee_buffer.ring_head + 1) % RING_SIZE;
      xbee_buffer.num_bytes++;

      xbee_remain += c * 10;

      state = PACKETLENGTH_LO;
      break;
    }

    // read low byte of data field length
    if(state == PACKETLENGTH_LO)
    {
      xbee_buffer.data[xbee_buffer.ring_head] = c;
      xbee_buffer.ring_head = (xbee_buffer.ring_head + 1) % RING_SIZE;
      xbee_buffer.num_bytes++;

      xbee_remain += c;

      state = DATAFIELD;
      break;
    }
```

8 http://blog.markshead.com/869/state-machines-computer-science/

```
    // read datafield
    if(state == DATAFIELD && xbee_remain > 0)
    {
      xbee_buffer.data[xbee_buffer.ring_head] = c;
      xbee_buffer.ring_head = (xbee_buffer.ring_head + 1) % RING_SIZE;
      xbee_buffer.num_bytes++;
      xbee_remain--;

      // if we've read all the data field, move on to the checksum
      if(xbee_remain == 0)
      {
        state = CHECKSUM;
        break;
      }
    }

    // read checksum
    if(state == CHECKSUM)
    {
      xbee_buffer.data[xbee_buffer.ring_head] = c;
      xbee_buffer.ring_head = (xbee_buffer.ring_head + 1) % RING_SIZE;
      xbee_buffer.num_bytes++;

      state = COMPLETE;
    }

    // if the xbee packet is done then print it ...
    if(state == COMPLETE)
    {
      // verify packet
      if(!validate_packet())
      {
        // if things have gone wrong, dump the packet to the terminal to help
        // diagnose the problem
        print_debug("\n\rCORRUPTED  ", 11);
        while(xbee_buffer.num_bytes > 0)
        {
          // get char from xbee buffer
          uint8_t c = xbee_buffer.data[xbee_buffer.ring_tail];
          xbee_buffer.ring_tail = (xbee_buffer.ring_tail + 1) % RING_SIZE;
          xbee_buffer.num_bytes--;

          char buf[5];
          sprintf(buf, "%02X ", c);
          print_debug(buf, 3);
        }
        state = INIT;
        return 0;
      }

      // set the state to INIT ready to parse the next packet
      state = INIT;
      return xbee_buffer.num_bytes;
    }
  }
  return 0;
}
```

A complete uVision project for this task is available on GitHub. Build this project and check it works. You should see output in TeraTerm something like that shown in Figure 33.

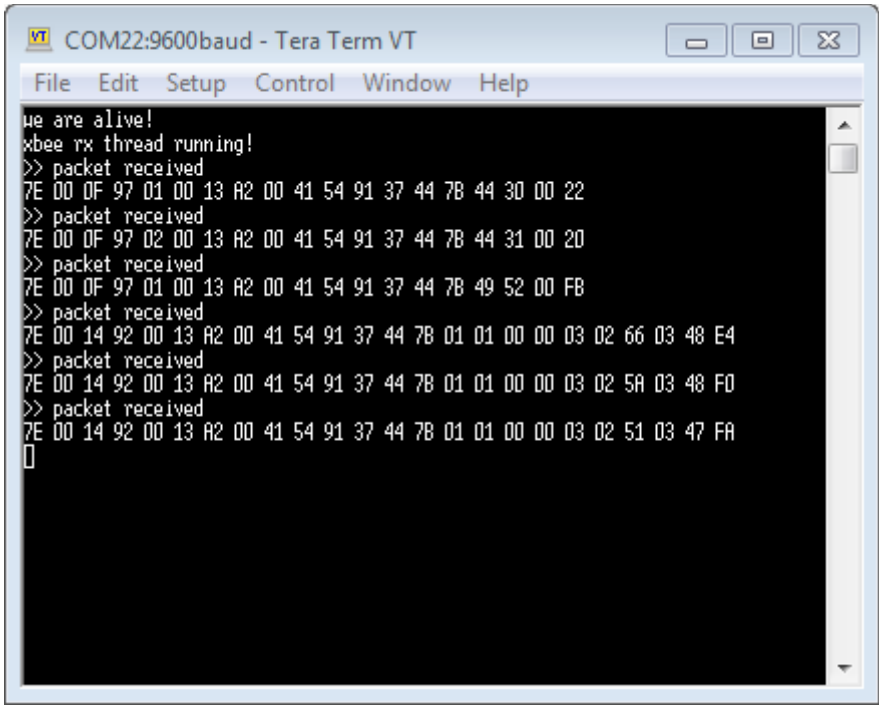


Figure 33 – TeraTerm window for XBee communication application

Now you should make some modifications to the program to add functionality:

1. Write an additional method within the xbee_rx_thread to take the XBee API packet and process it (e.g. extract the ADC readings from the periodic data samples). See Appendix A for a simple example.

2. Send this processed data to another thread (e.g. data_display_thread) using a mailbox and display it from there.

3. Extend this method to process different API packet types (for example, the remote command responses associated with the ADC0 / ADC1 configuration packets).

4. Use the network addresses of the XBee sensor nodes to simulate sensors located in different locations – e.g. radio 0013A200 40B90EE0 is located in the kitchen, etc.

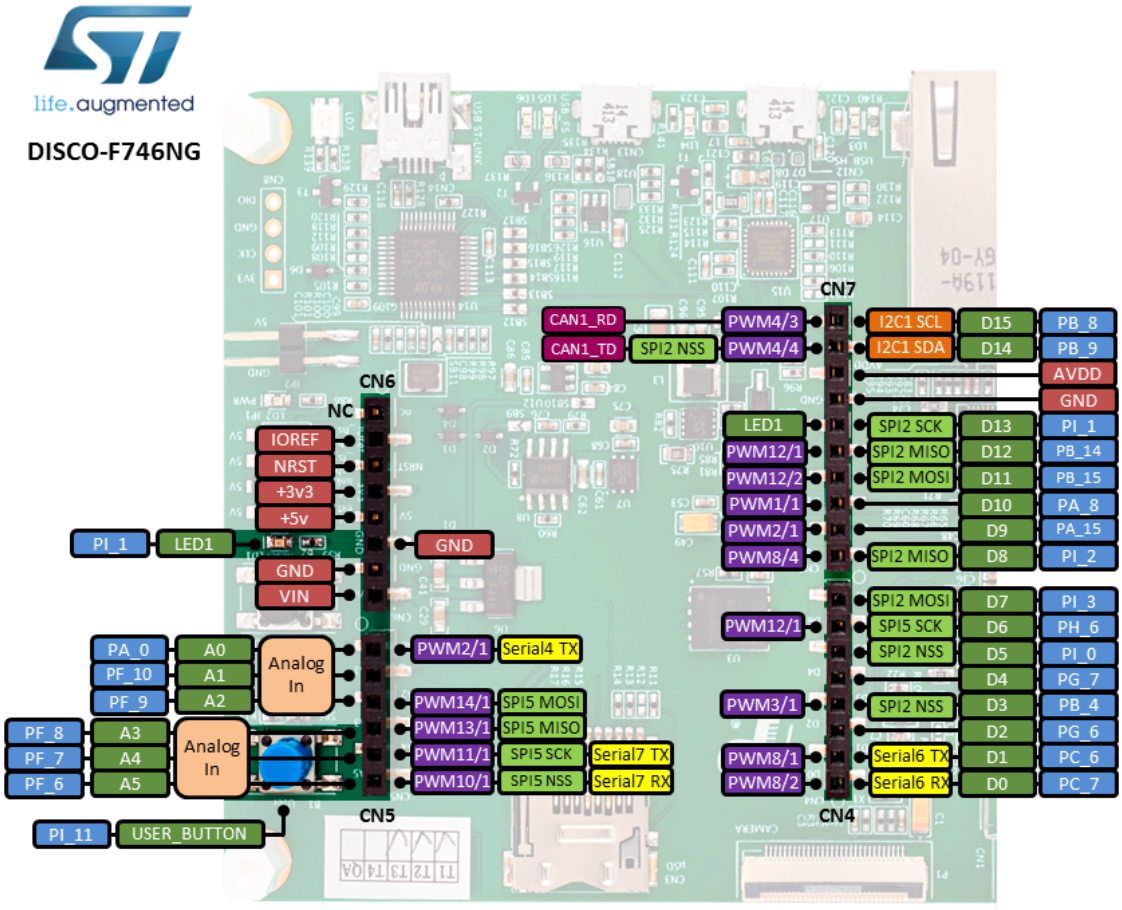
5. Use the LCD screen to display information received from the remote XBee sensor nodes (e.g. the XBee addresses and ADC values). See lab 103 for a reminder of how to use the LCD screen with the RTOS.

Appendix A – A simple packet processing method

```
// process an xbee packet
void process_packet(uint8_t* packet, int length)
{
    // check it is an explicit io sample received packet (because otherwise the
    // structure will be wrong)
    if(packet[3] == 0x92)
    {
        // print the xbee long address
        printf("xbee long address is: ");
        int i = 0;
        for(i = 4; i < 12; i++)
        {
            printf("%02X ", packet[i]);
        }
        printf("\r\n");

        // print the xbee short address
        printf("xbee short address is: ");
        printf("%02X %02X\r\n", packet[12], packet[13]);

        // print out the data that we have - this is the bit of the packet that
        // contains the adc values from the light sensor and temperature sensor
        int datastart = 16;
        if(datastart < (length - 1))
        {
            printf("data = ");
            while(datastart < (length - 1))
            {
                printf("%02X ", packet[datastart++]);
            }
            printf("\r\n");
        }
```

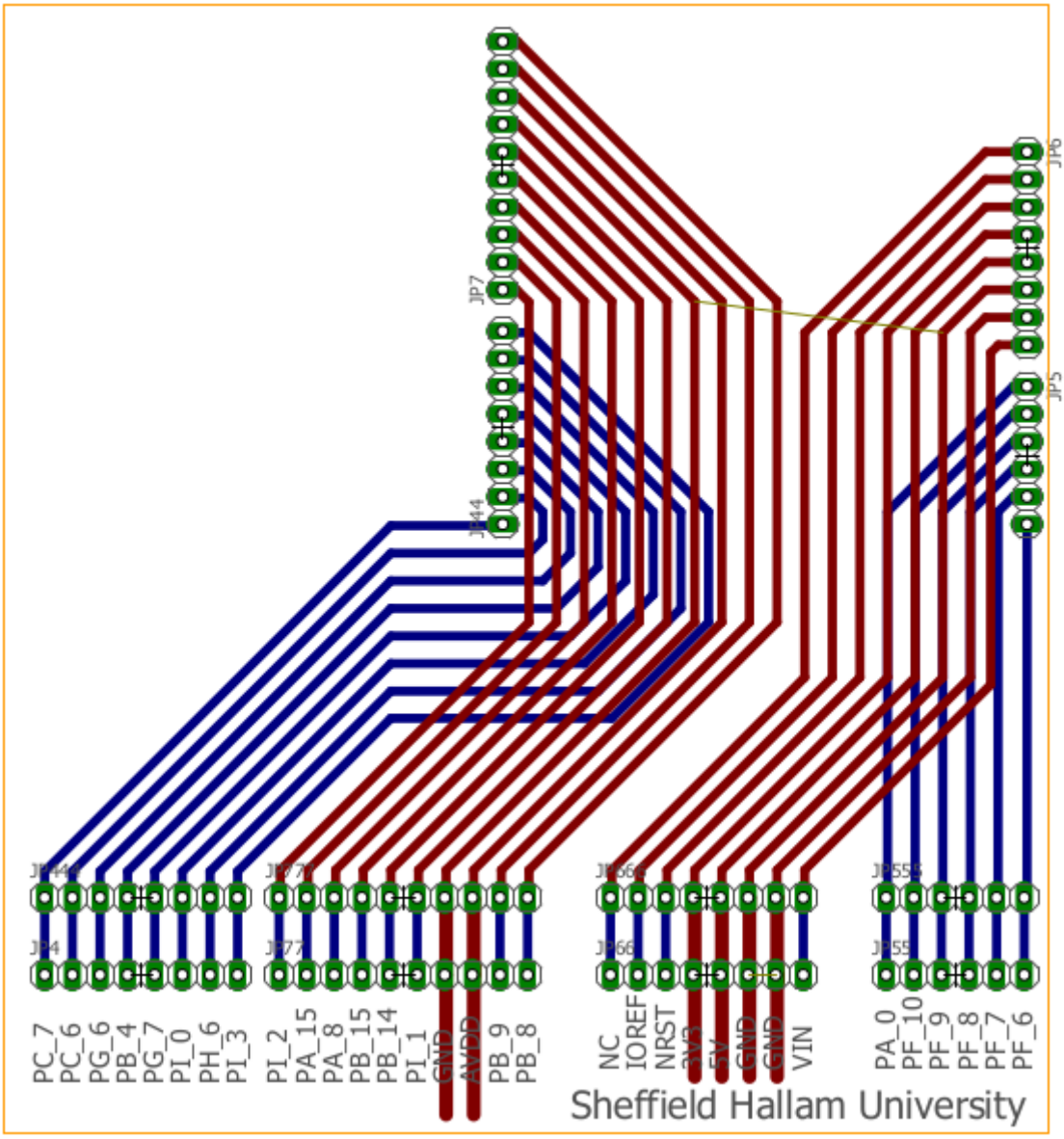
```
        // extract the raw adc values from the io sample rx packet
        //
        // the data field looks like:
        // 01                       number of samples
        // 00 00                    digital channel mask
        // 03                       analog channel mask
        // 02 7D 02 1C              analog samples
        //
        // note: we are assuming a maximum of 2 analog channels, no digital
        // channels, and that we are only using dio0 and dio1 (if we are using
        // other inputs we need to adjust this code)

        // if there are no digital channels ...
        if(packet[16] + packet[17] == 0)
        {
            // if there is one analog channel read that
            if(packet[18] == 1 || packet[18] == 2)
            {
                printf("adc 1 value is : %4d\r\n", (packet[19] << 8) | packet[20]);
            }
            // if there are two analog channels read both
            if(packet[21] == 3)
            {
                printf("adc 1 value is : %4d\r\n", (packet[19] << 8) | packet[20]);
                printf("adc 2 value is : %4d\r\n", (packet[21] << 8) | packet[22]);
            }
        }
        else
        {
            printf("network problems! %02X\r\n", packet[17]);
        }
        printf("\r\n");
    }
}
```

Appendix B – The STM32F7 discovery board schematic



STM32F7 discovery board pin outs

Appendix C – The STM32F7 discovery board SHU base board schematic



SHU breakout board for the STM32F7 discovery