



Πανεπιστήμιο Θεσσαλίας

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών
Υπολογιστών

Διπλωματική Εργασία

Τίτλος: “Ενοποίηση ασυρμάτων διεπαφών CC2500
με την πρωτότυπη πλατφόρμα ασυρμάτων
αισθητήρων του NITOS”



Όνοματεπώνυμο:
Δημήτριος Ζαφείρης

ΑΕΜ: 966

Επιβλέπων καθηγητής: Αθανάσιος Κοράκης
Συνεπιβλέπων καθηγητής: Σπυρίδων Λάλης



**ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΒΙΒΛΙΟΘΗΚΗ & ΚΕΝΤΡΟ ΠΛΗΡΟΦΟΡΗΣΗΣ
ΕΙΔΙΚΗ ΣΥΛΛΟΓΗ «ΓΚΡΙΖΑ ΒΙΒΛΙΟΓΡΑΦΙΑ»**

Αριθ. Εισ.: 13182/1
Ημερ. Εισ.: 01-04-2015
Δωρεά: Συγγραφέα
Ταξιθετικός Κωδικός: ΠΤ-ΗΜΜΥ
2014
ΖΑΦ



University of Thessaly

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING

Thesis

Title: “Integration of CC2500 wireless interfaces with
NITOS prototype wireless sensor platform”



Full name:
Dimitrios Zafeiris

AEM: 966

Supervisor professor: Athanasios Korakis
Co-supervisor professor: Spyridwn Lalis

Volos, September 2014

Ευχαριστίες

Η παρούσα Διπλωματική εργασία πραγματοποιήθηκε στο τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών στο Βόλο.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου, και λέκτορα του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, κ. Κοράκη Αθανάσιο, που μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το θέμα. Επίσης, θα ήθελα να ευχαριστήσω θερμά τον υποψήφιο Διδάκτορα, Καζδαρίδη Ιωάννη, που σε όλη την διάρκεια της Διπλωματικής μου ήταν εκεί και με καθοδηγούσε προς την σωστή κατεύθυνση, με χρήσιμες συμβουλές και προτάσεις. Τέλος, ένα μεγάλο ευχαριστώ σε όλους όσους μου συμπαραστάθηκαν όλο αυτό το διάστημα, και κυρίως την οικογένεια μου και τους φίλους μου.

CONTENTS

Ευχαριστίες.....	3
Περίληψη.....	6
Abstract.....	7
Motivation.....	8
1. Introduction.....	10
1.1 General information about CC2500.....	10
1.2. Packet Handling.....	11
1.3. Packet Format.....	12
1.4. Data buffering.....	12
1.5. General Control and Status Pins.....	12
1.6. Crystal oscillator.....	13
1.7. Pictures of CC2500.....	13
2. Arduino Board.....	15
2.1. General.....	15
2.2. Arduino's Hardware.....	15
2.3. Arduino's Software.....	16
3. Connection.....	18
3.1. General.....	18
3.2. SPI.....	19
3.3. Two ways of connection.....	19
3.3.1. Connection using a breadboard.....	19
3.3.2. Connection through a shield.....	21
3.3.3. NITOS CC2500 Shield.....	22
3.4. Compare the two ways.....	22
4. Code.....	23
4.1. General.....	23
4.2. Code Example.....	23
4.3. Flowchart of CC2500 procedures.....	25
4.3.1. For our first library.....	25
4.3.2. For our second library.....	26
4.4. Comments on code.....	27
5. Initialization of CC2500.....	29
6. Addressing.....	31
7. Throughput Performance.....	32
7.1. General.....	32
7.2. Examine Throughput Performance.....	32
7.3. Diagram of Throughput Performance.....	33
7.4. Comments on throughput.....	33
8. Range of CC2500.....	34
8.1. General.....	34
8.2. Examine performance at different distances.....	34
8.2.1. First experiment without obstacles.....	34
8.2.2. Second Experiment with obstacles.....	35
8.2.3. Overall distance performance diagram.....	36
9. Features of CC2500.....	37
9.1. Wake-on Radio.....	37
9.2. Different TX power.....	38
9.3. LQI and RSSI.....	39
9.3.1. General for LQI and RSSI.....	39
9.3.2. How RSSI and LQI calculated.....	40
9.3.2. RSSI Diagrams.....	41
9.3.2.1. General.....	41
9.3.2.2. RSSI diagrams with different channels.....	41

9.3.2.3. RSSI diagrams with different distances.....	44
9.4. Clear Channel Assessment (CCA).....	44
9.5. Burst Transmission	45
9.6. Frequency hopping.....	45
9.7. Data whitening.....	45
10. Frequency.....	46
11. CC Debugger	47
11.1. General.....	47
11.2. Features of SmartRF	49
11.2.1. Generate header files.....	49
11.2.2. See RSSI – time diagrams for a communication	49
11.2.3. Compute packet error rate.....	50
11.2.4. Give Device Commands	50
11.3. SmartRF example.....	51
12. CC2500 with MSP430	53
12.1. General.....	53
12.2. MSP430 with SmartRF	54
13. Future work.....	56
13.1. Current Consumption.....	56
13.2. Wake-on Radio feature	56
13.3. TinyOS library	56
References.....	57

Περίληψη

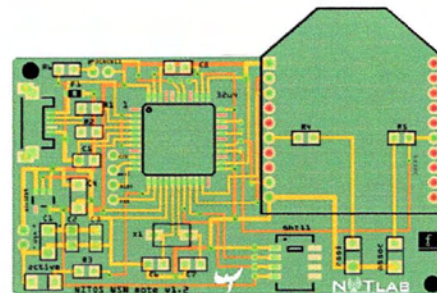
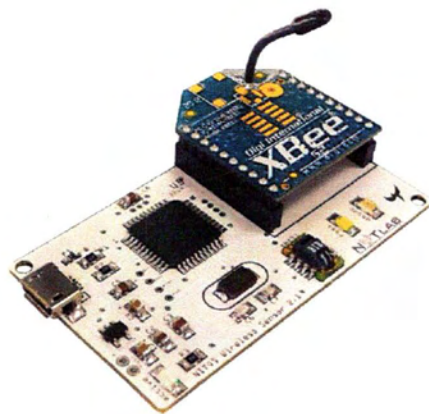
Αυτή η διπλωματική εργασία παρουσιάζει τις δυνατότητες της ασύρματης διεπαφής CC2500 της εταιρίας Texas Instruments (TI), μέσω εκτενών πειραμάτων. Το CC2500 της TI είναι ένας χαμηλής-ισχύος RF πομποδέκτης που λειτουργεί στο φάσμα συχνοτήτων 2.4GHz ISM. Σε αυτή την εργασία ενσωματώσαμε το CC2500 με πλατφόρμες μικροεπεξεργαστών Arduino, προκειμένου να αναπτυχθούν πρωτότυπα πειραματισμού. Αξιοποιώντας αυτά τα πρωτότυπα, εκτελέσαμε κάποια πειράματα και χαρακτηρίσαμε την απόδοση του CC2500 κάτω από ποικίλες διαμορφώσεις, όπως διαφορετικά Tx power και διαφορετικά εύρη επικοινωνίας, όπως επίσης και με ποικίλα payload μεγέθη για κάθε πακέτο. Επιπλέον, εξετάσαμε τις ειδικές λειτουργίες του CC2500 όπως το wake-on-radio και το clear channel assessment. Τέλος, εκμεταλλευτήκαμε το λειτουργικό SmartRF που μας παρέχει πλήρη πρόσβαση στους registers του CC2500, τους οποίους δεν μπορούμε να ρυθμίσουμε μέσω μιας πλατφόρμας μικροεπεξεργαστή.

Abstract

This thesis presents the capabilities of Texas Instruments (TI) CC2500 wireless interface, through extensive experiments. TI CC2500 is a low-power, RF transceiver operating at 2.4GHz ISM band. In this work we interface CC2500 with Arduino micro-controller boards in order to develop experimentation prototypes. Utilizing those prototypes we run experiments and we characterize the performance of CC2500 under various configurations, such as different Tx power and different communication range as well as various packet payload size. Moreover, we examine special features of CC2500 like the wake-on-radio and clear channel assessment procedure. Finally, we exploit TI Smart-RF studio which provides full access to CC2500 registers that cannot be configured through a micro-controller board.

Motivation

The NITOS prototype wireless sensor mote has been designed by NITLAB. It is comprised of open-source and configurable modules. NITOS mote features the ATmega32u4 microcontroller running at 8MHz and operating at 3.3V. The aforementioned microcontroller is fully compatible with the Arduino platform that enables ease of software development and provides compatibility with several commercial sensing modules. Moreover, the platform is equipped with an Xbee radio interface that enables communication with the respective gateway. The Xbee module is a tiny device ideal for setting up mesh networks and has a defined rate of 250 kbps. This module uses the IEEE 802.15.4 [1] stack which is the basis for the Zigbee protocol. Apart from the Xbee [2] module, NITOS mote can also feature a WiFi wireless interface in order to communicate with WiFi gateways. The developed mote currently features specific sensing modules, an air temperature and humidity sensor, a light intensity sensor and a human presence sensor. Various types of sensing modules and actuators can be further integrated exploiting existing Arduino software that implements several existing communications protocols. The firmware can be easily uploaded through the on-board USB connection.



The ultimate goal of this thesis is to extend NITOS WSN prototype capabilities by enabling the integration of new wireless interfaces. To this end, this thesis evaluates the features of the Texas Instruments CC2500 wireless RF module, towards providing knowledge about its specifications and functionalities. Additionally, this work paves the way for the integration between NITOS nodes and CC2500 module.

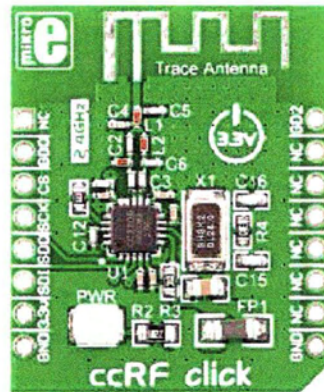
The motivation behind this work is to enhance NITOS node in terms of manufacturing cost, actual size and networking capabilities. As already

mentioned, NITOS node is based on XBee commercial interface for its wireless communication, which is a plug-and-play solution only basic configuration steps. However, XBee is not available at a good price in the market so it is deemed expensive for large-scale deployments. Additionally, XBee communicates with the microcontroller through a serial port (UART) in application layer that doesn't allow the configuration of complex MAC-layer parameters. In a few words XBee acts as a separate component on the NITOS node that doesn't expose all of its functionalities to be controlled and utilized. Moreover the firmware that Xbee runs it is closed-source, developed and distributed by Digi, not allowing the parameterization and evaluation of some features. Another drawback of XBee is the fact that draws high power consumption in relation to the rest wireless interfaces existing in the market applicable in wireless sensor platforms.

Considering all these drawbacks in this thesis we examine the potential of TI CC2500 module which features some progressive features such as wake-on-radio, etc. Additionally, CC2500 can be found in low-cost price in the market for large-scale deployments and draws much less power consumption compared to the XBee module. The other critical point for our decision is that CC2500 runs open-source stacks for the network communications which provides incentives for testing and experimentation.

1. Introduction

1.1 General information about CC2500



The CC2500 [3] is a low-cost 2.4 GHz transceiver designed for very low-power wireless applications. The circuit is intended for the 2400-2483.5 MHz ISM [4] (Industrial, Scientific and Medical) and SRD [5] (Short Range Device) frequency band and is developed by Texas Instruments [6].

CC2500 is used for:

- 2400-2483.5 MHz ISM/SRD band systems
- Consumer electronics
- Wireless game controllers
- Wireless audio
- Wireless keyboard and mouse
- RF [7] enabled remote controls

CC2500 provides extensive hardware support for packet handling, data buffering, burst transmissions [8], frequency hopping [9], clear channel assessment [10], link quality indication [11], different tx powers, data whitening [12] and wake-on-radio [13].

The main operating parameters and the 64-byte transmit/receive FIFOs of CC2500 can be controlled via an SPI [14] interface. In a typical system, the CC2500 will be used together with a microcontroller and a few additional passive components.

Key features of CC2500:

- High sensitivity (-104 dBm at 2.4 kBaud, 1% packet error rate)
- Low current consumption (13.3 mA in RX, 250 kBaud, input well above sensitivity limit)
- Programmable output power up to +1 dBm
- Excellent receiver selectivity and blocking performance
- Programmable data rate from 1.2 to 500 kBaud
- Frequency range: 2400 – 2483.5 MHz
- OOK, 2-FSK, GFSK, and MSK supported

- Suitable for frequency hopping and multi-channel systems due to a fast settling frequency synthesizer with 90 us settling time
- Automatic Frequency Compensation (AFC) can be used to align the frequency synthesizer to the received centre frequency
- Integrated analog temperature sensor
- Efficient SPI interface: All registers can be programmed with one “burst” transfer
- Digital RSSI [15] output
- Programmable channel filter bandwidth
- Programmable Carrier Sense (CS) indicator
- Programmable Preamble Quality Indicator (PQI) for improved protection against false sync word detection in random noise
- Support for automatic Clear Channel Assessment (CCA) before transmitting (for listen-before-talk systems)
- Support for per-package Link Quality Indication (LQI)
- Optional automatic whitening and de-whitening of data
- 400 nA SLEEP mode current consumption
- Fast startup time: 240 us from SLEEP to RX or TX mode (measured on EM design)
- Wake-on-radio functionality for automatic low-power RX polling
- Separate 64-byte RX and TX data FIFOs (enables burst mode data transmission)

1.2. Packet Handling

Transmit:

In transmit mode, the packet handler can be configured to add the following elements to the packet stored in the TX FIFO:

- A programmable number of preamble bytes
- A two byte synchronization (sync) word. Can be duplicated to give a 4-byte sync word (recommended). It is not possible to only insert preamble or only insert a sync word.
- A CRC checksum computed over the data field.

In addition, the following can be implemented on the data field and the optional 2-byte CRC checksum:

- Whitening of the data with a PN9 sequence.
- Forward error correction by the use of interleaving and coding of the data (convolutional coding).

Receive:

In receive mode, the packet handling support will de-construct the data packet by implementing the following (if enabled):

- Preamble detection
- Sync word detection
- CRC computation and CRC check

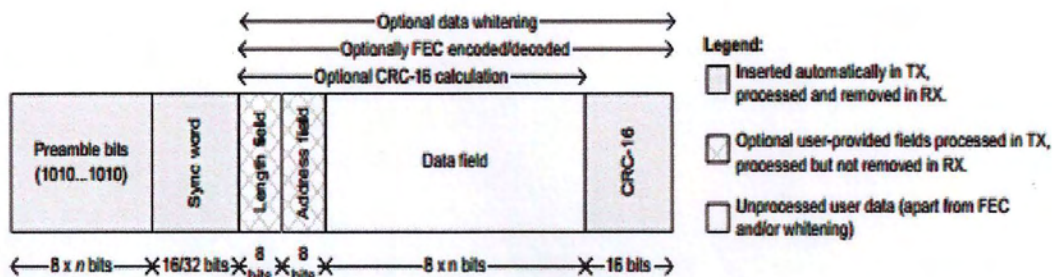
- One byte address check
- Packet length check (length byte checked against a programmable maximum length)
- De-whitening
- De-interleaving and decoding

Optionally, two status bytes with RSSI value, Link Quality Indication, and CRC status can be appended in the RX FIFO.

1.3. Packet Format

The format of the data packet can be configured and consists of the following items:

- Preamble
- Synchronization word
- Length byte or constant programmable packet length
- Optional address byte
- Payload
- Optional 2 byte CRC



1.4. Data buffering

The CC2500 contains two 64 byte FIFOs, one for received data and one for data to be transmitted. The SPI interface is used to read from the RX FIFO and write to the TX FIFO.

1.5. General Control and Status Pins

The CC2500 has two dedicated configurable pins (GDO0 and GDO2) and one shared pin (GDO1) that can output internal status information useful for control software. These pins can be used to generate interrupts on the MCU.

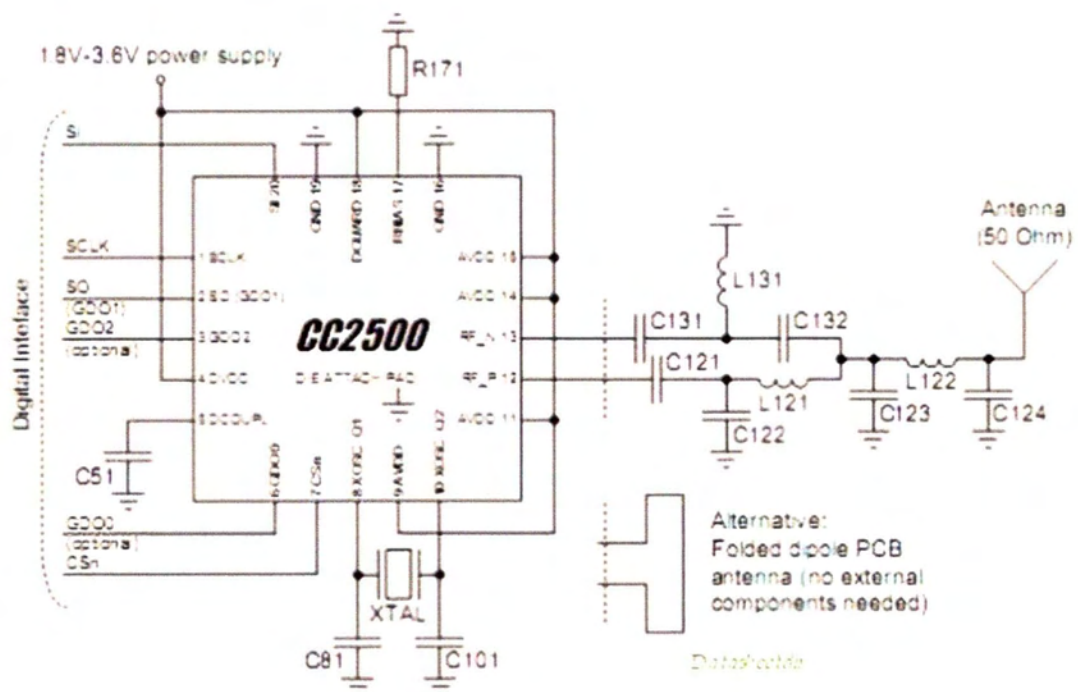
In the synchronous and asynchronous serial modes, the GDO0 pin is used as a serial TX data input pin while in transmit mode. The GDO0 pin can also be used for an on-chip analog temperature sensor. By measuring the voltage on the GDO0 pin with an external ADC, the temperature can be calculated.

In TX, the GDO0 pin is used for data input (TX data). Data output can be on GDO0, GDO1 or GDO2. This is set by the IOCFG0.GDO0_CFG, IOCFG1.GDO1_CFG and IOCFG2.GDO2_CFG fields.

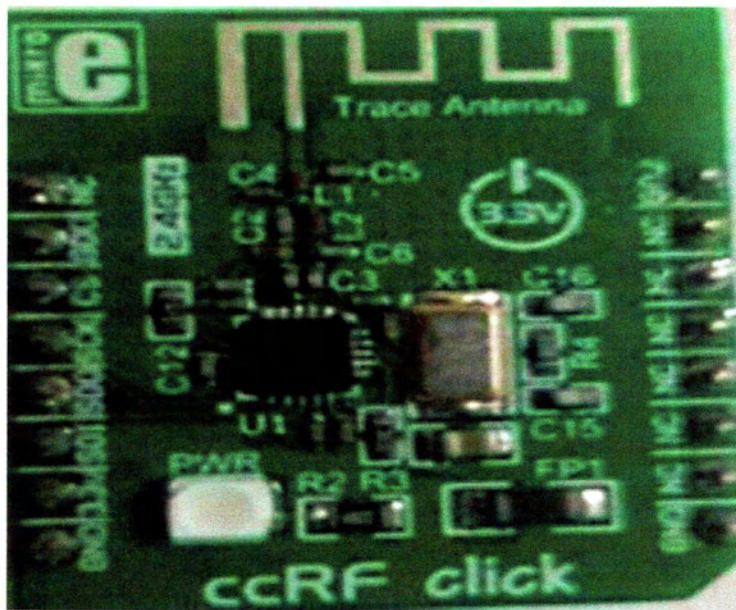
1.6. Crystal oscillator

The crystal oscillator generates the reference frequency for the synthesizer, as well as clocks for the ADC and the digital part. The crystal oscillator (XOSC) is either automatically controlled or always on, if MCSM0.XOSC_FORCE_ON is set. In the automatic mode, the XOSC will be turned off if the SXOFF or SPWD command strobes are issued; the state machine then goes to XOFF or SLEEP respectively. This can only be done from the IDLE state. The XOSC will be turned off when CSn is released (goes high). The XOSC will be automatically turned on again when CSn goes low. The state machine will then go to the IDLE state. The MISO pin on the SPI interface must be pulled low before the SPI interface is ready to be used. If the XOSC is forced on, the crystal will always stay on even in the SLEEP state.

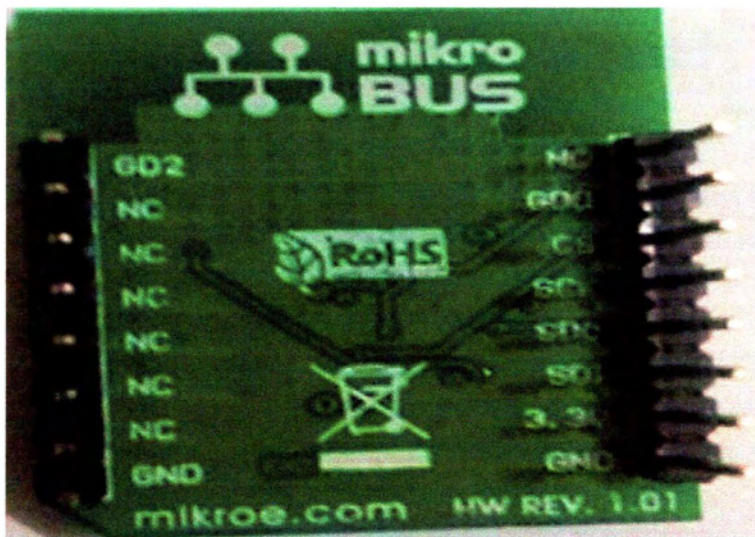
1.7. Pictures of CC2500



CC2500 circuit



Front view of a CC2500 chip



Rear view of a CC2500 chip

2. Arduino Board

2.1. General

Arduino [16] is a single-board microcontroller designed to make the process of using electronics in multidisciplinary projects more accessible. The hardware consists of a simple open source hardware board designed around an 8-bit Atmel AVR microcontroller, though a new model has been designed around a 32-bit Atmel ARM. The software consists of a standard programming language compiler and a boot loader that executes on the microcontroller.

Arduino boards can be purchased pre-assembled or do-it-yourself kits. Hardware design information is available for those who would like to assemble an Arduino by hand. There are sixteen official Arduinos that have been commercially produced to date.

2.2. Arduino's Hardware

Arduino is a single-board microcontroller, intended to make the application of interactive objects or environments more accessible. Current models feature a USB interface, 6 analog input pins, as well as 14 digital I/O pins which allow the user to attach various extension boards. An Arduino board consists of an Atmel 8-bit AVR microcontroller with complementary components to facilitate programming and incorporation into other circuits.

An Arduino board consists of an Atmel 8-bit AVR microcontroller with complementary components to facilitate programming and incorporation into other circuits. An important aspect of the Arduino is the standard way that connectors are exposed, allowing the CPU board to be connected to a variety of interchangeable add-on modules known as shields. Some shields communicate with the Arduino board directly over various pins, but many shields are individually addressable via an I²C serial bus, allowing many shields to be stacked and used in parallel. Official Arduinos have used the megaAVR series of chips, specifically the ATmega8, ATmega168, ATmega328, ATmega1280, and ATmega2560. A handful of other processors have been used by Arduino compatibles. Most boards include a 5 volt linear regulator and a 16 MHz crystal oscillator (or ceramic resonator in some variants), although some designs such as the LilyPad run at 8 MHz and dispense with the onboard voltage regulator due to specific form-factor restrictions. An Arduino's microcontroller is also pre-programmed with a boot loader that simplifies uploading of programs to the on-chip flash memory, compared with other devices that typically need an external programmer.

At a conceptual level, when using the Arduino software stack, all boards are programmed over an RS-232 serial connection, but the way this is implemented varies by hardware version. Serial Arduino boards contain a simple inverter circuit to convert between RS-232-level and TTL-level signals. Current Arduino boards are programmed via USB, implemented using USB-to-serial adapter chips such as the FTDI FT232. Some variants, such as the

Arduino Mini and the unofficial Boarduino, use a detachable USB-to-serial adapter board or cable, Bluetooth or other methods. (When used with traditional microcontroller tools instead of the Arduino IDE, standard AVR ISP programming is used.)

The Arduino board exposes most of the microcontroller's I/O pins for use by other circuits. The Diecimila, Duemilanove, and current Uno provide 14 digital I/O pins, six of which can produce pulse-width modulated signals, and six analog inputs. These pins are on the top of the board, via female 0.1 inch headers. Several plug-in application shields are also commercially available.

There are a great many Arduino-compatible and Arduino-derived boards. Some are functionally equivalent to an Arduino and may be used interchangeably. Many are the basic Arduino with the addition of commonplace output drivers, often for use in school-level education to simplify the construction of buggies and small robots. Others are electrically equivalent but change the form factor, sometimes permitting the continued use of Shields, sometimes not. Some variants even use completely different processors, with varying levels of compatibility.



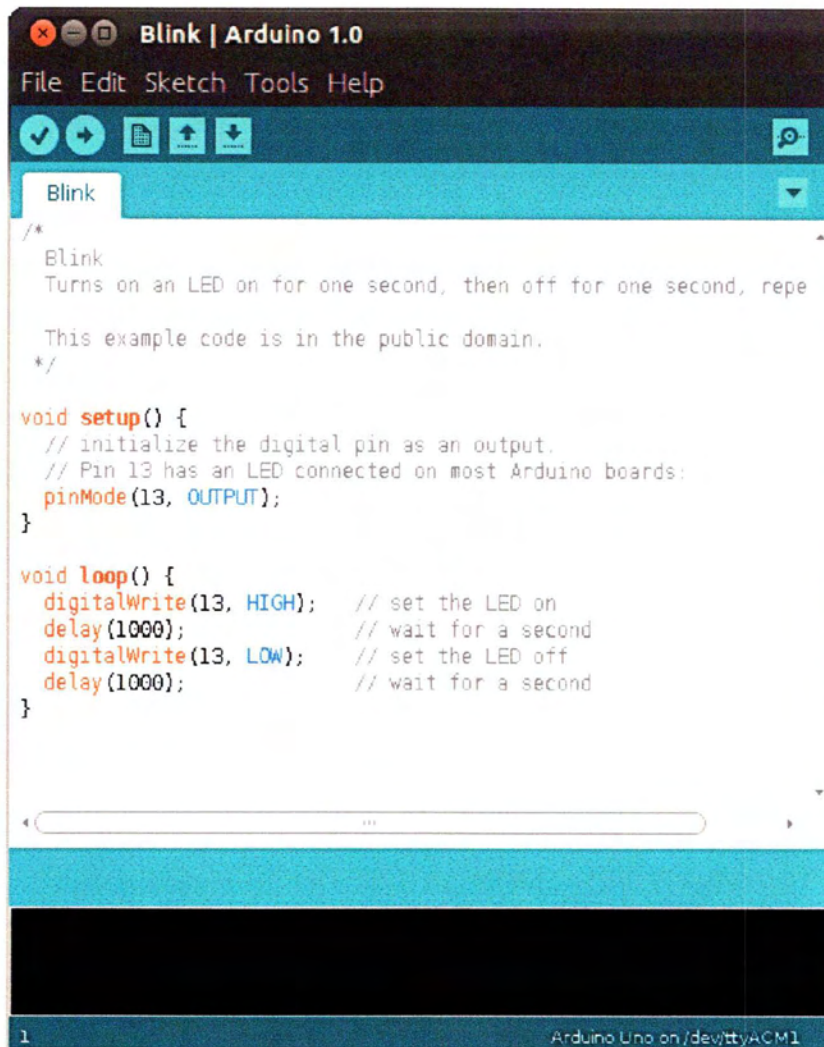
Arduino Leonardo

2.3. Arduino's Software

The Arduino integrated development environment (IDE) is a cross-platform application written in Java, and is derived from the IDE for the Processing programming language and the Wiring projects. It is designed to introduce programming to artists and other newcomers unfamiliar with software development. It includes a code editor with features such as syntax highlighting, brace matching, and automatic indentation, and is also capable of compiling and uploading programs to the board with a single click. There is typically no need to edit makefiles or run programs on a command-line interface.

Arduino programs are written in C or C++. The Arduino IDE comes with a software library called "Wiring" from the original Wiring project, which makes many common input/output operations much easier. Users only need define two functions to make a runnable cyclic executive program:

setup(): a function run once at the start of a program that can initialize settings
loop(): a function called repeatedly until the board powers off

A screenshot of the Arduino IDE interface. The window title is "Blink | Arduino 1.0". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu bar is a toolbar with icons for saving, running, and uploading. The main editor area shows the following code:

```
/*
 * Blink
 * Turns on an LED on for one second, then off for one second, repe
 *
 * This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH); // set the LED on
  delay(1000);           // wait for a second
  digitalWrite(13, LOW); // set the LED off
  delay(1000);           // wait for a second
}
```

The status bar at the bottom indicates "1" on the left and "Arduino Uno on /dev/ttyACM1" on the right.

Blink example in Arduino IDE

The Arduino IDE uses the GNU toolchain and AVR Libc to compile programs, and uses avrdude to upload programs to the board.

As the Arduino platform uses Atmel microcontrollers, Atmel's development environment, AVR Studio or the newer Atmel Studio, may also be used to develop software for the Arduino.

3. Connection

3.1. General

Transmit and receive FIFOs, as well as the main operating parameters of the CC2500, can be controlled via an SPI interface.

CC2500 can be used together with a microcontroller and a few additional passive components. So, we can connect the CC2500 with an Arduino board.

SPI connection can be done with Arduino's ICSP pins:



Notice that in Arduino Leonardo, SS is in digital pin17. Pin D17 (SS) does not have a corresponding pin on the ICSP header, not anywhere on the Arduino Leonardo board. In order to use it, we have to solder a wire to either the end of the Rx LED or to the empty space attached to it.



In order to avoid that, we can just replace SS with some other variable (e.g. CS) in our code. And then connect it with a corresponding pin on the Arduino board.

So, the CC2500 connections with Arduino are the followings:

CC2500	Arduino
3.3V	3.3V
Gnd	Gnd
SDI	MOSI (ICSP pins)
SDO	MISO (ICSP pins)
SCK	SCK (ICSP pins)
GDO	Pin3 (defined in code)
CS	Pin10 (defined in code)

3.2. SPI

CC2500 is configured via a simple 4-wire SPI compatible interface (MOSI, MISO, SCLK, CS) [17], where CC2500 is the slave. This interface is also used to read and write buffered data. All transfers on the SPI interface are done most significant bit first.

With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices.

Typically there are three lines common to all the devices:

- MISO (Master In Slave Out) - The Slave line for sending data to the master,
- MOSI (Master Out Slave In) - The Master line for sending data to the peripherals,
- SCK (Serial Clock) - The clock pulses which synchronize data transmission generated by the master

and one line specific for every device:

- SS (Slave Select) - the pin on each device that the master can use to enable and disable specific devices. When a device's Slave Select pin is low, it communicates with the master. When the SS pin is high, it ignores the master.

The SS pin must be kept low during transfers on the SPI bus. If SS goes high during the transfer of a header byte or during read/write from/to a register, the transfer will be cancelled. When SS is pulled low, the MCU must wait until CC2500 MISO pin goes low before starting to transfer the header byte. This indicates that the crystal is running. Unless the chip was in the SLEEP or XOFF states, the MISO pin will always go low immediately after taking SS low.

3.3. Two ways of connection

3.3.1. Connection using a breadboard

At first, we integrated CC2500 transceiver with the simple and common way, using a breadboard. CC2500 can be connected with an Arduino, with some wires.

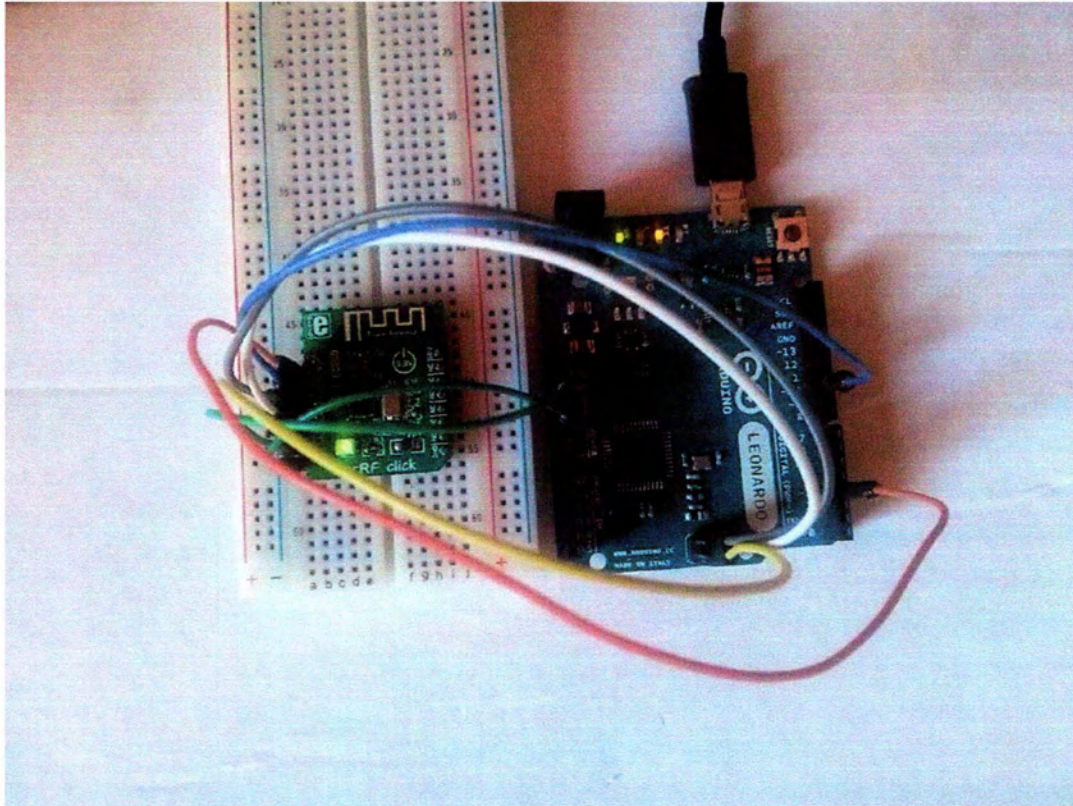
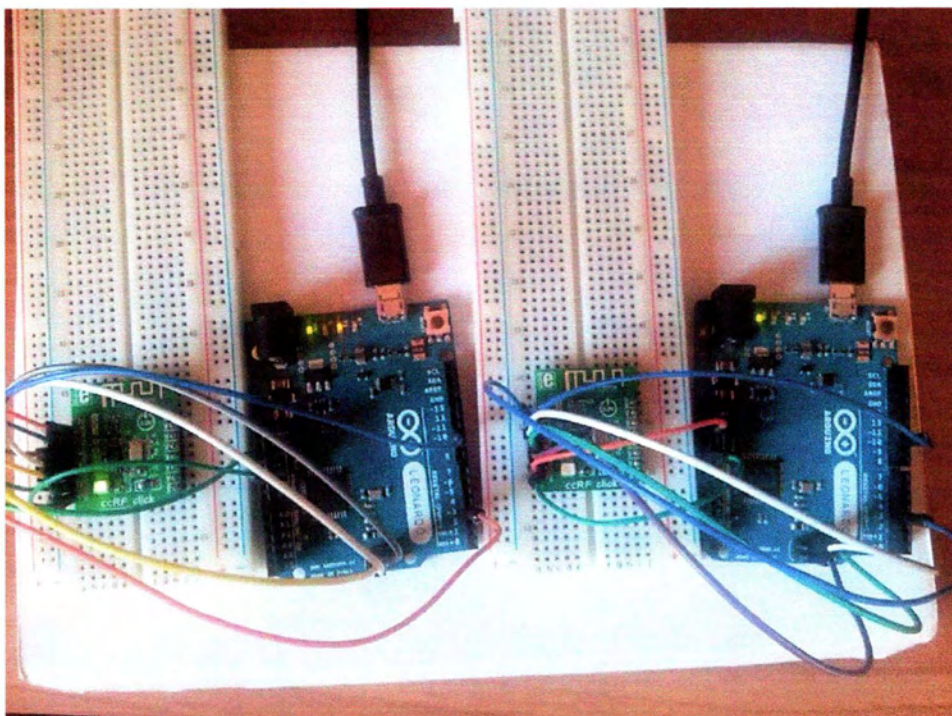


Photo of how a CC2500 can be connected with an Arduino Leonardo, using a breadboard

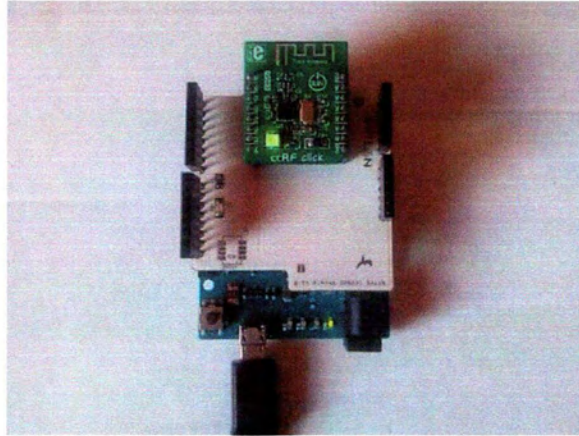
Two CC2500 chips are connected with an Arduino Leonardo, in order to achieve communication with each other. We deployed a set-up in which one Arduino+CC2500 operates as transmitter and the other one as receiver.



Picture of how we connected two CC2500 chips, using breadboards

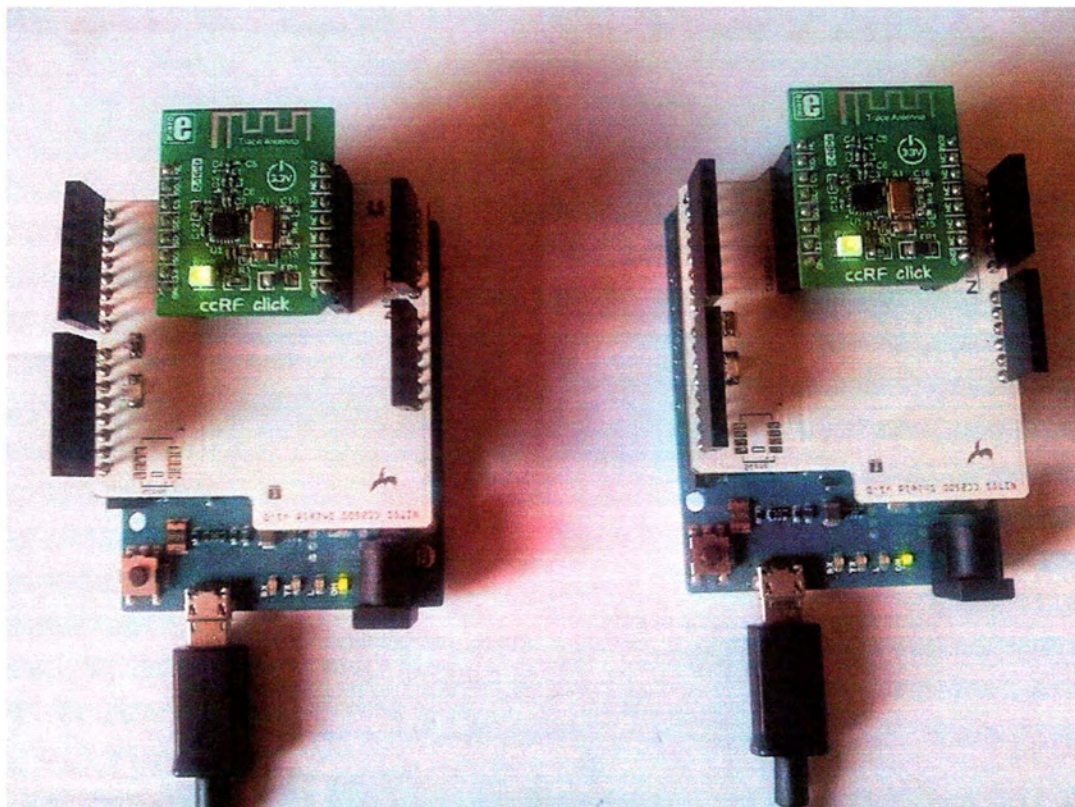
3.3.2. Connection through a shield

CC2500 can be connected with an Arduino board, through a shield.



CC2500+Arduino connection through a shield.

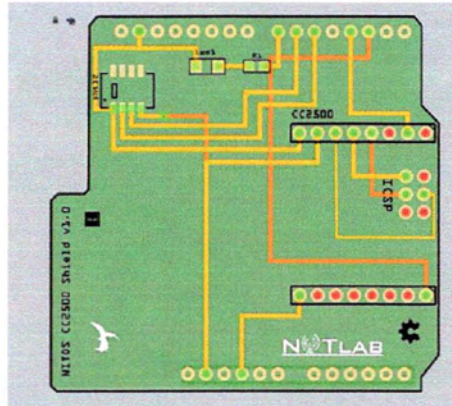
In order to communicate with two CC2500 chips, we have to get two same shields (constructed analogue to our pin connections).



Two CC2500 chips are connected with an Arduino Leonardo, through two same shields, in order to achieve communication with each other. We deployed a set-up in which one Arduino+CC2500 operates as transmitter and the other one as receiver.

3.3.3. NITOS CC2500 Shield

After all the wiring problems have been solved and as soon as we have a correct way of how CC2500 connect with the Arduino board, using a breadboard and some wires, we designed the shield.



This is the NITOS [18] CC2500 shield we've made, according to the correct pin connections we've found. In the figure you can see, how shield connect CC2500 with the Arduino via SPI and some other additional pins, like GDO0, and of course Voltage and Gnd.

3.4. Compare the two ways

SPI connection is really sensitive. So, in the first way, using a breadboard and so many wires, we have big losses (wires may be tangled up, damaged or just don't contact well with the boards).

In contrast, in the connection with the shield, we don't meet any serious losses issues, so we can maximize the performance of the communication.

4. Code

4.1. General

As mention, we used Arduino IDE to run our code. We achieved communication between two CC2500 with the help of these two libraries:

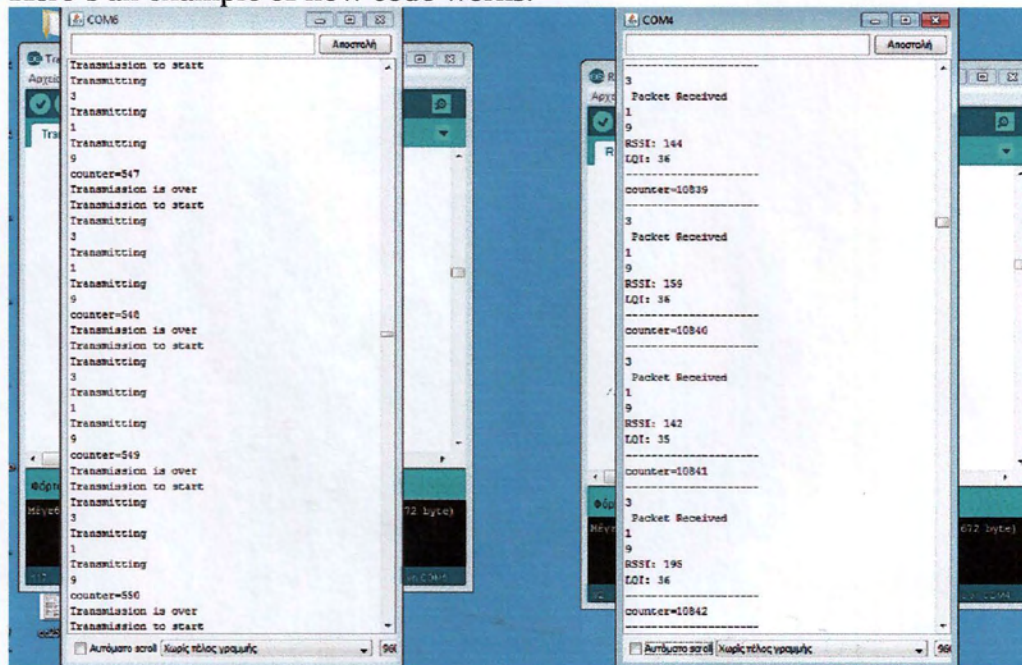
- <https://github.com/yasiralijaved/Arduino-CC2500-Library> [19]
- <https://github.com/Zohan/ArduinoCC2500Demo> [20]

The main idea is that the one Arduino+CC2500 operates as transmitter and the other one as receiver. The two main functions are TxData_RF() and RxData_RF(). Transmission is done by defining the packet length and writing every byte we want to transmit separately. The data are stored in TX FIFO. In receive mode, the data recovered and stored in RX FIFO in RxData_RF() procedure.

Other procedures we've used are init_CC2500, WriteReg, ReadReg, SendStrobe and Read_Config_Regs. In section 4.3, we explain how each procedure works.

4.2. Code Example

Here's an example of how code works:

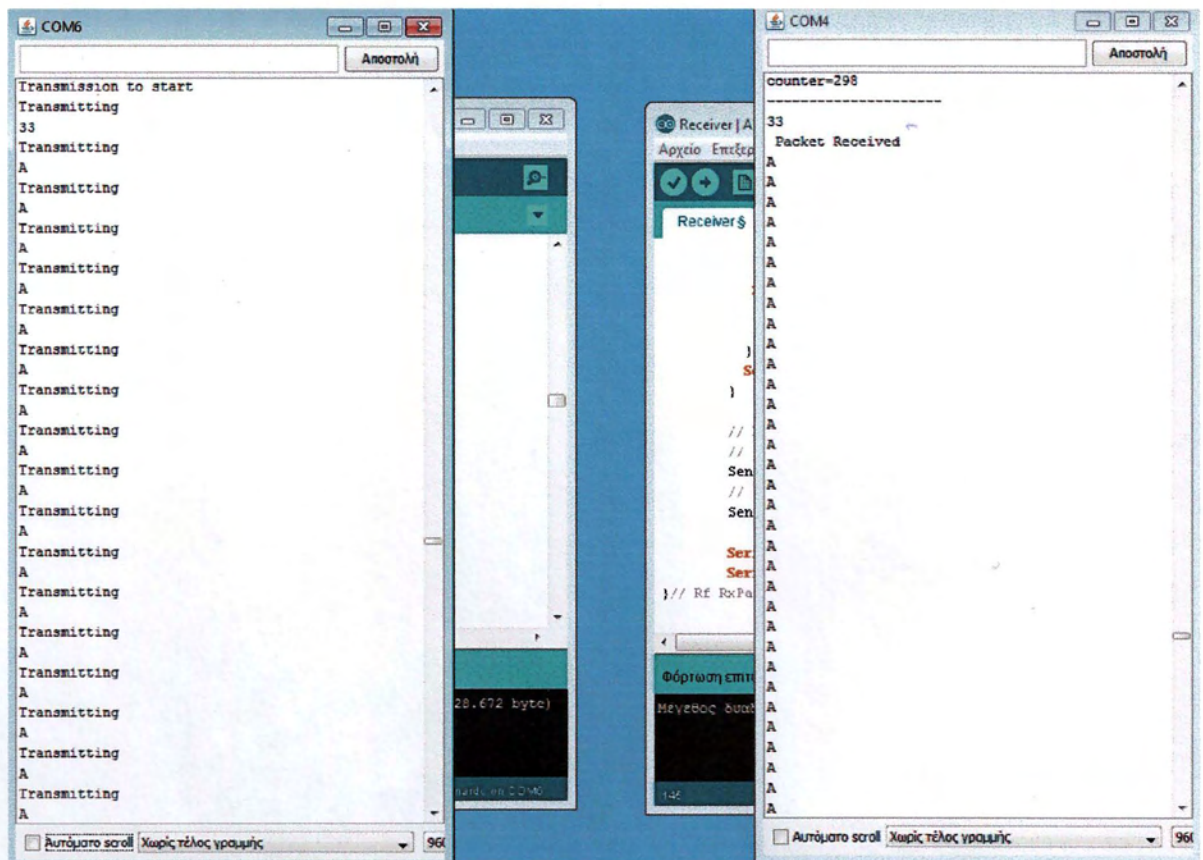


The image shows two side-by-side serial monitor windows from the Arduino IDE. The left window, titled 'COM6', shows the output of a transmitter. It displays a series of log messages: 'Transmission to start', 'Transmitting', and 'Transmission is over'. The payload for each transmission is shown as a sequence of three bytes: '3', '1', and '9'. A counter variable is also shown, increasing from 547 to 556. The right window, titled 'COM4', shows the output of a receiver. It displays 'Packet Received' messages with the same three-byte payload: '3', '1', and '9'. It also shows RSSI and LQI values for each packet and a counter variable increasing from 10839 to 10842.

On the left side of the image, you see the code and the results of the serial screen, for the Arduino+CC2500 that behave as transmitter. In this example, we've chosen to send packets with 3 bytes payload. The first is the number of bytes and the rest ones are the random HEX numbers. More specifically, the HEX number "0x01" and the HEX number "0x09". Also, there is a counter just to let us know how many packets have been sent.

On the right side, you see how the receiver code works. As correct packets received, number 3 has printed as well as the message "Packet Received". That means that we've received a correct number with 3 bytes packet payload. If we have received a correct packet, the packet payload will also be printed, as you see. In our case, HEX numbers "0x01" and "0x09". Optionally, after packet payload, LQI and RSSI values have been also printed, as they have been flushed in RX FIFO. Finally, there's a counter too, just for our help.

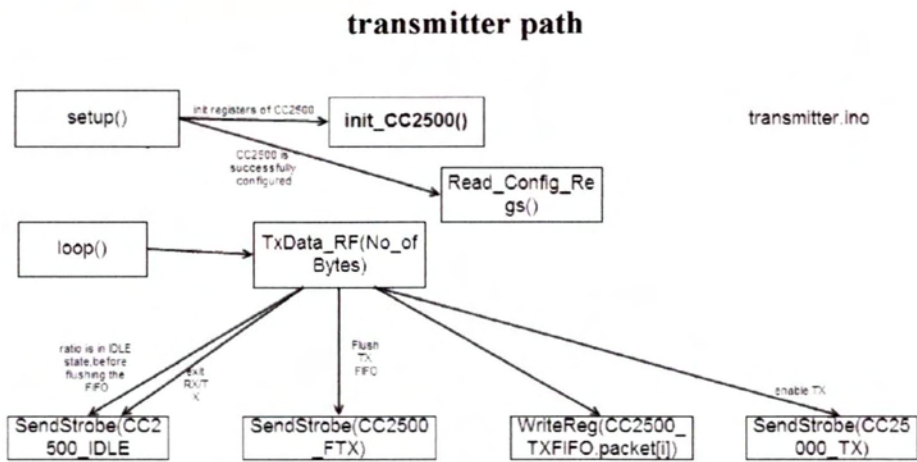
Here's another example of our code, when transmitting packets with 51 bytes payload. The first byte is for packet length and the next 50 bytes are the HEX number "0x0A".



On receiver side, number 33(0x33 is 51 in decimal) has been printed, as well as, 50 times the HEX number "0x0A".

4.3. Flowchart of CC2500 procedures

4.3.1. For our first library



Setup: initialize pins and begin SPI

Init_CC2500: init registers of CC2500, according to our pre-defined header files.

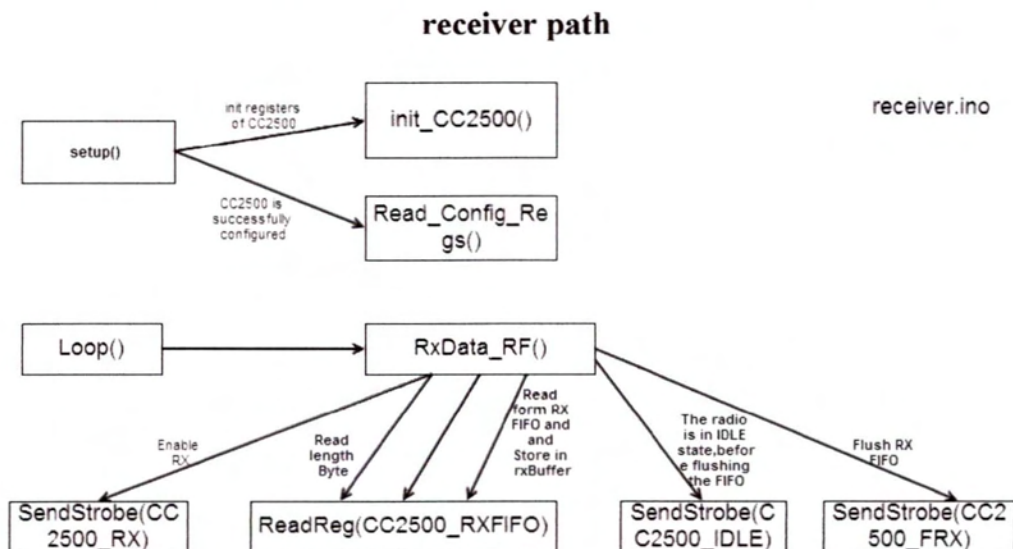
Read Config Regs: Read values of the registers of CC2500 and make sure that CC2500 is successfully configured.

Loop: Call the function TxRF to start transmission. Wait time for next transmission, can be modified with ready function delay().

TxData_RF: The procedure of transmitting a packet.

SendStrobe: Send Device commands. E.g. send CC2500 in SLEEP state (SPWD or SWOR), or in IDLE state (SIDLE).

WriteReg: Transfer SPI address and value. For register's values and data.



Setup: initialize pins and begin SPI

Init_CC2500: init registers of CC2500, according to our pre-defined header files.

Read_Config_Regs: Read values of the registers of CC2500 and make sure that CC2500 is successfully configured.

Loop: Call the function RxRF to receive a packet.

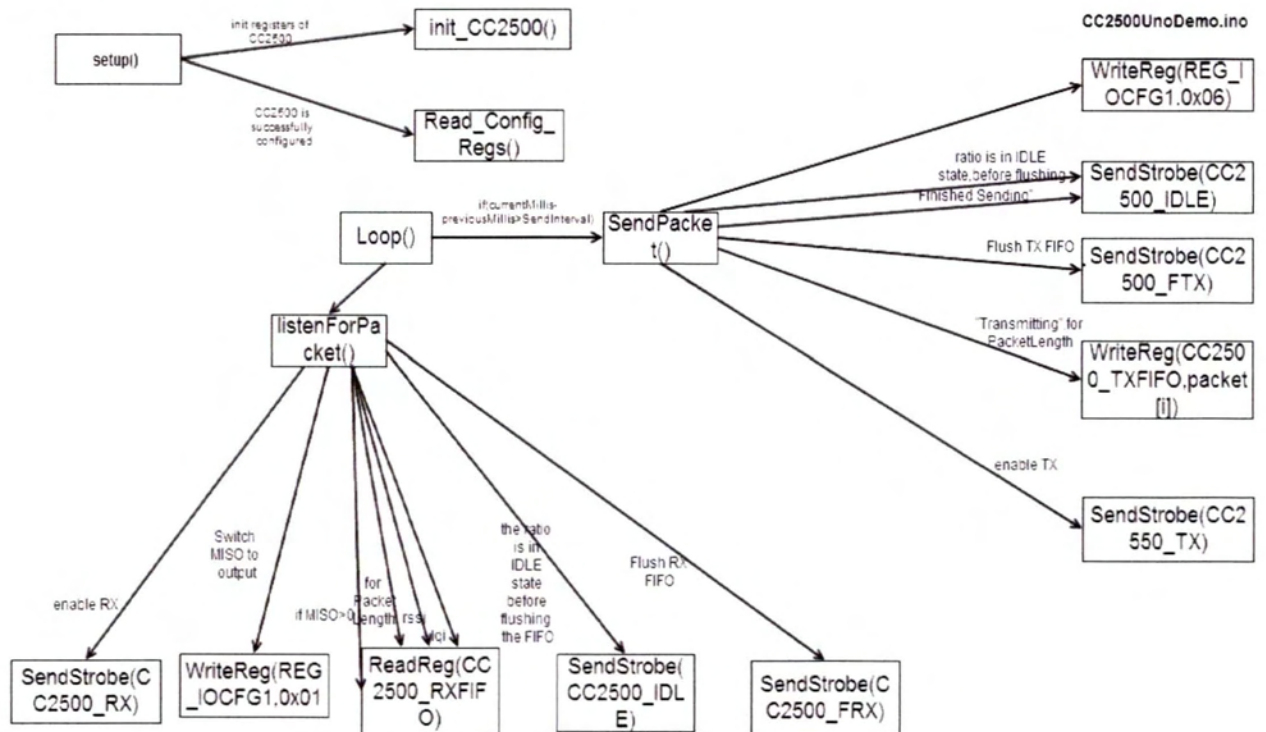
RxDData_RF: The procedure of receiving a packet.

SendStrobe: Send Device commands. E.g. send CC2500 in SLEEP state (SPWD or SWOR), or in IDLE state (SIDLE).

ReadReg: Read SPI address and value. For register's values and data.

4.3.2. For our second library

Flowchart for code CC2500UnoDemo.ino



Setup: initialize pins and begin SPI

Init_CC2500: init registers of CC2500, according to our pre-defined header files.

Read_Config_Regs: Read values of the registers of CC2500 and make sure that CC2500 is successfully configured.

Loop: Call sendPacket to transmit a packet or call listenForPacket to receive data

sendPacket: The procedure of transmitting a packet.

listenForPacket: The procedure of receiving a packet.

SendStrobe: Send Device commands. E.g. send CC2500 in SLEEP state (SPWD or SWOR), or in IDLE state (SIDLE).

WriteReg: Transfer SPI address and value. For register's values and data.

ReadReg: Read SPI address and value. For register's values and data.

4.4. Comments on code

- Sending RES Strobe in the setup function of both, transmitter and receiver, makes sure that the two CC2500 chips is programmed with the same registers, so they will communicate for sure.
- Before transmitting a packet and after exiting a transmission, it would be better to send SIDLE strobes. This action will move the chip to IDLE state. Fewer packets will be sent, as code become more complex, but it would help us receive packets faster.

Here's an example diagram without sending IDLE strobe:

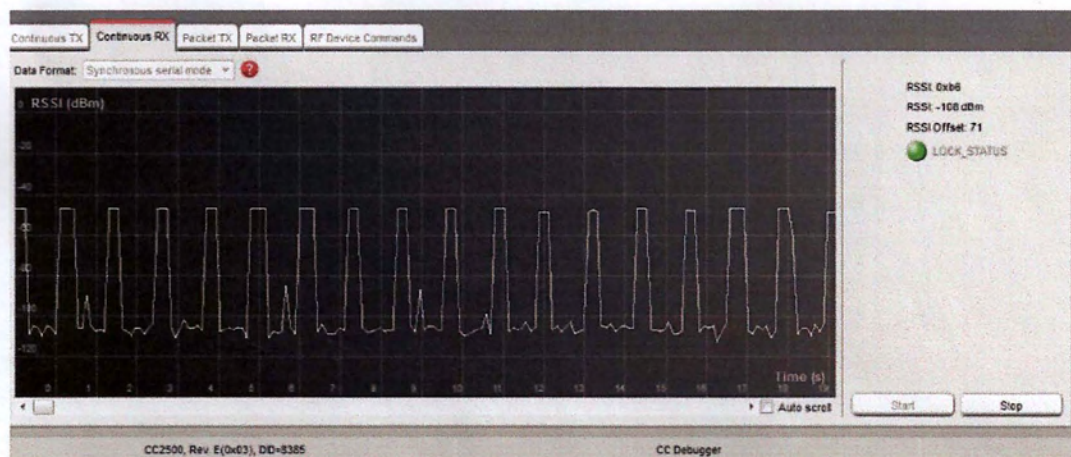


Diagram Signal Strength - Time

5. Initialization of CC2500

The configuration of CC2500 is done by programming 8-byte registers. After chip reset, all registers have default values. The optimum register setting might differ from the default value. After a reset all registers that shall be different from the default value therefore needs to be programmed through the SPI interface.

The optimum configuration data based on selected system parameters are most easily found by using the SmartRF Studio software. We'll show this software later.

There are 47 normal 8-bit configuration registers. Many of these registers are for test purposes only, and need not be written for normal operation of CC2500.

There are also 12 status registers. These registers, which are read-only, contain information about the status of CC2500 (like LQI and RSSI).

Also, there are 13 command strobe registers. Accessing these registers will initiate the change of an internal state or mode.

Strobes Table

Address	Strobe Name	Description
0x30	SRES	Reset chip.
0x31	SFSTXON	Enable and calibrate frequency synthesizer (if <code>MCSM0.FS_AUTOCAL=1</code>). If in RX (with CCA): Go to a wait state where only the synthesizer is running (for quick RX / TX turnaround).
0x32	SXOFF	Turn off crystal oscillator.
0x33	SCAL	Calibrate frequency synthesizer and turn it off. SCAL can be strobed from IDLE mode without setting manual calibration mode (<code>MCSM0.FS_AUTOCAL=0</code>)
0x34	SRX	Enable RX. Perform calibration first if coming from IDLE and <code>MCSM0.FS_AUTOCAL=1</code> .
0x35	STX	In IDLE state: Enable TX. Perform calibration first if <code>MCSM0.FS_AUTOCAL=1</code> . If in RX state and CCA is enabled: Only go to TX if channel is clear.
0x36	SIDLE	Exit RX / TX, turn off frequency synthesizer and exit Wake-On-Radio mode if applicable.
0x38	SWOR	Start automatic RX polling sequence (Wake-on-Radio) as described in Section 19.5 if <code>WORCTRL.RC_PD=0</code> .
0x39	SPWD	Enter power down mode when <code>CSn</code> goes high.
0x3A	SFRX	Flush the RX FIFO buffer. Only issue <code>SFRX</code> in IDLE or <code>RXFIFO_OVERFLOW</code> states.
0x3B	SFTX	Flush the TX FIFO buffer. Only issue <code>SFTX</code> in IDLE or <code>TXFIFO_UNDERFLOW</code> states.
0x3C	SWORRST	Reset real time clock to Event1 value.
0x3D	SNOP	No operation. May be used to get access to the chip status byte.

Registers Table

Address	Register	Description	Preserved in SLEEP State	Details on Page Number
0x00	IOCFG2	GD02 output pin configuration	Yes	61
0x01	IOCFG1	GD01 output pin configuration	Yes	61
0x02	IOCFG0	GD00 output pin configuration	Yes	61
0x03	FIFOTHR	RX FIFO and TX FIFO thresholds	Yes	62
0x04	SYNC1	Sync word, high byte	Yes	62
0x05	SYNC0	Sync word, low byte	Yes	62
0x06	PKTLEN	Packet length	Yes	62
0x07	PKTCTRL1	Packet automation control	Yes	63
0x08	PKTCTRL0	Packet automation control	Yes	64
0x09	ADDR	Device address	Yes	64
0x0A	CHANNR	Channel number	Yes	64
0x0B	FSCTRL1	Frequency synthesizer control	Yes	65
0x0C	FSCTRL0	Frequency synthesizer control	Yes	65
0x0D	FREQ2	Frequency control word, high byte	Yes	65
0x0E	FREQ1	Frequency control word, middle byte	Yes	65
0x0F	FREQ0	Frequency control word, low byte	Yes	65
0x10	MDMCFG4	Modem configuration	Yes	66
0x11	MDMCFG3	Modem configuration	Yes	66
0x12	MDMCFG2	Modem configuration	Yes	67
0x13	MDMCFG1	Modem configuration	Yes	68
0x14	MDMCFG0	Modem configuration	Yes	68
0x15	DEVIATN	Modem deviation setting	Yes	69
0x16	MCSM2	Main Radio Control State Machine configuration	Yes	70
0x17	MCSM1	Main Radio Control State Machine configuration	Yes	71
0x18	MCSM0	Main Radio Control State Machine configuration	Yes	72
0x19	FOCCFG	Frequency Offset Compensation configuration	Yes	73
0x1A	BSCFG	Bit Synchronization configuration	Yes	74
0x1B	AGCTRL2	AGC control	Yes	75
0x1C	AGCTRL1	AGC control	Yes	76
0x1D	AGCTRL0	AGC control	Yes	77
0x1E	WOREVT1	High byte Event 0 timeout	Yes	77
0x1F	WOREVT0	Low byte Event 0 timeout	Yes	78
0x20	WORCTRL	Wake On Radio control	Yes	78
0x21	FREND1	Front end RX configuration	Yes	78
0x22	FREND0	Front end TX configuration	Yes	79
0x23	FSCAL3	Frequency synthesizer calibration	Yes	79
0x24	FSCAL2	Frequency synthesizer calibration	Yes	79
0x25	FSCAL1	Frequency synthesizer calibration	Yes	80
0x26	FSCAL0	Frequency synthesizer calibration	Yes	80
0x27	RCCTRL1	RC oscillator configuration	Yes	80
0x28	RCCTRL0	RC oscillator configuration	Yes	80
0x29	FSTEST	Frequency synthesizer calibration control	No	80
0x2A	PTEST	Production test	No	80
0x2B	AGCTEST	AGC test	No	81
0x2C	TEST2	Various test settings	No	81
0x2D	TEST1	Various test settings	No	81
0x2E	TEST0	Various test settings	No	81

6. Addressing

CC2500 supports address filtering. It's a way of packet filtering. Setting the value of the register PKTCTRL1.ADR_CHK to any other value, except from 0, enables the packet address filter.

In detail:

- 00: No address check.
- 01: Address check, compare destination address byte in the packet with the value of ADDR register.
- 10: Check broadcast address 0x00.
- 11: Check both, 0x00 and 0xFF broadcast addresses.

If the received address matches a valid address, the packet is received and written into the RX FIFO. If the address match fails, the packet is discarded and receive mode restarted.

If the received address matches a valid address when using infinite packet length mode and address filtering is enabled, 0xFF will be written into the RX FIFO followed by the address byte and then the payload data.

Note1: If the value of the register PKTCTRL1.ADR_CHK is 00, then there is no address check. Packet filtration can be done with maximum length filtering or CRC filtering.

Note2: Optional broadcast addresses are 0 (0x00) and 255 (0xFF).

7. Throughput Performance

7.1. General

As soon as, we achieved to communicate with two CC2500 chips, we had to test the performance of our communication.

As mention, CC2500 has two 64 bytes FIFOs, and data payload is stored there, for every packet we have sent or received. That means that we are able to send packets with different packet payload. Specifically, our range extends from 1 to 64 bytes. For that reason, we have to examine throughput in any case, so we are able to estimate when CC2500 transceivers maximize their performance.

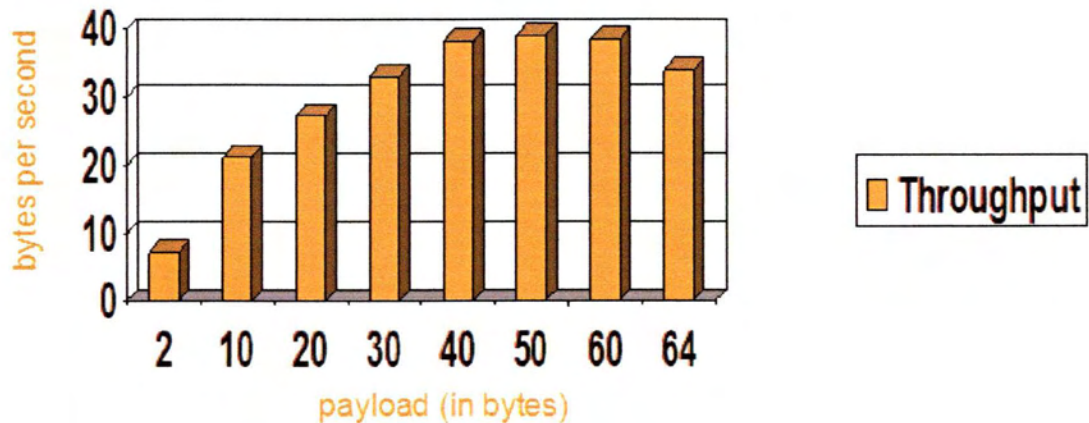
7.2. Examine Throughput Performance

We examined throughput [21] performance in different cases. In detail:

- 2 bytes payload: 144 from 150 packets delivered correctly in a minute. Throughput = 7.2 bytes per second.
- 10 bytes payload: 116 from 130 packets delivered correctly in a minute. Throughput=21.2 bytes per second.
- 20 bytes payload: 78 from 112 packets delivered correctly in a minute. Throughput=27.3 bytes per second.
- 30 bytes payload: 64 from 110 packets delivered correctly in a minute. Throughput: 33 bytes per second.
- 40 bytes payload: 56 from 104 packets delivered correctly in a minute. Throughput=38.2 bytes per second.
- 50 bytes payload: 46 from 100 packets delivered correctly in a minute. Throughput=39.1 bytes per second.
- 60 bytes payload: 38 from 88 packets delivered correctly in a minute. Throughput=38.6 bytes per second.
- 64 bytes payload: 32 from 88 packets delivered correctly in a minute. Throughput=34.1 bytes per second.

7.3. Diagram of Throughput Performance

In order to have more safe results about throughput performance, we made a diagram. It shows how throughput has been influenced, as we increased the payload of the packet.



Vertically: Throughput in bytes per second
Horizontal: packet payload in bytes

7.4. Comments on throughput

Packets with payload over 64 bytes don't delivered at all (CC2500 has two 64 bytes FIFOs for packet payload).

As we see, max throughput can be achieved when transmitting packets 50 bytes long. That's why, even if we transmit packets with bigger payload, we have bigger losses of packets when payload increases. So, throughput starts to decrease from the level of 50 bytes payload and above.

LQI and RSSI values are automatically added as two extra bytes appended after the payload (when `PKTCTRL1.APPEND_STATUS=1`). This register has default value 1, so one thought would be to make the value of `PKTCTRL1.APPEND_STATUS=0`, in order not to carry these two metrics as two extra bytes after the payload. But, all the experiments we've made in this case, didn't even give us different results, compared with the above diagram. So, making `PKTCTRL1.APPEND_STATUS=0` won't improve throughput performance.

Optional address byte is being stored in TX FIFO when address filtering is enabled. So, in order to maximize the beneficial bytes of our transmission, we'll just have to disable address check. This would release one byte to the TX FIFO. But, not having address check is not always a required situation for our transceivers.8. Range of CC2500

8. Range of CC2500

8.1. General

CC2500 is a transceiver designed for wireless applications. So, it is reasonable the performance of a communication between two CC2500, to affect when transmitting or receiving from different distances.

We have to examine how CC2500 behaves in different distances, in order to achieve better communication in specific conditions.

8.2. Examine performance at different distances

We made two different experiments to figure out how CC2500 behaves in different distances.

First, we examine how CC2500 behaves when the communication is done in the same room. The one CC2500 was straight ahead from the other, without physical obstacles.

Then, we tried to communicate from different rooms, with walls and other physical obstacles to affect the communication.

8.2.1. First experiment without obstacles

For our first experiment, (same room without obstacles) we've noted the following results:

- No distance: 136 of 140 packets delivered correctly (0.97).
- 1m: 122 of 140 packets delivered correctly (0.87).
- 2m: 114 of 140 packets delivered correctly (0.81).
- 3m: 102 of 140 packets delivered correctly (0.72).
- 5m: 88 of 140 packets delivered correctly (0.63).
- 7m: 86 of 140 packets delivered correctly (0.61).
- 7.5m and above: nothing happened.

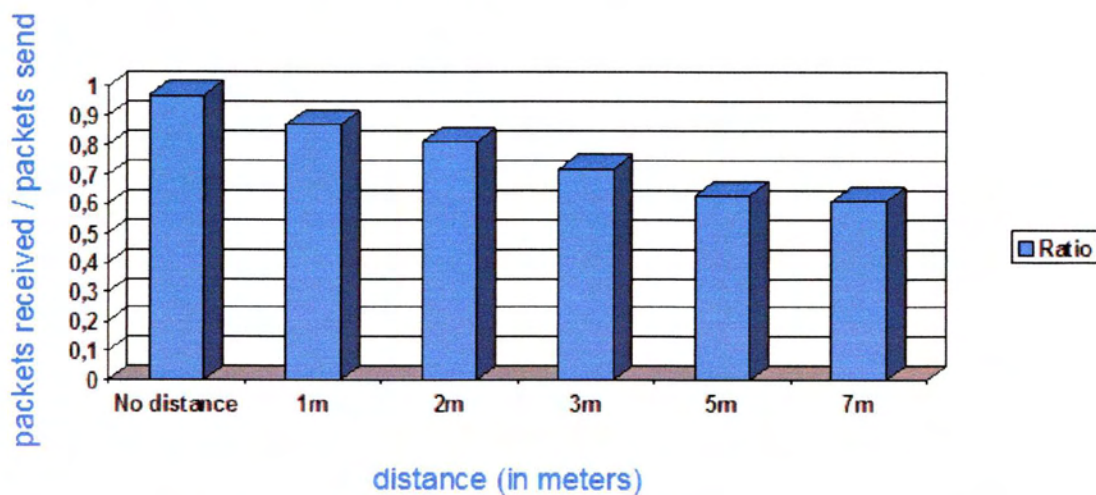
As we clearly see, as distance increases, fewer packets delivered correctly from the receiver.

That's what we want to show, as also to find the limit of how far two CC2500 can communicate with each other.

In detail, at 7 meters distance, we had a pretty good packet delivery ratio (0.61), depending on the distance we've been. But, when we put the chip a little longer (about 7.5 meters or even less), packets didn't delivered at all. So, in conclusion, the limit for a successive transmission of a packet is about 7.1-7.5 meters.

Also, as noticed from the experiment above, when two CC2500 is almost beside to each other, the packet delivery ratio is quite high (0.96) .So, it's the perfect condition to transmit and receive data as it's complete reasonable for a wireless communication.

Here's the diagram of the first experiment:



Vertically: Packet delivery ratio (packets received correctly / packets send)
 Horizontal: Distance in meters

8.2.2. Second Experiment with obstacles

In the second experiment we've made, we tried to make two CC2500 communicate from different rooms, with walls and other objects between them. We've had the following results:

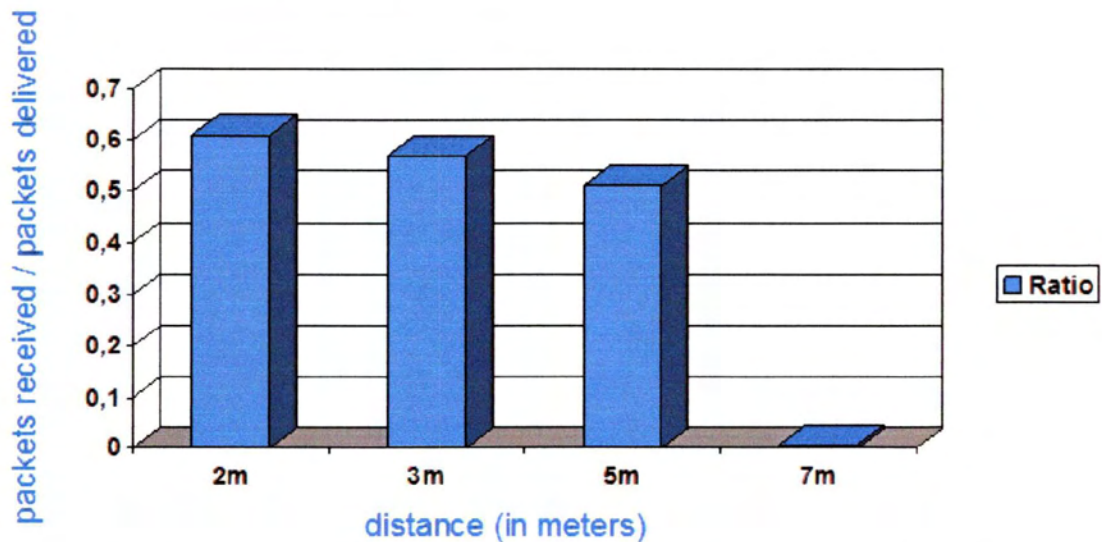
- 2m: 86 of 140 packets delivered correctly (0.61).
- 3m: 80 of 140 packets delivered correctly (0.57).
- 5m: 72 of 140 packets delivered correctly (0.51).
- 7m: About 10 packets of 140 delivered correctly (0.07).

As noticed, the packet delivery ratio is lower now, than it was in the first case without obstacles. That's something normal and something that we want to show and for that reason, we've made this test.

Wireless communication is affected by obstacles, or even the conditions that are prevailing in a room, so the results we've found are normal.

The limit of a successive transmission here is about 7 meters, even though the packet radio in this case is extremely low (0.07) and it is not worth enough communicating from this distance.

Here's the diagram of the second experiment:

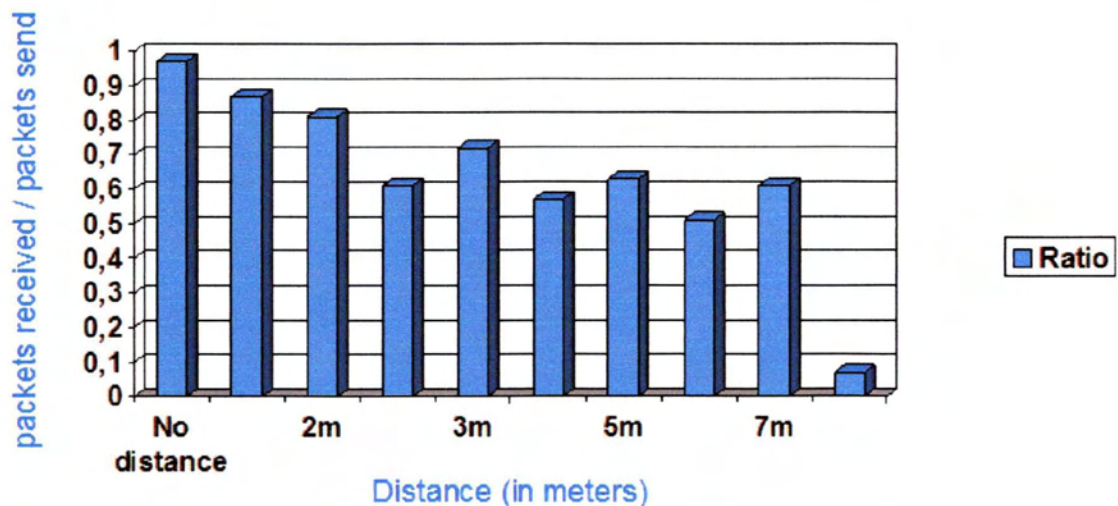


Vertically: Packet delivery ratio (packets received correctly / packets send)

Horizontal: Distance in meters

8.2.3. Overall distance performance diagram

Here's the overall diagram that resulted for both tests:



Vertically: Packet delivery ratio (packets received correctly / packets send)

Horizontal: Distance in meters

9. Features of CC2500

9.1. Wake-on Radio

The Wake on Radio (WOR) functionality enables CC2500 to periodically wake up from SLEEP and listen for incoming packets without MCU interaction. CC2500 can be set in SLEEP mode and then notify and go again in a regular state.

It is important because current consumption in SLEEP mode is only 400nA. In order to go in SLEEP mode, CC2500 has to be in IDLE state first. IDLE state is a state that CC2500 doesn't transmit or receive. CC2500 is just waiting. Current consumption in IDLE state is 1.5mA. It is possible to go in IDLE state by just sending the SIDLE command strobe.

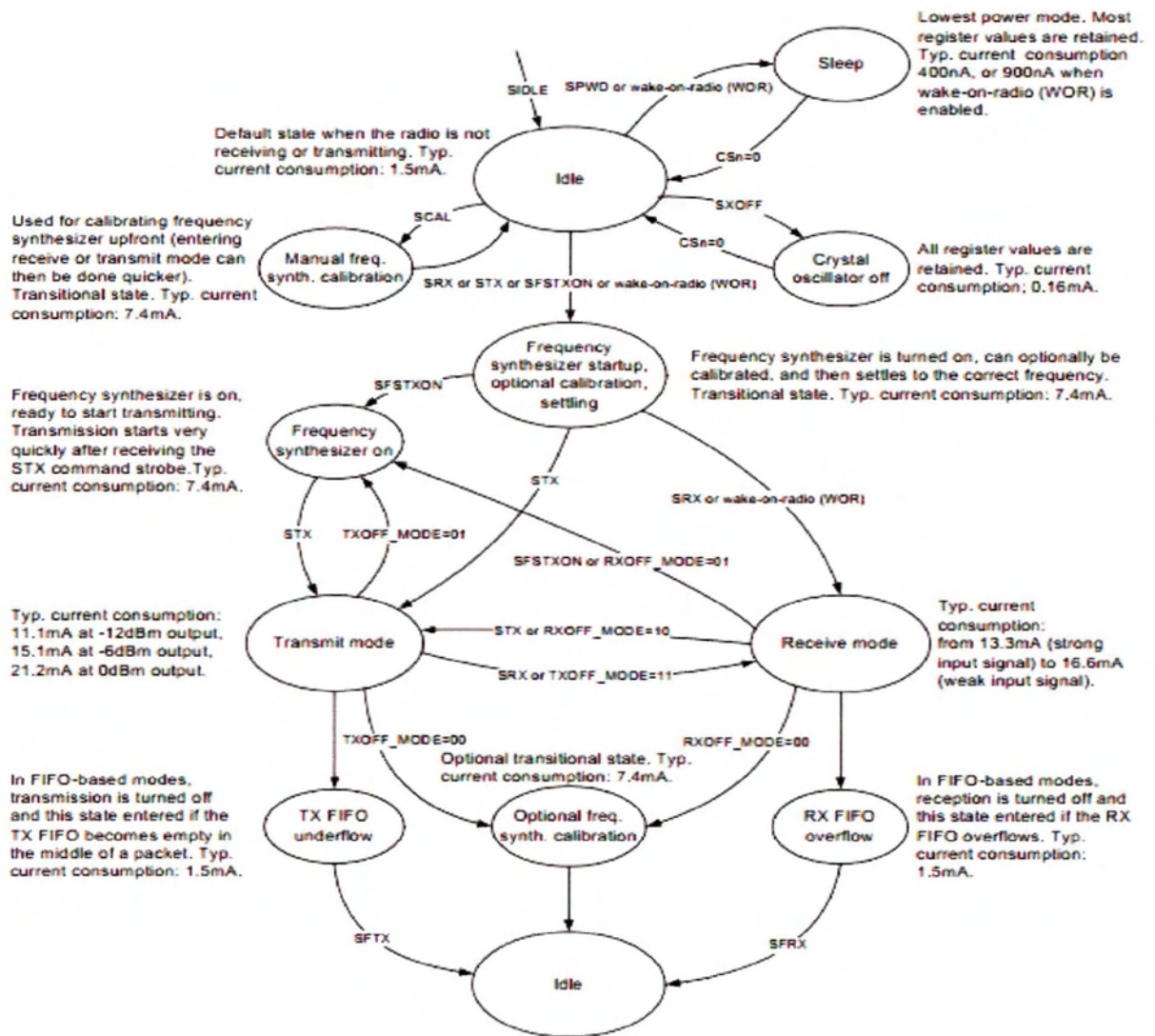
Then, when SPWD or WOR command strobes are issued, the chip goes in SLEEP mode automatically. Of course, it is necessary the SS pin is HIGH.

Finally, when SS goes LOW again, the chip enters in IDLE state again. After that, CC2500 is ready to go in TX or RX state. In addition, CC2500 can exit WOR mode if we send an IDLE command strobe.

Also, there is something crucial about the Wake on Radio. The RC oscillator must be enabled before the WOR strobe can be used, as it is the clock source for the WOR timer. The on-chip timer will set CC2500 into the IDLE state and then the RX state. After a programmable time in RX, the chip goes back to the SLEEP state, unless a packet is received.

RC oscillator: The frequency of the low-power RC oscillator used for the WOR functionality varies with temperature and supply voltage. In order to keep the frequency as accurate as possible, the RC oscillator will be calibrated whenever possible, which is when the XOSC is running and the chip is not in the SLEEP state.

Here's a flowchart of current consumption of all states of CC2500:



9.2. Different TX power

As mentioned before, CC2500 has different current consumption, depending on which state it is.

Except from that, CC2500 has the ability to transmit data with different tx power every time.

It is a simple procedure, as to select power control settings, we just have to change value of PATABLE [22] register.

More specific, the 0x3E address is used to access the PATABLE, which is used for selecting PA power control settings. The PATABLE is an 8-byte table, but not all entries into this table are used. The entries to use are selected by the 3-bit value FRENDO.PA_POWER.

Table with TX power, respective PATABLE values and current consumption.

Tx power (dBm)	PATABLE value	Current Consumption(mA)
-55	0x00	8.5
-30	0x50	9.9
-28	0x44	9.7
-26	0xC0	10.2
-24	0x84	10.1
-22	0x81	10
-20	0x46	10.1
-16	0x55	10.8
-10	0x97	12.2
-8	0x6E	14.1
-4	0xA9	16.2
-2	0xBB	17.7
0	0xFE	21.2
1	0xFF	21.5

9.3. LQI and RSSI

9.3.1. General for LQI and RSSI

LQI and RSSI are often used as measures for the wireless link quality.

LQI estimates how easily a received packet can be demodulated. It reflected the bit error rate of the connection.

RSSI estimates the signal level in the chosen channel. This metric provides a measure of the signal strength at the receiver.

The RSSI value is measured in dBm and expresses the signal power. The values typically range from -45dBm to -100dBm. The lower value is determined by the receiver input threshold, and the upper by the airborne signal strength.

The LQI value reflects the link quality seen from the receiver side. The LQI value correlates with the Packet Reception Rate (PRR) and is therefore a very important figure in mesh routing protocols. The value combines the RSSI value with a correlation of the expected and received data, thus being able to reflect a bad link quality in a noisy environment that results in a high RSSI value. There is no exact formula for how to calculate the LQI value and the estimation method is implementation specific. The LQI value ranges between 0 and 255, where the highest value represents the maximum quality frames.

We can easily access these metrics:

- By reading the RX FIFO. LQI and RSSI are automatically added to the last and first byte appended after the data payload. If PKTCTRL1.APPEND_STATUS is enabled, two status bytes will be appended to the payload of the packet. The status bytes contain RSSI and LQI values, as well as the CRC OK flag. So, when receiving a correct packet, if we read the whole RX FIFO, we'll see the values of LQI and RSSI, after the packet payload.
- By reading the values of LQI and RSSI registers (read-only registers).

0x33 (0xF3): LQI – Demodulator Estimate for Link Quality

Bit	Field Name	Reset	R/W	Description
7	CRC_OK		R	The last CRC comparison matched. Cleared when entering/restarting RX mode. Only valid if PKTCTRL0.CC2400_EN=1.
6:0	LQI_EST[6:0]		R	The Link Quality Indicator estimates how easily a received signal can be demodulated. Calculated over the 64 symbols following the sync word.

0x34 (0xF4): RSSI – Received Signal Strength Indication

Bit	Field Name	Reset	R/W	Description
7:0	RSSI		R	Received signal strength indicator

9.3.2. How RSSI and LQI calculated

In Transmit mode:

The modulator will first send the programmed number of preamble bytes. If data is available in the TX FIFO, the modulator will send the two-byte (optionally 4-byte) sync word and then the payload in the TX FIFO. If CRC is enabled, the checksum is calculated over all the data pulled from the TX FIFO and the result is sent as two extra bytes following the payload data. These two extra bytes contain the values of RSSI and LQI.

In Receive mode:

If automatic CRC check is enabled, the packet handler computes CRC and matches it with the appended CRC checksum. At the end of the payload, the packet handler will optionally write two extra packet status bytes that contain CRC status, LQI and RSSI value.

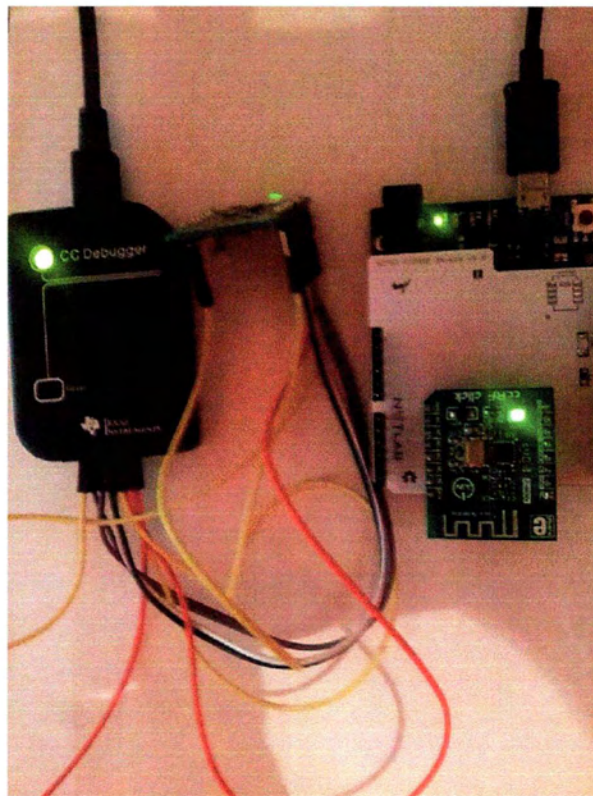
9.3.2. RSSI Diagrams

9.3.2.1. General

It's useful to see how RSSI value changes as time goes by in a communication. For this purpose, we've made some diagrams for RSSI, depending on time.

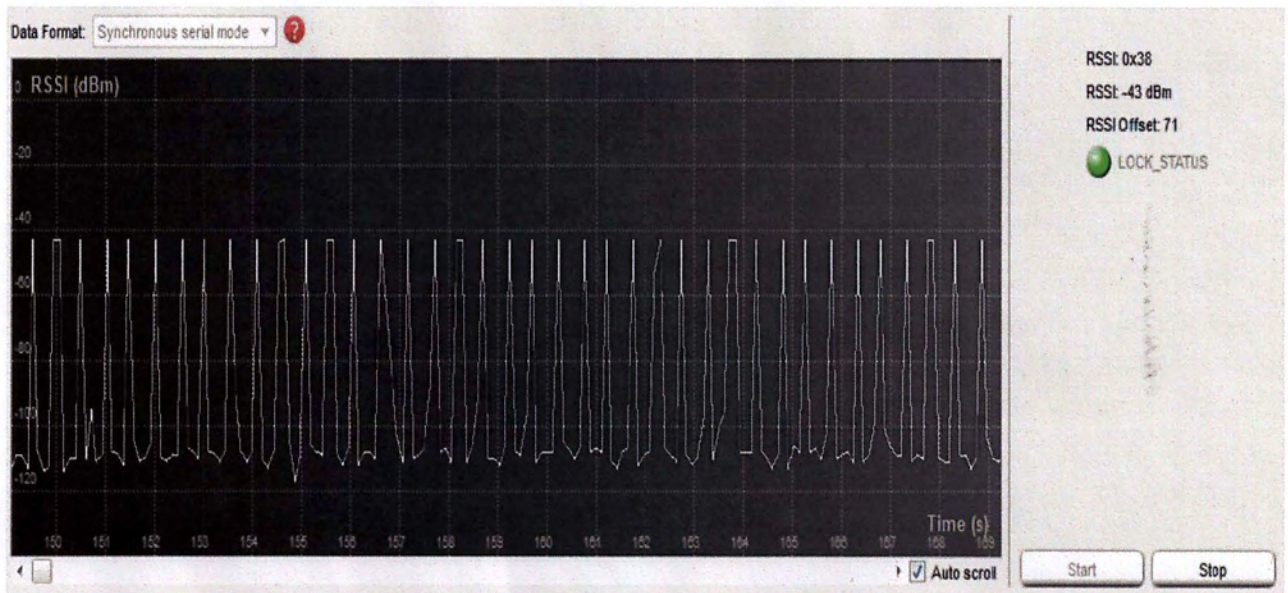
We've made these measurements with SmartRF, a software also designed by Texas Instruments. We connected with a CC Debugger a CC2500 chip, and opened the SmartRF program. Then, we started to transmit from another Arduino+CC2500. From SmartRF, we went on Continuous RX mode, so the CC2500, which was connected there, started to receive. We changed the registers of both connections in order the combination worked right. (We'll talk about CC Debugger and SmartRF, in more detail, later.)

The setup we made is in the photo below:



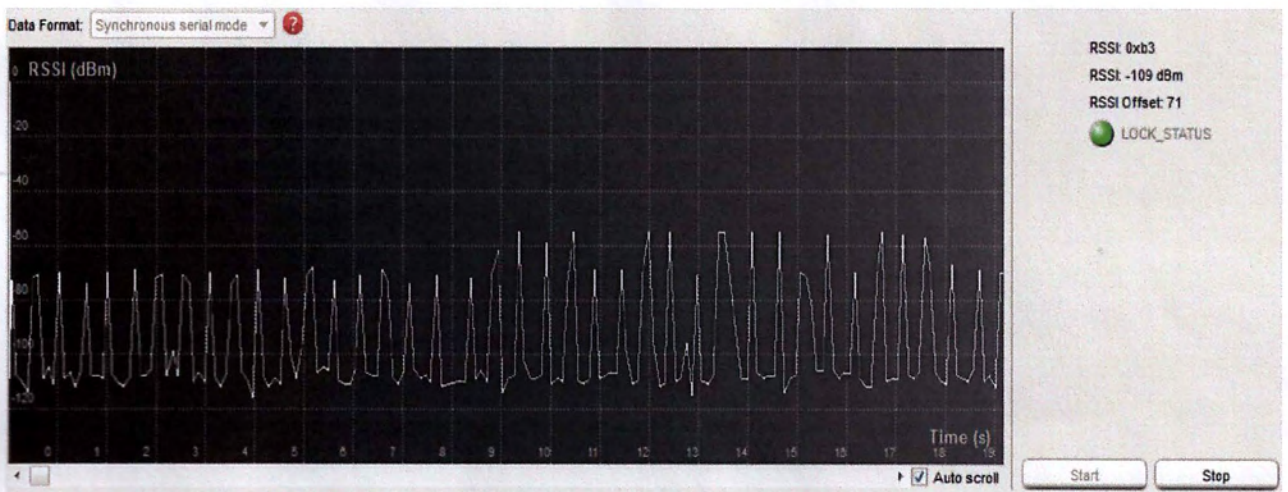
9.3.2.2. RSSI diagrams with different channels

In the first diagram, we chose channel 0 (frequency = 2432.999908 MHz) and we kept the two set-ups in the same distance:



RSSI signal remains stable for every packet that the CC2500 receives.

In the second diagram, we changed the channel from the one connection to 1 (frequency = 2433.199859 MHz):

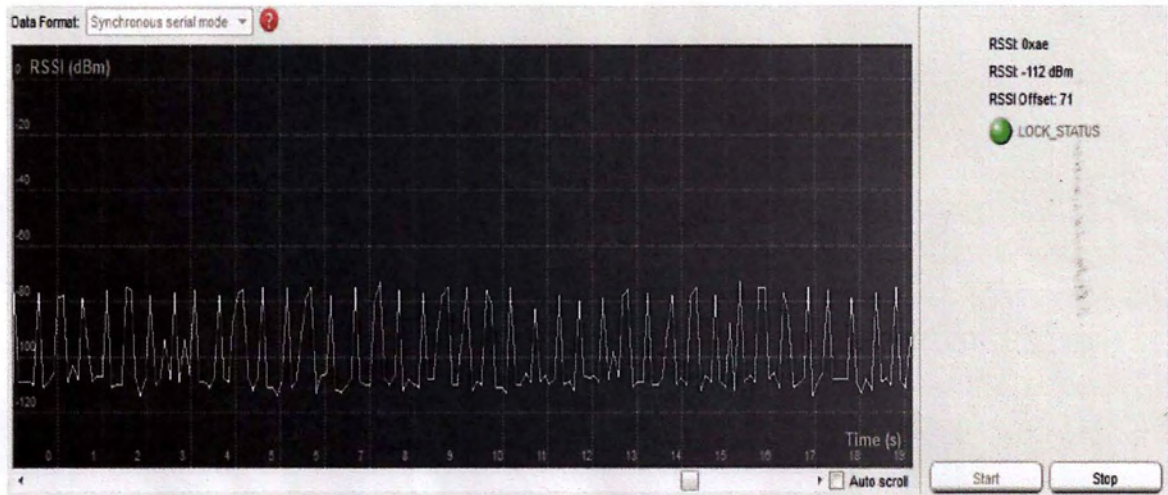


RSSI signal is obviously lower. That make sense due to the frequency we've changed.

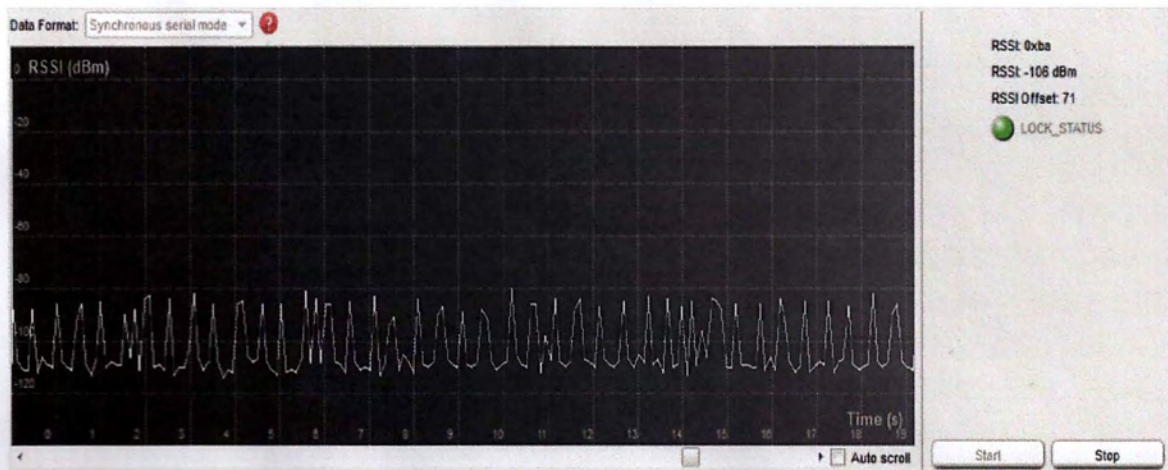
Then, we increased the channel number of the one connection, consecutively. The other connection has a stable channel number equal to zero (frequency = 2432.999908 MHz).

The diagrams we've recorded are below.

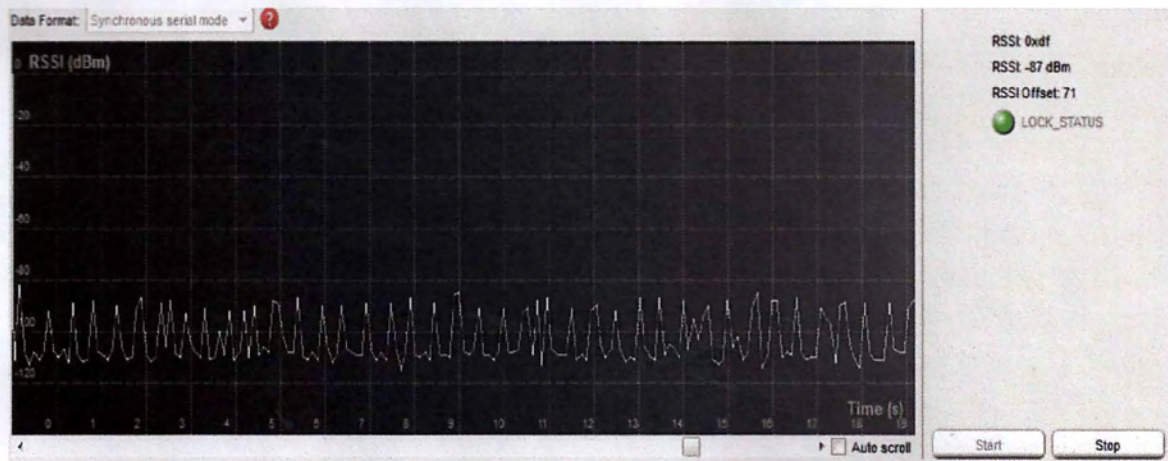
For channel= 2 (frequency = 2433.399810 MHz):



For channel = 3 (frequency = 2433.599761 MHz):



For channel = 4 (frequency = 2433.799712 MHz):

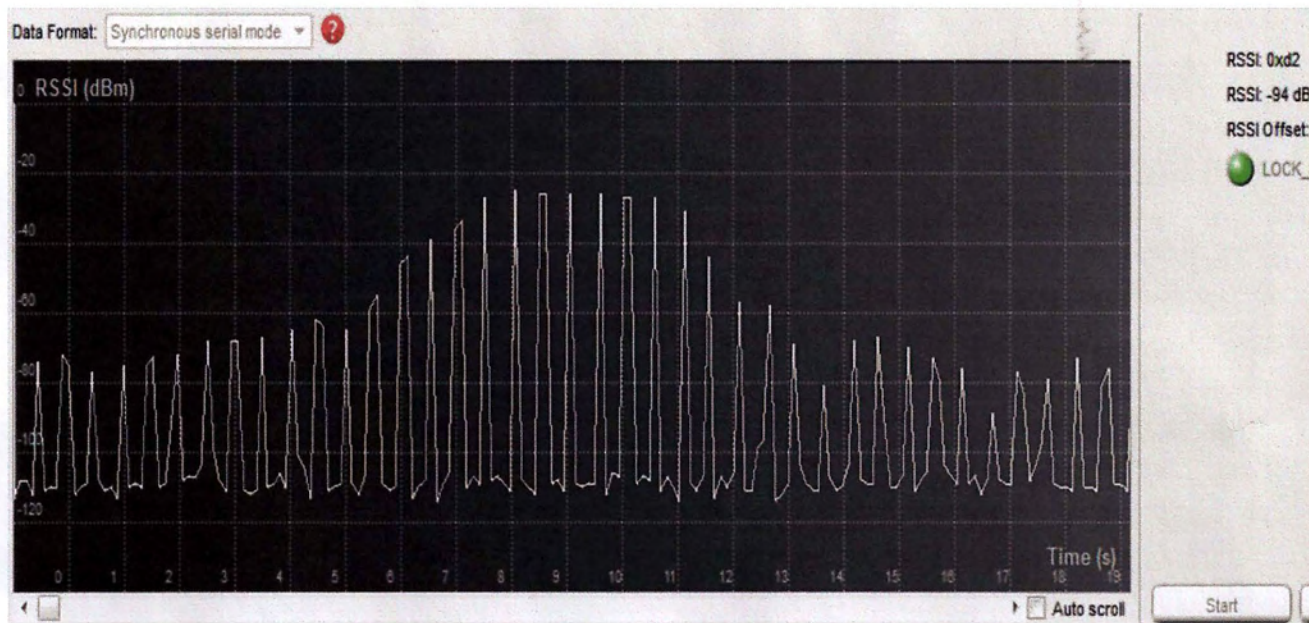


From channel 5 and above, RSSI signal starts to reduce significantly.

9.3.2.3. RSSI diagrams with different distances

RSSI is a strength signal value, and as a result, it's logical to depend from the distance that we try to communicate.

So, we've created a diagram to see how RSSI affected, when the transmitter changes positions continuously. The diagram is shown below:



We've started the test, holding the one node in about 1 meter away from the other. And periodically, we started to get closer. At about 10s the two CC2500 was almost attached to each other. That's why, we notice how strong the signal is. Then, as seconds gone by, we started periodically to get the nodes away from each other. About 2 meters away at 20th second.

9.4. Clear Channel Assessment (CCA)

The Clear Channel Assessment (CCA) is used to indicate if the current channel is free or busy. The current CCA state is viewable on any of the GDO pins by setting `IOCFGx.GDOx_CFG=0x09`.

There are four modes that CCA can be programmed. `MCSM1.CCA_MODE` selects the mode to use when determining CCA. We only entered TX mode if the clear channel requirements are fulfilled. The chip will otherwise remain in RX. The four CCA requirements that can be programmed:

- Always (CCA disabled, always goes in TX).
- If RSSI is below a threshold.
- Unless currently not receiving a packet.

- Both the above (RSSI below threshold and not currently receiving a packet).

With the MCSM1 register, we can choose which mode we 'd like for our communication, by changing the value of MCSM1.CCA_MODE to 0x00, 0x01, 0x10 and 0x11, respectively for each mode.

9.5. Burst Transmission

The high maximum data rate of CC2500 opens up for burst transmissions. A low average data rate link (e.g. 10 kBaud), can be realized using a higher over-the-air data rate. Buffering the data and transmitting in bursts at high data rate (e.g. 500 kBaud) will reduce the time in active mode, and hence also reduce the average current consumption significantly. In addition, reducing the time in active mode will reduce the likelihood of collisions with other systems.

9.6. Frequency hopping

The 2.400 – 2.4835 GHz band is shared by many systems both in industrial, office and home environments. It is therefore recommended to use frequency hopping spread spectrum (FHSS) or a multi-channel protocol because the frequency diversity makes the system more robust with respect to interference from other systems operating in the same frequency band. CC2500 is highly suited for FHSS or multichannel systems due to its agile frequency synthesizer and effective communication interface.

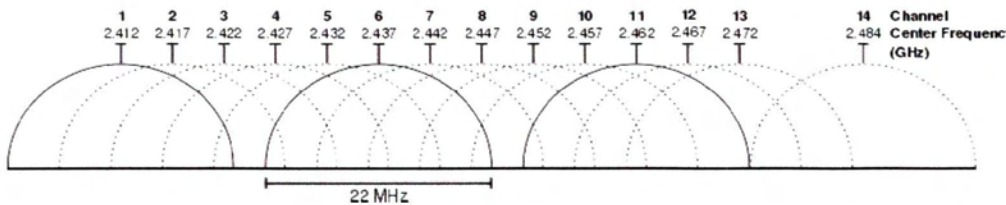
9.7. Data whitening

Real world data often contain long sequences of zeros and ones. Performance can then be improved by whitening the data before transmitting, and de-whitening the data in the receiver. With CC2500, this can be done automatically by setting PKTCTRL0.WHITE_DATA=1. All data, except the preamble and the sync word, are then XOR-ed with a 9-bit pseudo-random (PN9) sequence before being transmitted. At the receiver, the data are XOR-ed with the same pseudo random sequence. This way, the whitening is reversed, and the original data appear in the receiver. Data whitening can only be used when PKTCTRL0.CC2400_EN=0.

10. Frequency

As mention, CC2500 frequency range is from 2400 to 2483.5 MHz. That means that we'll probably meet some "noise" from other wireless devices, which uses the same frequency band.

So, we made an experiment in order to prove that. We set our router, at home, in channel 11, according to this wifi channel diagram [23]:



In this way, our router works on 2462 MHz, which is a frequency into CC2500 frequency range. More specifically, CC2500 has frequency equal to 2462 MHz, when its channel is 145.

Then, we tried to communicate with two CC2500 transceivers. By selecting channel 0 to channel 131 (2459.1 MHz), we noticed that throughput has common values as usual. So, there's nothing special to mention so far, because we're far from the channel that the router worked.

Then, at channel 132 (2459.4 MHz), communication started to have big losses. Specifically, only 40 packets of 140, delivered right in a minute.

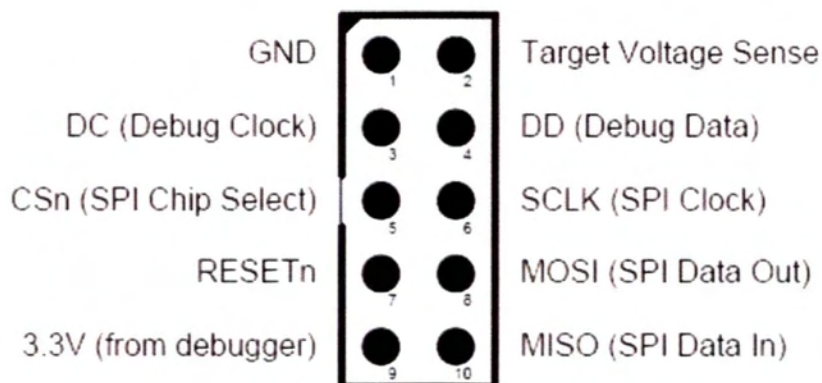
From channel 133 and above, nothing delivered correctly to the receiver. That's explains what we've said in the beginning of our experiment. CC2500 may meet some serious issues if the communication is been doing over other wireless devices. In a case like this, frequency hopping or a multi-channel protocol may be really important.

11. CC Debugger

11.1. General

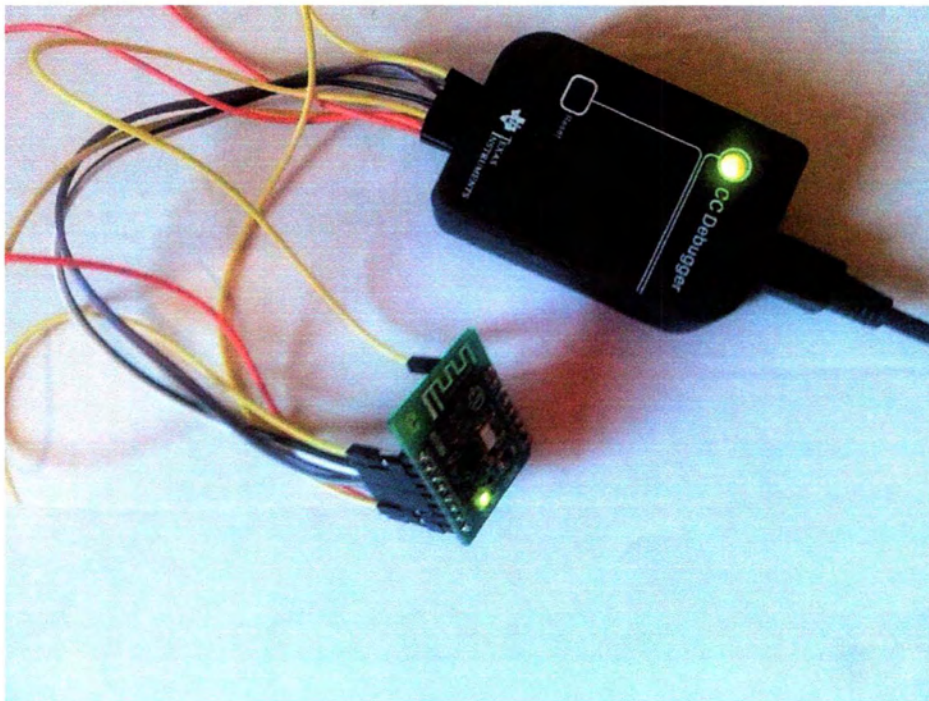
The CC Debugger [24] is a small programmer and debugger for the TI Low Power RF System-on-Chips, like CC2500.

CC2500 can be directly connected and programmed, with a CC Debugger. CC Debugger supports SPI connection, so it's really easy to connect it with a CC2500 transceiver.

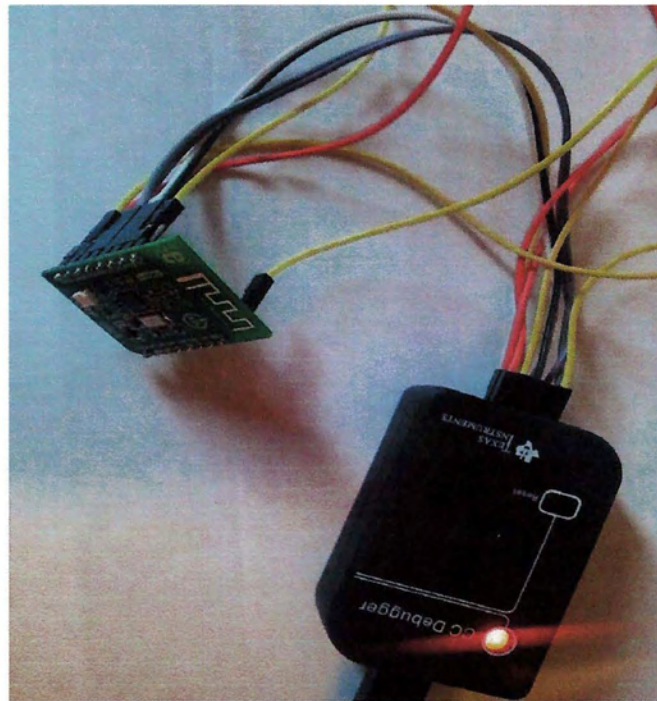


CC Debugger pins

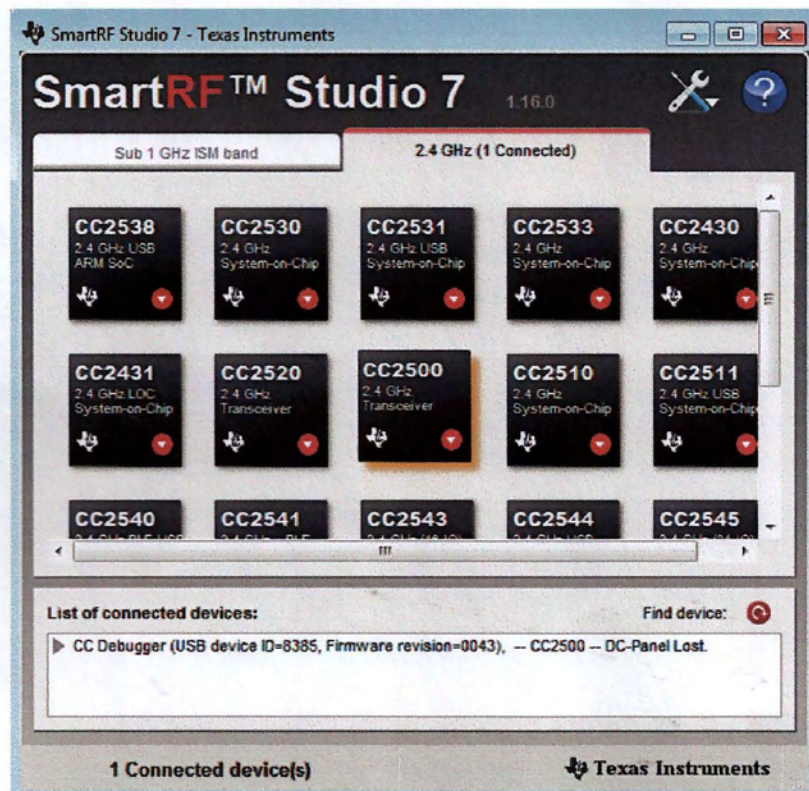
If everything in the connection with CC2500 goes well, a green light will light up.



Otherwise, a red light will show us, that something is wrong with our connection.



CC Debugger uses SmartRF [25] software, which gives us a lot of opportunities to have a better look at CC2500 performance. SmartRF can operate with 1 GHz and 2.4 GHz chips (like CC2500).



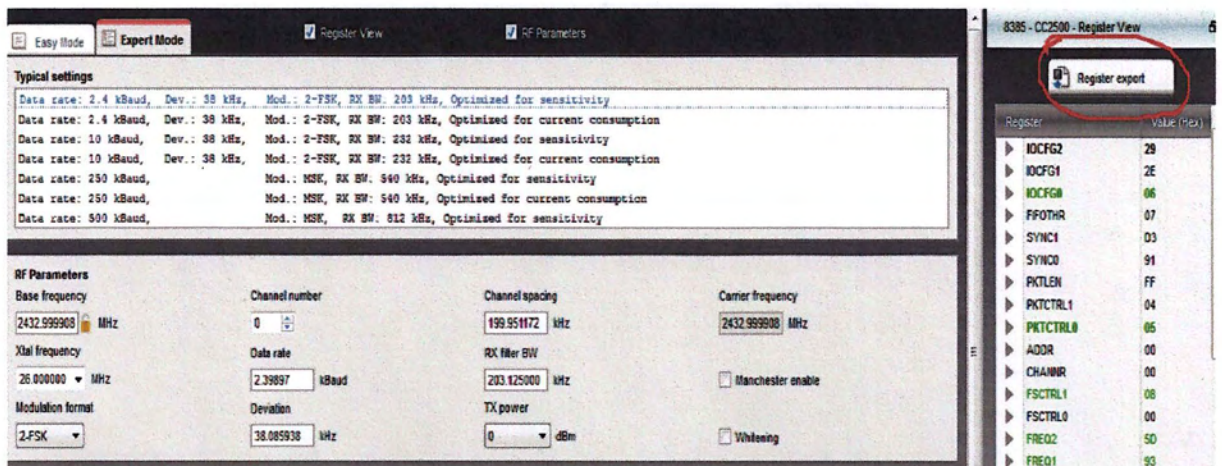
In the picture, is showed that a CC2500 chip is connected, via a CC Debugger

11.2. Features of SmartRF

SmartRF gives the potential, to a user, to have a better and more integrated view of CC2500 transceivers.

11.2.1. Generate header files

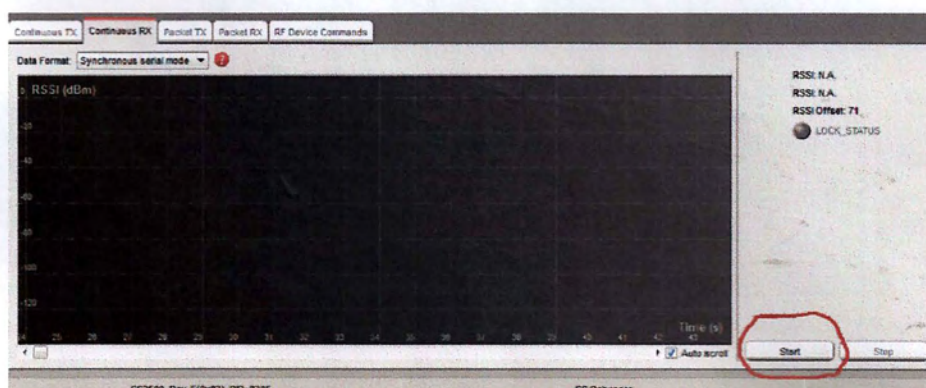
In SmartRF there are ready tables of registers and other buttons that help us state, what conditions we want for our communication. In that way, it is really easy for us to find the preferred registers values and then generate header files.



When change a parameter on the left board, registers values will be automatically affected to the right board, depending on what we've changed. Then, by clicking "Register export" you can create your header files and automatically saved in your computer.

11.2.2. See RSSI – time diagrams for a communication

When analysing what RSSI is before, we've seen different diagrams of RSSI, depending on time. This is extremely easy, using SmartRF. You just have to set one node (Arduino+CC2500) as transmitter, and the CC2500 which is connected to SmartRF, as receiver (Continuous RX). To start Continuous RX at SmartRF, you only need to push the button "start".

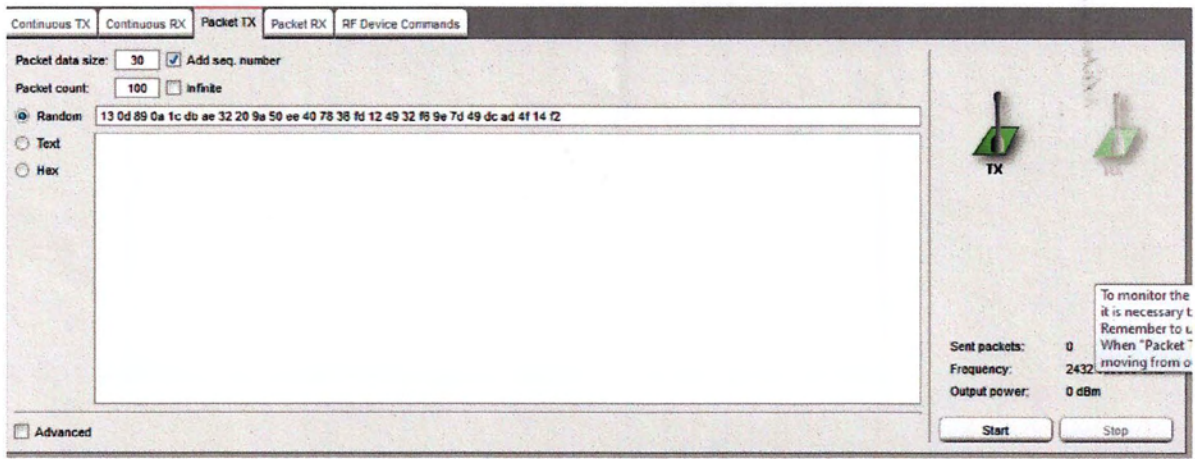


These diagrams may be really useful to evaluate the strength of the signal, depending on the frequency, the distance etc.

11.2.3. Compute packet error rate

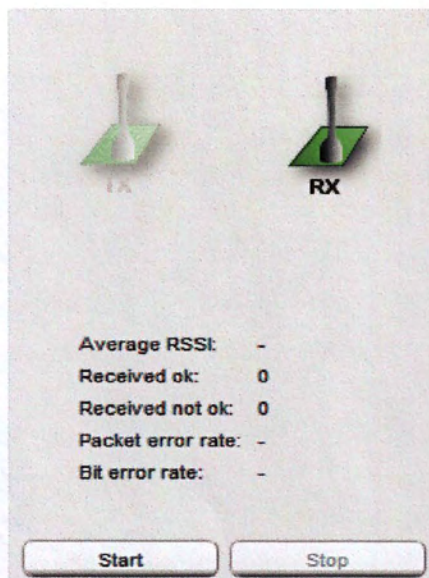
Ready processes “packet TX” and “Packet RX” are useful, too.

“Packet TX” gives user the opportunity to select packet payload he wants, and then transmit it.



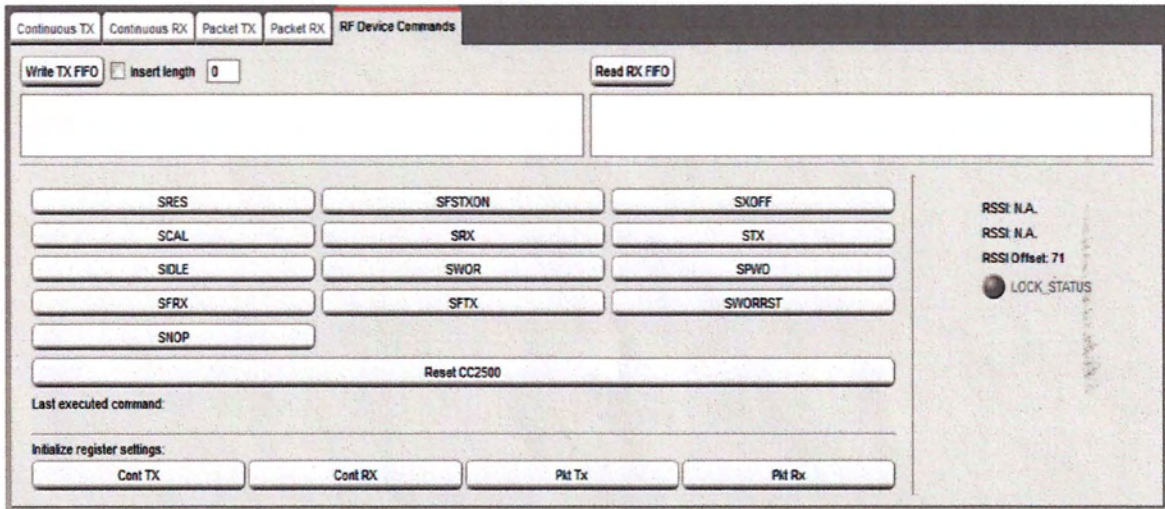
You can choose exactly what you want to transmit

“Packet RX” evaluates packet error rate, by calculating how many packets received OK and how many received NOT OK.



11.2.4. Give Device Commands

SendStrobe commands can be used to give device commands to CC2500. We can go in SLEEP state (SPWD), IDLE state (SIDLE), wake up from sleep and many others. All by just clicking a button in SmartRF software.



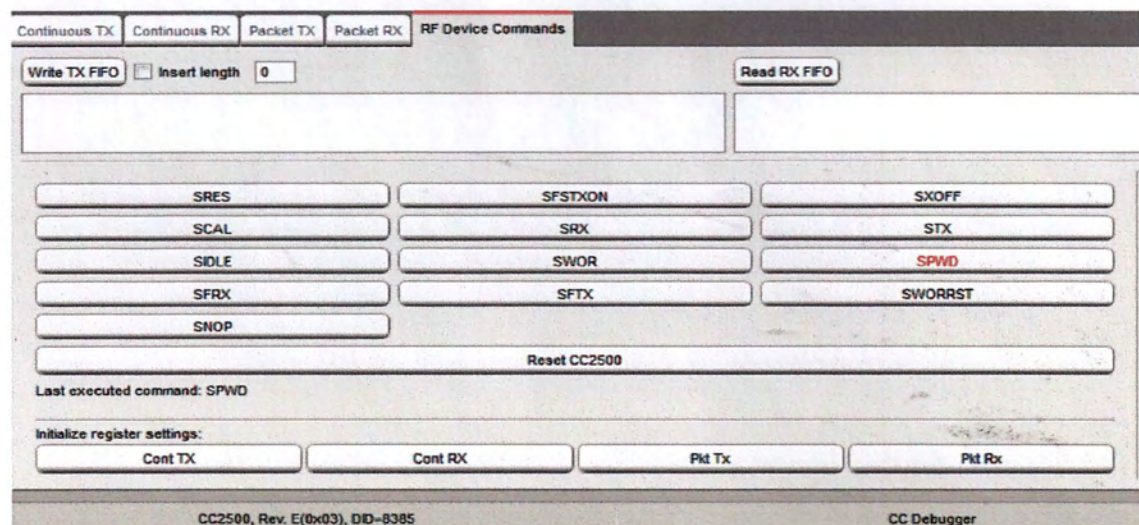
11.3. SmartRF example

We've mentioned before that CC2500 has the ability to go in SLEEP mode. Let's see how we could do that using SmartRF with a simple example.

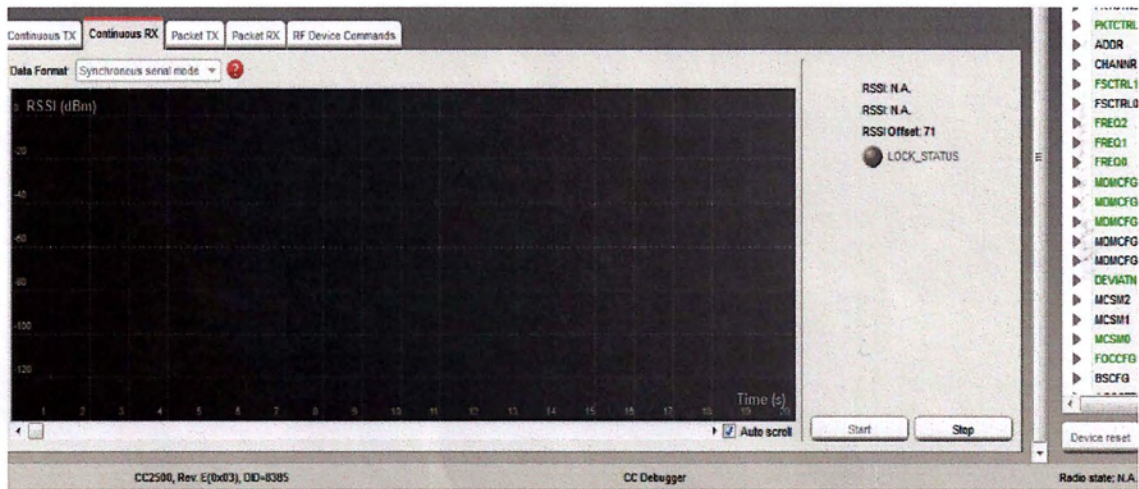
1. First, go in RX state, while one other CC2500+Arduino is transmitting.



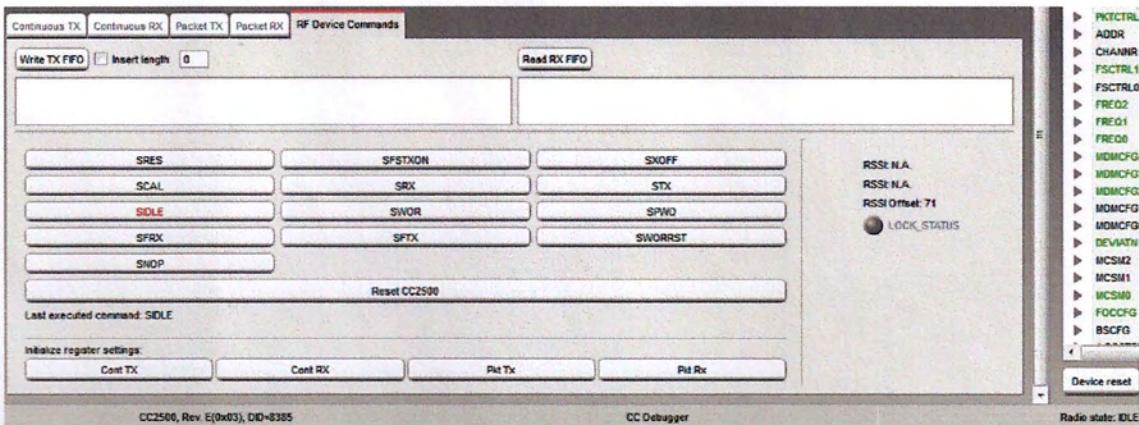
2. Then, send SPWD command and the chip goes in sleep mode.



3. Notice that nothing happened, even if we tried to go in RX state. CC2500 is in SLEEP mode.



4. Send SIDLE command to go in IDLE state.



5. Go in RX state again. Since we've left SLEEP mode, CC2500 can return to RX state.



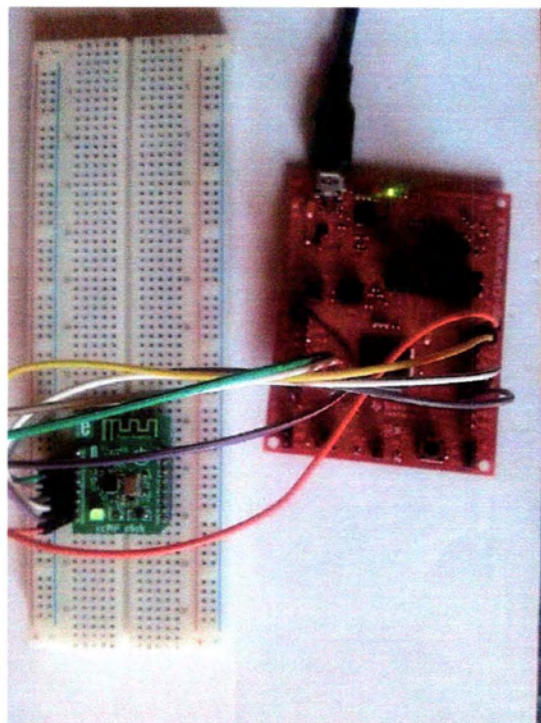
12. CC2500 with MSP430

12.1. General

CC2500 can also be connected with MSP430 [26] platforms. Specifically, we've connected CC2500 with MSP430 F5529 [27] and became acquainted with this board.

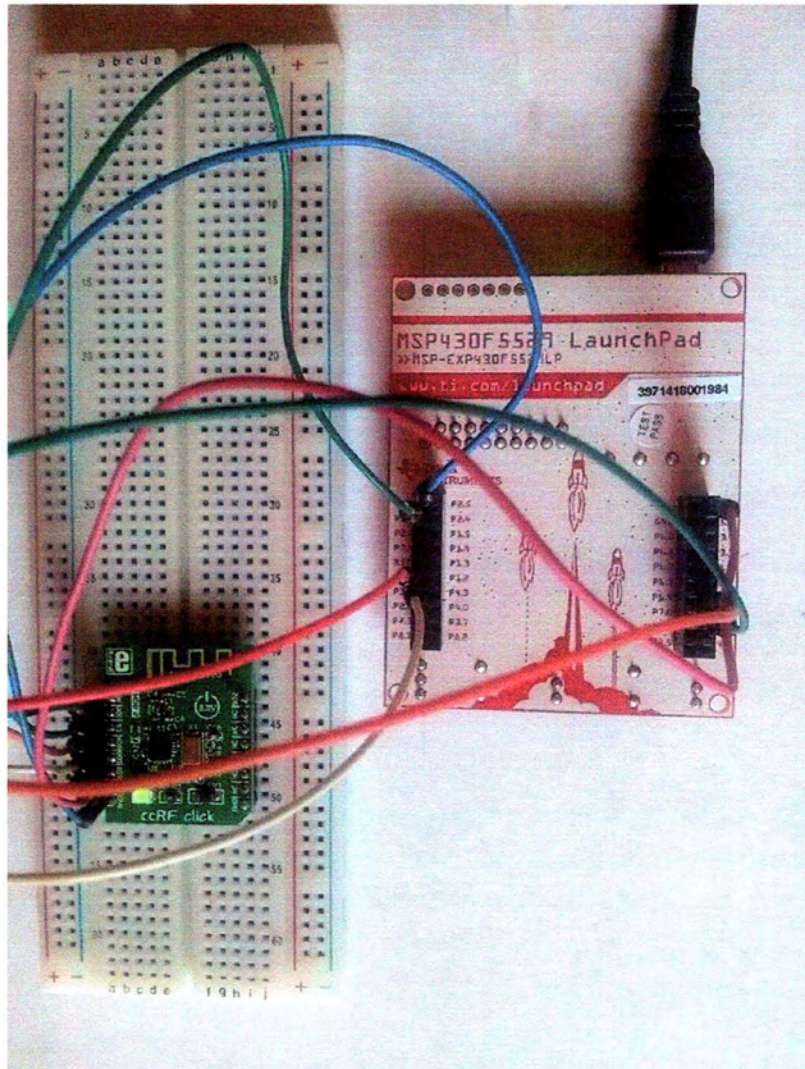


Configuration is done using the SPI interface. We connected CC2500 to MSP430, using a breadboard, like in the picture below.



MSP430 F5529 board is a two-side board, as it contains pins in both sides.

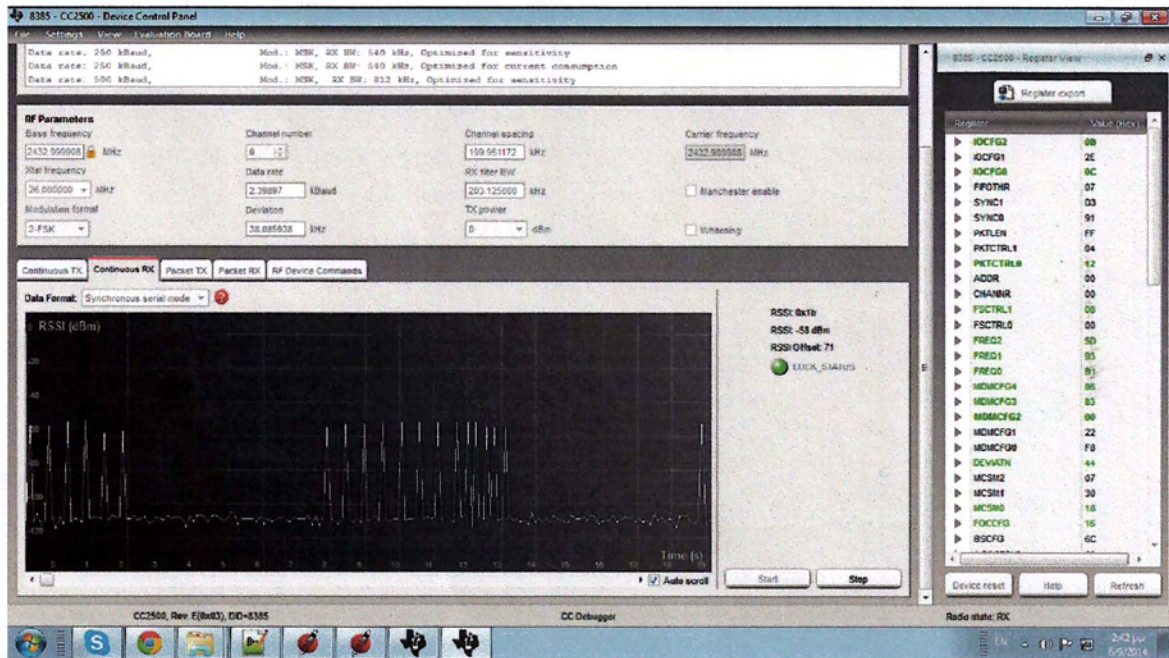
Here's the connection with CC2500 from the bottom side:



We used Energia [28] software to communicate with two CC2500 chips. Energia is an open-source electronics prototyping platform, with the goal to bring the Wiring and Arduino framework to the Texas Instruments MSP430 based LaunchPad.

12.2. MSP430 with SmartRF

We connected a CC2500 chip with CC Debugger and opened SmartRF program. We put this node into Continuous RX mode. In parallel, we put another node (MSP430+CC2500) in transmission mode. Here's the diagram that emerged:



As we see, signal strength is quite strong, as in the connection with Arduino.

In the time slots that signal appear to be attenuated, it's just because we've plugged out the usb cable of MSP430 (to show that the experiment is valid).

13. Future work

As mentioned in the beginning of this thesis, our main goal was to integrate CC2500 with NITOS prototype wireless sensor platform and configure this wireless interface to discover what benefits we will have by using this, instead of XBee.

After a lot of experiments, different configurations and observations, we were able to clearly understand these benefits of CC2500 and also to plan some future work about our research.

13.1. Current Consumption

We've seen that we can easily access PATABLE register, in order to transmit with different Tx power. Also, we've seen the amount of current consumption in different states, like IDLE state and SLEEP state. So, in the future, we could measure the actual current consumption in different cases and configurations, in order to design a CC2500's consumption pattern.

13.2. Wake-on Radio feature

CC2500 supports wake-on radio feature. In this thesis, we've seen how CC2500 can use this feature. In the future, we could exploit this CC2500's feature and implement a mechanism for this purpose. We can show how efficient this mechanism would be, by measuring the power consumption expenditure.

13.3. TinyOS library

TinyOS [29] is a free and open source software component-based operating system and platform targeting wireless sensor networks (WSNs). In the future, we could examine TinyOS library, and what this library, based on MSP430 platform, supports.

References

- [1] 802.15.4 stack wiki page: http://en.wikipedia.org/wiki/IEEE_802.15.4
- [2] Xbee wiki page: <http://en.wikipedia.org/wiki/XBee>
- [3] CC2500 Datasheet: <http://www.ti.com/lit/ds/symlink/cc2500.pdf>
- [4] ISM band wiki page: http://en.wikipedia.org/wiki/ISM_band
- [5] SRD band wiki page: http://en.wikipedia.org/wiki/Short_Range_Devices
- [6] Texas Instruments home page: <http://www.ti.com/>
- [7] RF module wiki page: http://en.wikipedia.org/wiki/RF_module
- [8] Burst transmission wiki page:
http://en.wikipedia.org/wiki/Burst_transmission
- [9] Frequency hopping wiki page: http://en.wikipedia.org/wiki/Frequency-hopping_spread_spectrum
- [10] Clear channel assessment paper:
<https://www.cse.sc.edu/files/reu/2007papers/EngstromPaper.pdf>
- [11] LQI and RSSI paper:
<https://pure.ltu.se/portal/files/44057485/ReportAmeasurementStudyOfPredictingThroughputFromLQIandRSSI.pdf>
- [12] Data whitening datasheet: <http://www.ti.com/lit/an/swra322/swra322.pdf>
- [13] Wake on Radio datasheet:
<http://www.ti.com/lit/an/swra126b/swra126b.pdf>
- [14] SPI interface datasheet: <http://www.ti.com/lit/an/swra112b/swra112b.pdf>
- [15] RSSI datasheet: <http://www.ti.com/lit/an/swra114d/swra114d.pdf>
- [16] Arduino home page: <http://arduino.cc/>
- [17] SPI wiki page:
http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
- [18] NITLab home page: <http://nitlab.inf.uth.gr/NITlab/>
- [19] Our first library: <https://github.com/yasiralijaved/Arduino-CC2500-Library>
- [20] Our second library: <https://github.com/Zohan/ArduinoCC2500Demo>
- [21] Throughput wiki page: <http://en.wikipedia.org/wiki/Throughput>
- [22] PATABLE access datasheet:
<http://www.ti.com/lit/an/swra110b/swra110b.pdf>
- [23] WLAN channels wiki page:
http://en.wikipedia.org/wiki/List_of_WLAN_channels
- [24] CC Debugger user's guide:
<http://www.ti.com/lit/ug/swru197h/swru197h.pdf>
- [25] SmartRF home page: <http://www.ti.com/tool/smartrfm-studio>
- [26] MSP430 wiki page: http://en.wikipedia.org/wiki/TI_MSP430
- [27] MSP430 F5529 user's guide:
<http://www.ti.com/lit/ug/slau533b/slau533b.pdf>
- [28] Energia home page: <http://energia.nu/>
- [29] TinyOS wiki page: <http://en.wikipedia.org/wiki/TinyOS>



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΒΙΒΛΙΟΘΗΚΗ



004000124468