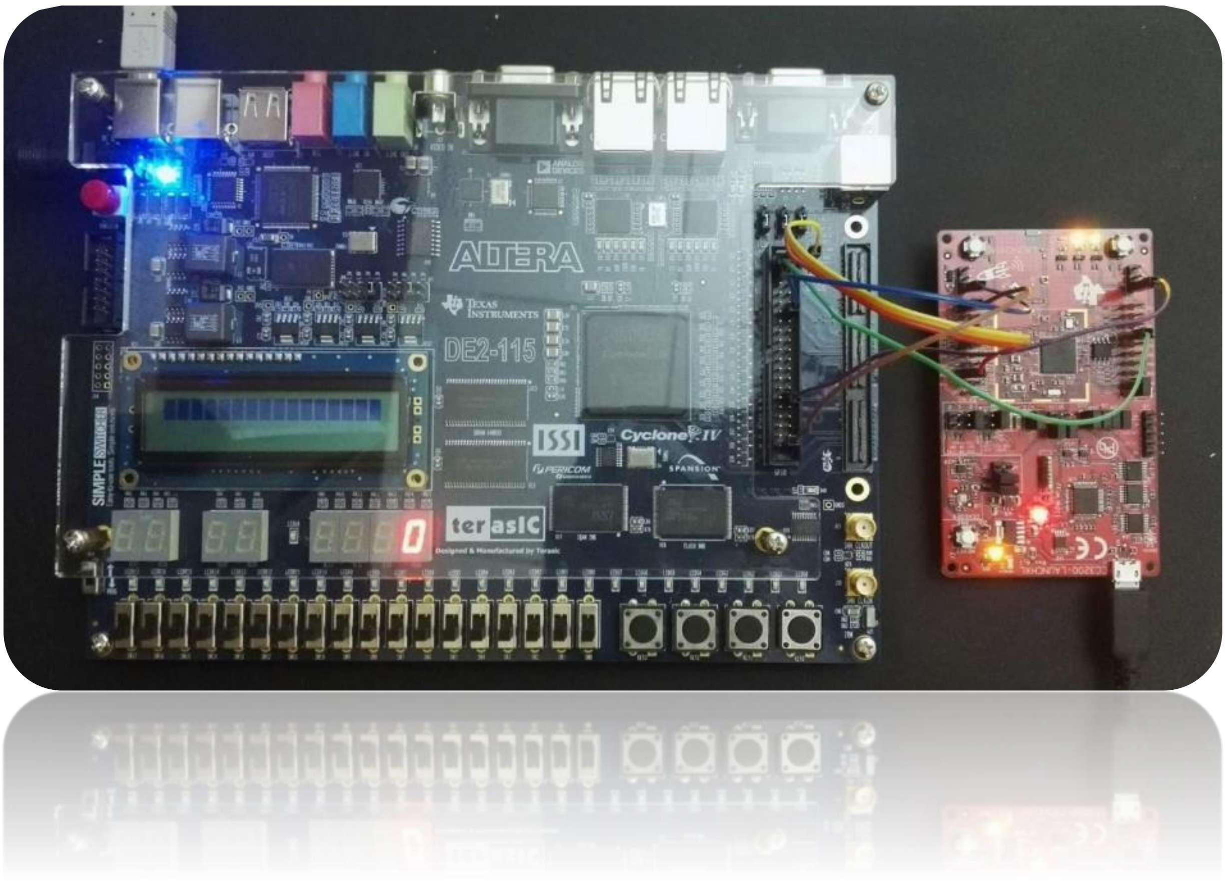
FPGA/SOC Wireless Data Transmission

User Manual



Contents

Project Overview

Context

This application allows for transmitting data from a FPGA to a PC or any TCP/IP capable device through WiFi via a wireless enabled MCU (CC3200 from TI), at a rate up to 8 Mbits/sec.

A FPGA receive data from several sensors, store them in memory and transmit them to a Microcontroller (MCU) through a SPI connexion. The MCU first connect itself to an access point (WiFi) and then to a receiving server (in our case a python program). Afterward, the MCU starts accepting the SPI data sent from the FPGA and store them in a Buffer. Once the buffer is full enough, the MCU send data to a computer through the TCP/IP protocol. Meanwhile, it still receives data from the SPI. These parallel processes (SPI reception, TCP transmission) repeat endlessly as long as the (socket) connexion between the MCU and the PC stays alive.

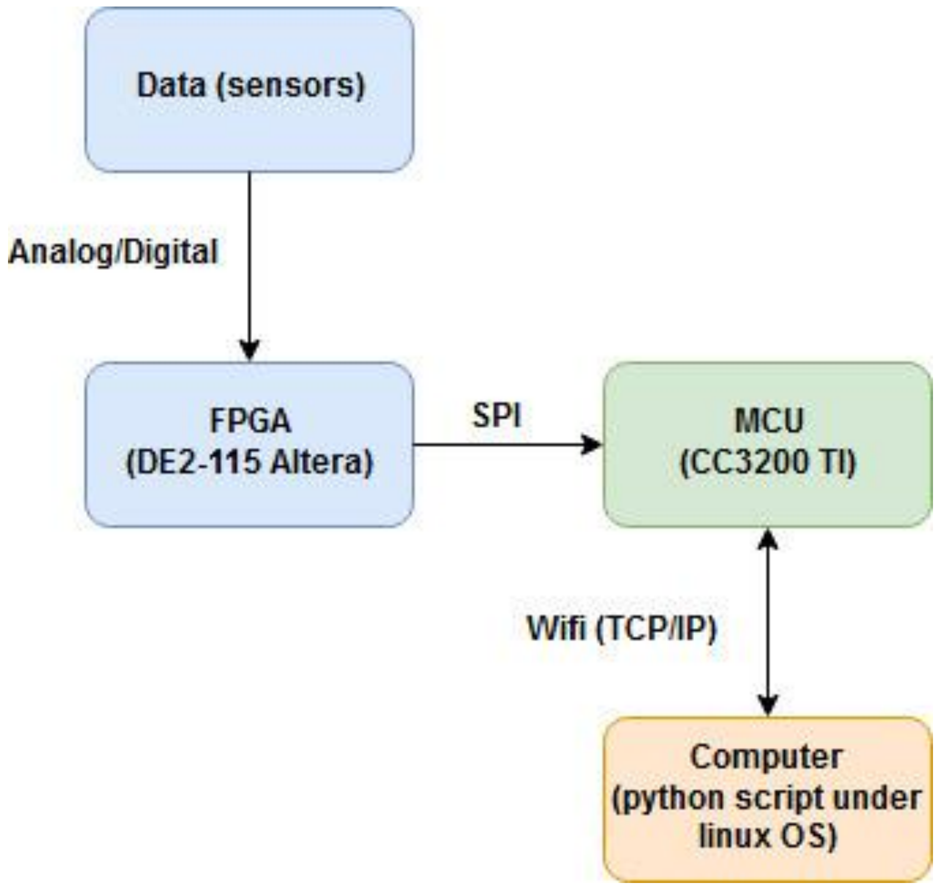


Figure 1 General Diagram of the system

Components and Software Used

OS

Windows (all tools are available on Linux too)

FPGA

Altera DE2-115

Quartus II v13.1 or later version

SOC

Ti CC3200 launchpad

Code Composer Studio, latest release

Debug

Putty

Logic analyser

PC Receiver

Linux (netcast)

Python 2.7

Useful documents

CC3200 Getting Started

Full Name: CC3200 SimpleLink Wi-Fi and IoT Solution With MCU LaunchPad Getting Started Guide User's Guide

CC3200 Launchpad User’s guide (hardware review)

CC3200 SimpleLink™ Wi-Fi® and IoT Solution with MCU LaunchPad Hardware User's Guide

CC3200 Reference Manuals (full doc about the chip peripheral)

CC3200 SimpleLink Wi-Fi and Internet-of Things Solution, a Single Chip Wireless MCU Technical Reference Manual

CC3200 SimpleLink User’s Guide

CC3100/CC3200 SimpleLink™ Wi-Fi® Internet on-a-Chip User's Guide

CC3200 Programmer’s guide

CC3200 SimpleLink ™ Wi-Fi ® and IoT Solution, a Single Chip Wireless MCU Programmer 's Guide

CC3200SDK documentation and examples (inside the SDK after download)

Uniflash V4 Quick guide (online resource)

CCS UniFlash v3.4.1 Release Notes (wiki)

System Details

FPGA side

The purpose of the configuration loaded in the FPGA of the DE2-115 development board from Altera is to acquire data and send them to the MCR through a SPI (Serial Peripheral Interface) bus. Data are acquired from sensors or in our case, from a module called “Timer” that regularly create data (under the form of a counter: data_0 = 32’d0, data_1 = 32’d1, data_2 = 32’d2, etc...). These data will be stored in memory in a SRAM FIFO (2 MB) configuration. As soon as 255 words (32 bits) of data are stored in the FIFO, the SPI module will start to output them by bulk of 255 words + 1 ending word, needed for SPI synchronization with the MCU SPI peripheral. A “block SPI” signal has been implemented to stop sending data to the MCU, in case the latter is slowed down in its Main program. The speed at which the data is created and set to the FIFO is set according the SW switches (SW0 to SW11) on the board (all switches OFF means max speed, All ON means slowest speed).

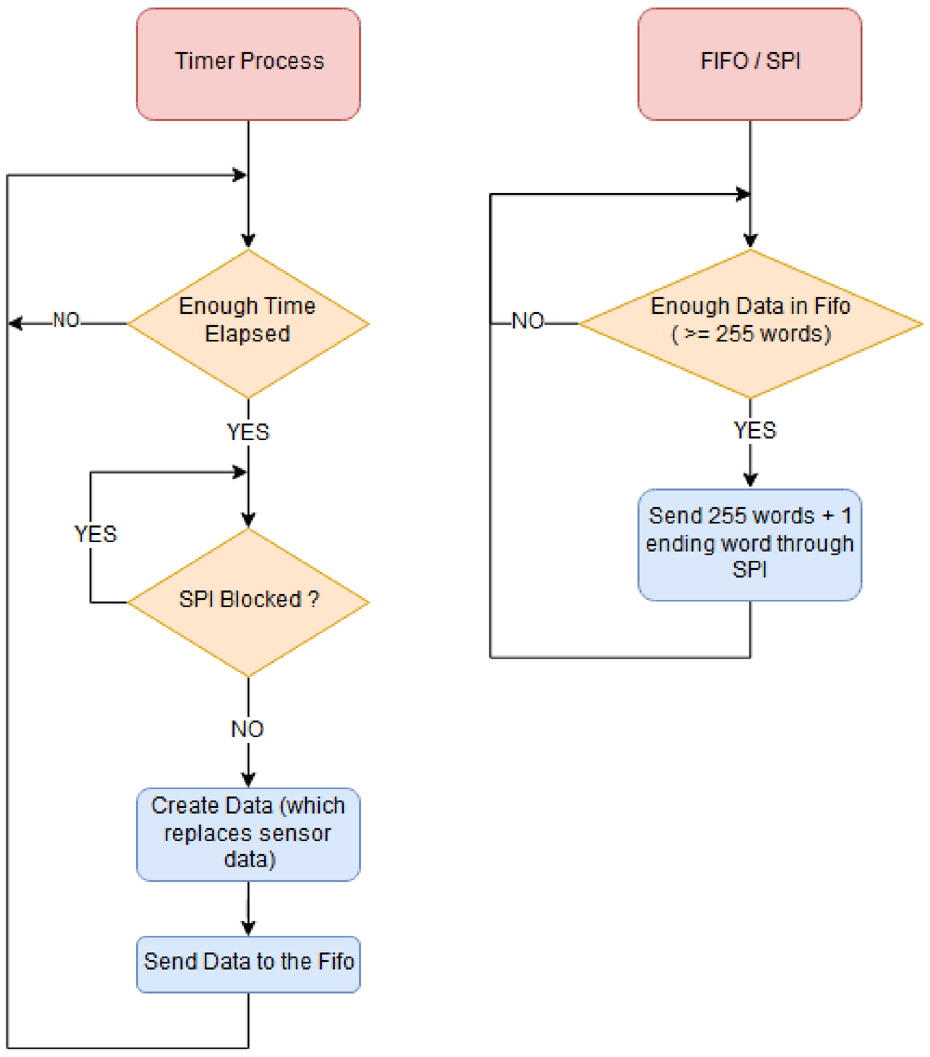

Timer Process
Enough Time Elapsed
NO
YES
YES
SPI Blocked ?
NO
Create Data (which replaces sensor data)
Send Data to the Fifo
FIFO / SPI
Enough Data in Fifo (>= 255 words)
NO
YES
Send 255 words + 1 ending word through SPI

Figure 2: FPGA Processes; Timer module; Fifo_spi module

MCU side

The purpose of the MCU program downloaded in the CC3200 wireless MCU for Texas Instrument (TI) is to receive the SPI data from the FPGA, store them, and send them wirelessly to a computer.

The MCU first connect to a WiFi Access Point (AP) according to the standard protocol 802.11b/g. Once this connexion established, the program will try endlessly to connect to our server which IP address is known.

Once a TCP connexion is established, it will start to accept SPI data. These data will be stored into a Ring Buffer each time a full SPI packet (255 words + 1 ending word) will be received. It is done thanks to an interrupt (function triggered by hardware/software event) in our case a "SPI transmission complete event" sent by the SPI peripheral.

When the Ring Buffer contains enough data, the main program will start to send them to the receiving computer with the TCP/IP socket protocol at a speed of 12 Mbits/sec. This is a data rate for direct sending (without processing). In our case, since between two sending, there is a little data processing (hence a performance decrease), the maximal data rate is around 8 Mbits/sec (depending on the network quality).

One crucial point is to elaborate: the moment when the TCP/IP connexion get slower, and even totally stopped for few milliseconds. Every once in a while depending on our network traffic, the transmission of data gets interrupted and no data can be sent for few milliseconds. This means that during that time, the MCU SPI peripheral would continue to receive data, and due to that network interrupt, it will be unable to process/send them. At that rate (8 Mbits/sec) it is impossible for us to store these received data in our Ring Buffer for more than few microseconds, due to the small memory of the MCU. That is why, when we will detect a network perturbation, we will send a signal to the FPGA to stop transmitting data until the network connexion come back to its normal state.

Finally a watchdog has been implemented. It a security that each time the program gets unexpectedly stuck (for instance at the connexion to a Access Point) the MCU will reset. The general idea is that throughout the program, we acknowledge the watchdog (with the call of a function). If at some point the watchdog did not get acknowledged for more than 5 seconds, the MCU will reset and restart its program from the beginning.

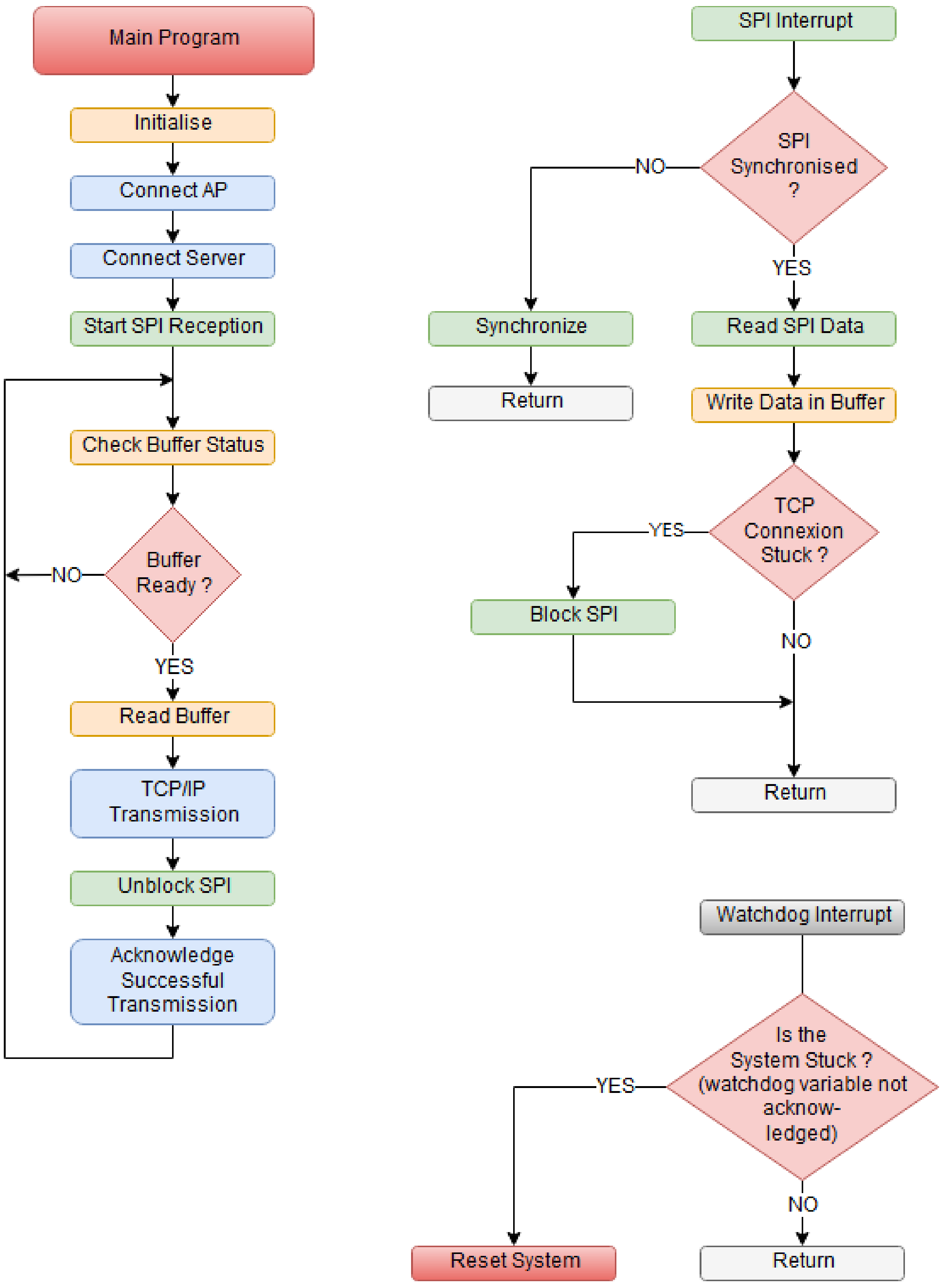


Figure 3: MCU Program Flow Chart; Main Program, SPI Interrupt, Watchdog Interrupt

Receiver side

The receiver side consists of several python scripts running on a Unix OS. The two most important ones are the one receiving data, and the one checking their integrity. The former will create a server, wait for connexions, start receiving a certain amount of TCP packets, save them into a file and finally close the connexion and display information about the connexion speed. The second program will "walk" through all data of the above file and check whether some expected data are missing by comparing the value of one data and the value of the previous one. That way, we can evaluate how many 32 bits words of data have been lost during the transmission.

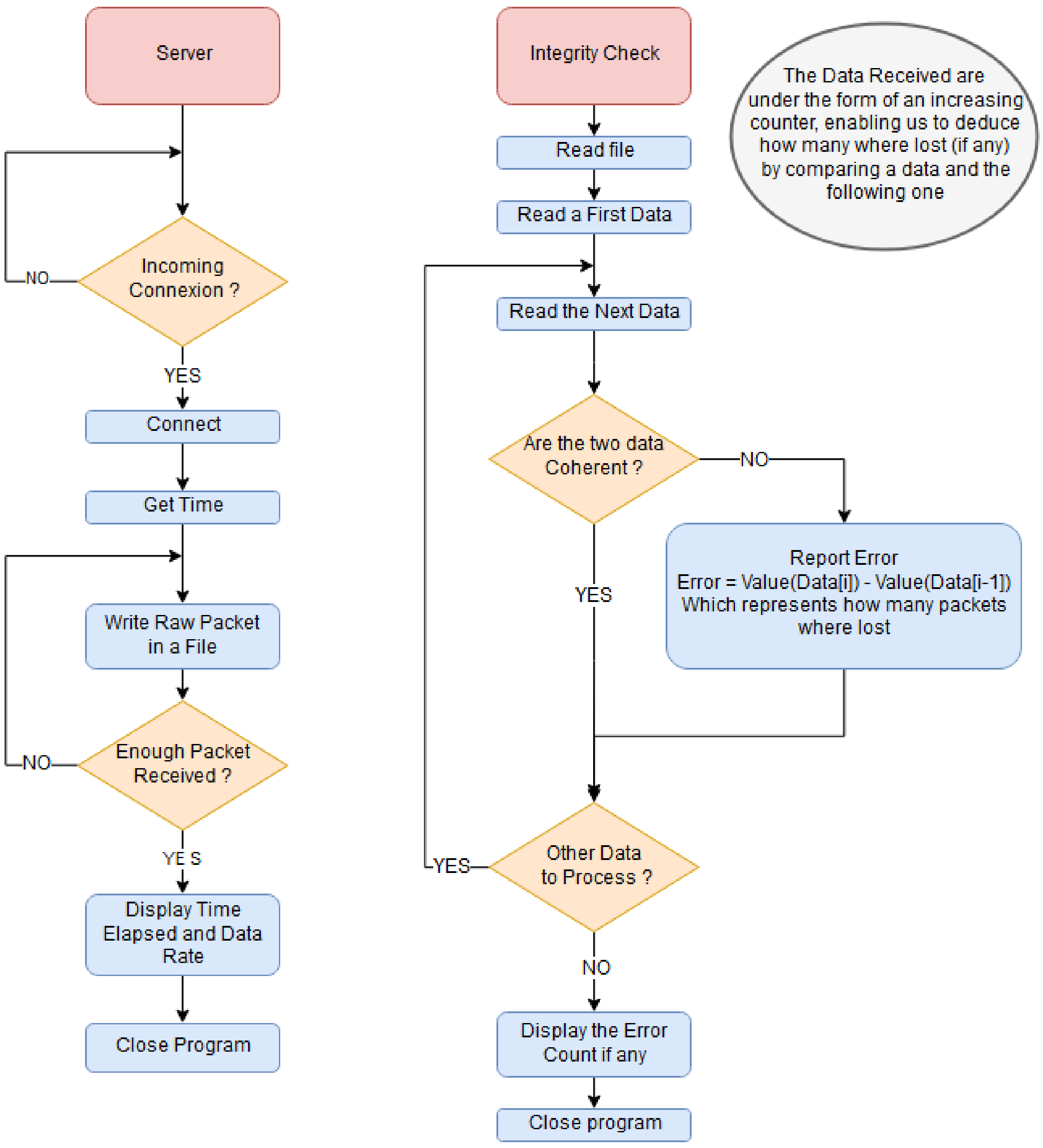

Server
Incoming Connexion ?
NO
YES
Connect
Get Time
Write Raw Packet in a File
Enough Packet Received ?
NO
YE S
Display Time Elapsed and Data Rate
Close Program
Integrity Check
Read file
Read a First Data
Read the Next Data
Are the two data Coherent ?
NO
YES
Report Error
Error = Value(Data[i]) - Value(Data[i-1])
Which represents how many packets where lost
Other Data to Process ?
YES
NO
Display the Error Count if any
Close program
The Data Received are under the form of an increasing counter, enabling us to deduce how many where lost (if any) by comparing a data and the following one

Figure 4: Receiver Flow Chart; Server; Data integrity test

Software installation

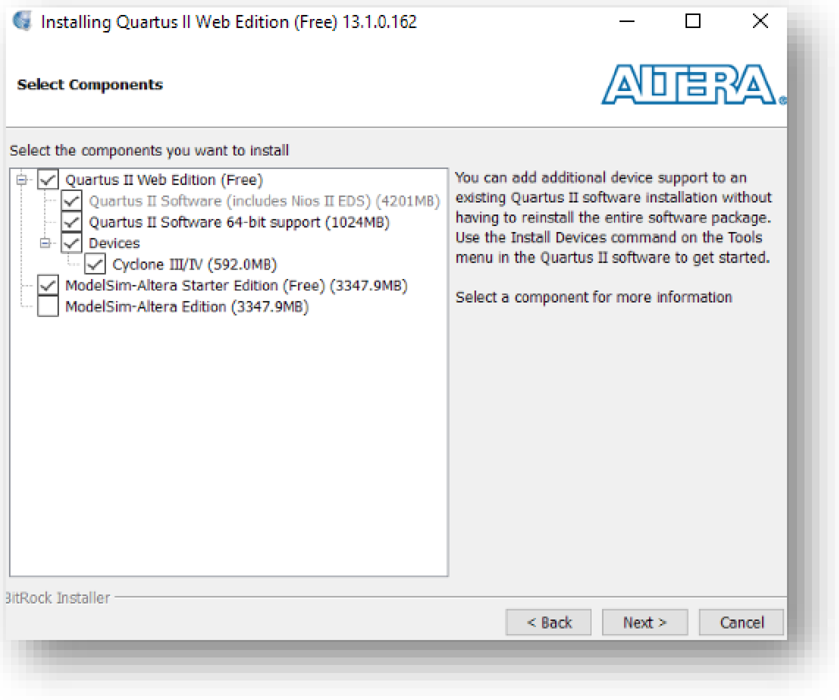
FPGA side

Quartus II:
Quartus II is the IDE used to develop the configuration (Verilog code) and program (write the Ram) the FPGA.

Quartus II web edition 13.1 Download (several method available)
http://dl.altera.com/13.1/?edition=web
The version 13.1 is used here, for in the newer versions, a delay of 10 seconds has been implemented (in the free version) for each "analysis and elaboration" (pre-compilation for simulation), which is slightly inconvenient. For the installation with the intention of using the DE2-115 board, the following software components (to select on altera website) are required:

Quartus II Web Edition (Free)
- Quartus II Software
- ModelSim-Altera Edition (optional)
- Cyclone III, Cyclone IV device support

The installation is straightforward, we will ensure to select the right components as per the picture thereafter, for Quartus 13.1.


Installing Quartus II Web Edition (Free) 13.1.0.162

Select Components

Select the components you want to install

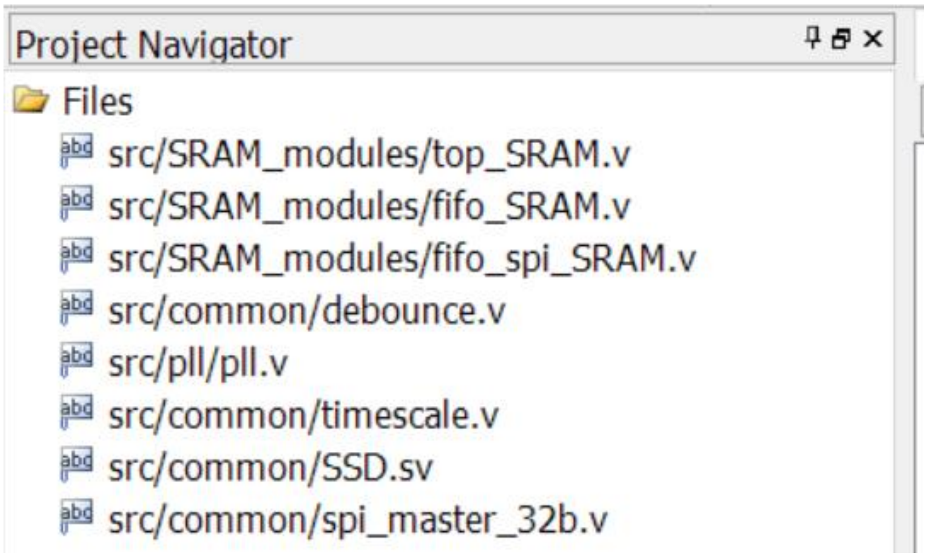
- [✓] Quartus II Web Edition (Free)
  - [✓] Quartus II Software (includes Nios II EDS) (4201MB)
  - [✓] Quartus II Software 64-bit support (1024MB)
  - [✓] Devices
    - [✓] Cyclone III/IV (592.0MB)
- [✓] ModelSim-Altera Starter Edition (Free) (3347.9MB)
- [ ] ModelSim-Altera Edition (3347.9MB)

You can add additional device support to an existing Quartus II software installation without having to reinstall the entire software package. Use the Install Devices command on the Tools menu in the Quartus II software to get started.

Select a component for more information

Figure 5: Quartus Installation step

After installation (following Altera's guidelines), open the Altera project in the folder Fpga_Soc_Interface/ (Fpga_Soc_Interface.qfp) with Quartus. On the file explorer, we will right click on top_SRAM.v and set as top level entity. Compile it Processing ➔ Start Compilation.



Project Navigator

Files
- src/SRAM_modules/top_SRAM.v
- src/SRAM_modules/fifo_SRAM.v
- src/SRAM_modules/fifo_spi_SRAM.v
- src/common/debounce.v
- src/pll/pll.v
- src/common/timescale.v
- src/common/SSD.sv
- src/common/spi_master_32b.v

Figure 6: Quartus project navigator

SOC (CC3200 LaunchPad) side:

Please refer to CC3200 Getting Started guide for deeper details:

Code Composer Studio (CCS):
CCS has been used for that project rather than the open sources tools (Make, GCC, GDB, and openOCD). The open source method has been tested. It worked for few examples and generally the ones that didn't involve WiFi transmission and RTOS. More complex projects revealed to have a very limited support and open source debug tools were very limited too, hence, unfortunately not very well adapted for that project.

CCS Download:
http://processors.wiki.ti.com/index.php/Download_CCS

The installation is straightforward, thus not detailed. The only detail is, at some point of the installation, we will be asked to select software components. The only one needed is the following:

- SimpleLink CC3xxx Wireless MCUs

Once CCS installed, we need to download the CC3200 SDK.
http://www.ti.com/tool/download/CC3200SDK
We will run the SDK-installer executable, which simply unpack the archives contained in it, to our selected TI repertory (preferably the same one as CCS's).
We launch CCS and select a workspace folder. That workspace will contain our project folders as it is the rule with eclipse based IDEs.

CCS Tips
We will select Windows ➔ Preferences
In Preferences ➔ Show advanced settings (bottom of the page)

For Dark theme:

1) In Preferences go to ➔ General ➔ Appearance ➔ Theme: Dark ➔ OK
2) In Preferences go to ➔ C/C++ ➔ Editor ➔ Mark Occurrence ➔ "Annotation" link ➔ C/C++ occurrence ➔ Text as "Box"
3) In Preferences go to ➔ C/C++ ➔ Editor ➔ Mark Occurrence ➔ "Annotation" link ➔ C/C++ Occurrence ➔ Text as "Box"
4) In Preferences go to ➔ C/C++ ➔ Editor ➔ Inactive Code Highlight ➔ Colour: select "Black ■"

For faster debug/modification/re-debug:

In Preferences go to ➔ General ➔ Keys
And set F8, F9, F10, F11, F12 as Build, Debug, Run, Pause and Terminate, with Unbind/Binding boxes
It allows us to modify our code, run a debug session, terminate is and restart that process again rather quickly. Since we cannot restart the chip while debugging it, we have to quit the debug mode (terminate command) and restart it (Debug command) to restart the program in debug mode.

Hardware Configuration

Debug tools:

By the nature of this project debugging through serial connection (USART) or any other heavily intrusive tools is not possible. Indeed the SOC's CPU is under heavy load during the transmission of the data because of the high throughout and the frequent interrupts. That is why tools, as non-intrusive as possible, have been used. Among them, logic analyser, data dump, breakpoints, core dump, register and memory read (in parallel to the CPU load) has been used. The logic analyser solution has been proved the most convenient one with memory read. The logic analyser used is the USB device from Saleae (Logic pro 16, 16 channels with 100 Msamples/sec on 4 channels).

Router Access point:

To simplify the communication and avoid protocol collisions as much as possible, a dedicated access point has been used to make the bridge between the SOC and the receiver PC running the python program. The access point during the test wasn’t connected to the internet in order to limit these collisions and minimise the error rate. The access point was configured in WPA2 password protection.
This application at the moment works with fixed IP address for the Server, where the emitting device must be aware of the IP address to send packets

PC receiver side

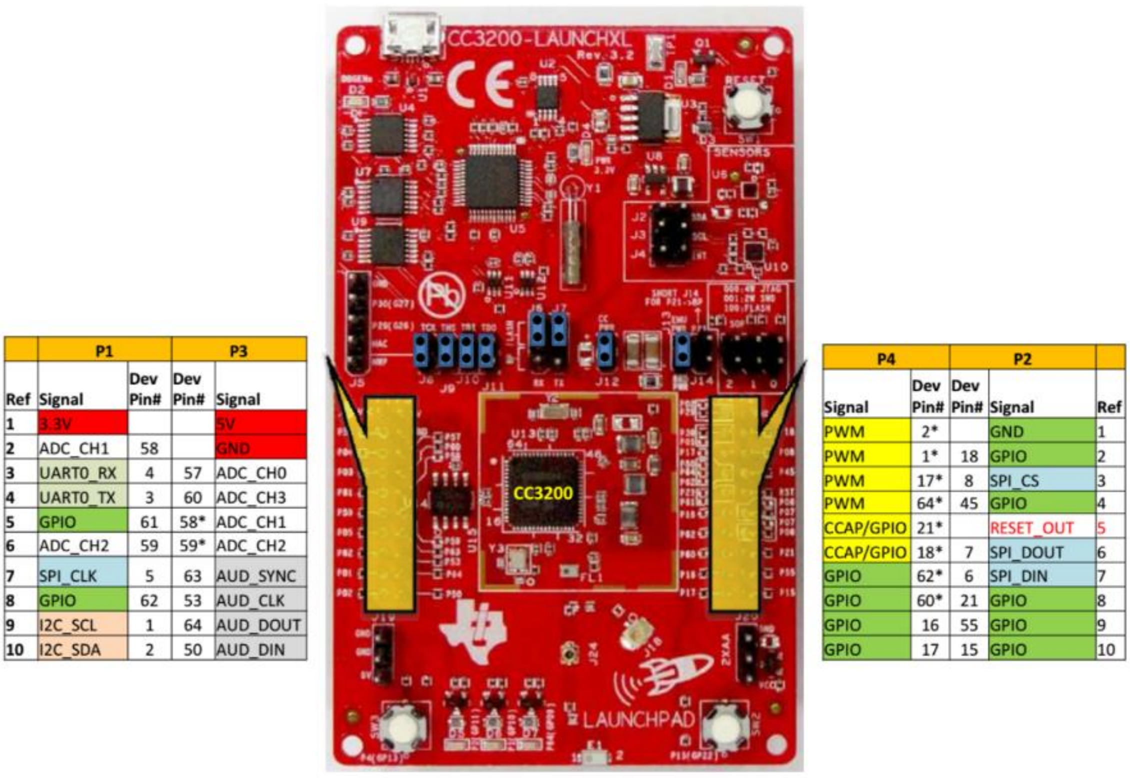
The PC on the receiving side is intended to run a Linux OS, the distribution used here was Linux Mint. The data reception and tests has been made with Python (2.7).

Connexions

The link between the DE2-115 (FPGA) and the CC3200 launchpad (SOC) uses the SPI protocol through simple jump wires between the FPGA pins. 4 wires are dedicated for the SPI (SCLK, MISO, MOSI, and NSS), 1 wire for the “SpiBlock” signal, and 3 wires to connect the ground of the two boards (essential to maintain signal stability).

According to the representation of the headers in the pictures thereafter, the connexions (jump wires) between the CC3200 Launchpad and the DE2-115 are:

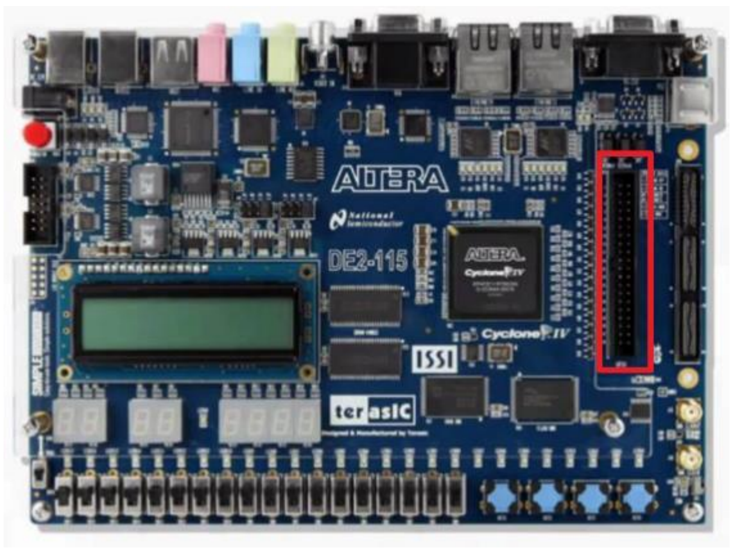
- P**2** Ref **3** (SPI_CS) to DE2-115 GPIO[**2**]
- P**2** Ref **6** (SPI_DOUT) to DE2-115 GPIO[**1**]
- P**2** Ref **7** (SPI_DIN) not connected (SPI reception is not used)
- P**1** Ref **7** (SPI_CLK) to DE2-115 GPIO[**0**]
- P**2** Ref **10** (GPIO) to DE2-115 GPIO[**35**]
- **Two ground pins** from the DE2-115 and CC3200 Launchpad where connected together to realise a common ground needed for the SPI signals. In our case, the current and voltage between the two board’s grounds were measured with a multimeter to detect any dangerous current flow, enabling us to make these connexions safely.

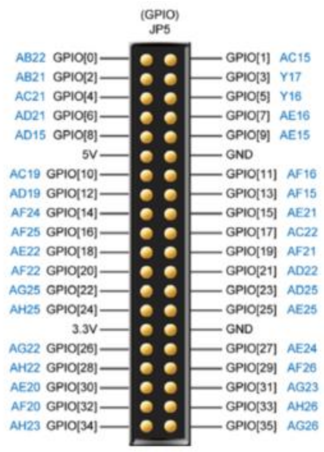


	P1			P3
Ref	Signal	Dev Pin#	Dev Pin#	Signal
1	3.3V			5V
2	ADC_CH1	58		GND
3	UART0_RX	4	57	ADC_CH0
4	UART0_TX	3	60	ADC_CH3
5	GPIO	61	58*	ADC_CH1
6	ADC_CH2	59	59*	ADC_CH2
7	SPI_CLK	5	63	AUD_SYNC
8	GPIO	62	53	AUD_CLK
9	I2C_SCL	1	64	AUD_DOUT
10	I2C_SDA	2	50	AUD_DIN

P4		P2		
Signal	Dev Pin#	Dev Pin#	Signal	Ref
PWM	2*		GND	1
PWM	1*	18	GPIO	2
PWM	17*	8	SPI_CS	3
PWM	64*	45	GPIO	4
CCAP/GPIO	21*		RESET_OUT	5
CCAP/GPIO	18*	7	SPI_DOUT	6
GPIO	62*	6	SPI_DIN	7
GPIO	60*	21	GPIO	8
GPIO	16	55	GPIO	9
GPIO	17	15	GPIO	10

Figure 7 : TI CC3200 LaunchPad pin description





AB22	GPIO[0]	GPIO[1]	AC15
AB21	GPIO[2]	GPIO[3]	Y17
AC21	GPIO[4]	GPIO[5]	Y16
AD21	GPIO[6]	GPIO[7]	AE16
AD15	GPIO[8]	GPIO[9]	AE15
	5V	GND	
AC19	GPIO[10]	GPIO[11]	AF16
AD19	GPIO[12]	GPIO[13]	AF15
AF24	GPIO[14]	GPIO[15]	AE21
AF25	GPIO[16]	GPIO[17]	AC22
AE22	GPIO[18]	GPIO[19]	AF21
AF22	GPIO[20]	GPIO[21]	AD22
AG25	GPIO[22]	GPIO[23]	AD25
AH25	GPIO[24]	GPIO[25]	AE25
	3.3V	GND	
AG22	GPIO[26]	GPIO[27]	AE24
AH22	GPIO[28]	GPIO[29]	AF26
AE20	GPIO[30]	GPIO[31]	AG23
AF20	GPIO[32]	GPIO[33]	AH26
AH23	GPIO[34]	GPIO[35]	AG26

Figure 8: Altera DE2-115 pin description

Set-Up and Testing

This section present an explication step by step on how to program the FPGA, program/debug the MCU, set-up the receiving server and verify the integrity of the received data.

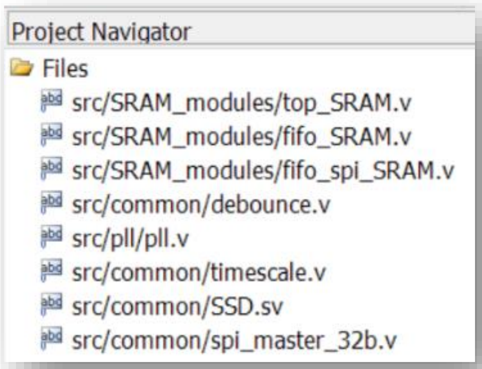
FPGA Set-up for the DE2-115 boar from Altera:

We will open the Quartus project:

- We navigate in our project folder (through windows explorer): FPGA/Fpga_Soc_Interface.qfp (right click ➔ open with ➔ We select Quartus II executable
- File explorer ➔ right click on src/top_SRAM.v ➔ set as top-level entity
- Assignment ➔ Import assignments ➔ select « DE2_115_pin_assignements.qsf »
- We compile it: Processing ➔ Start Compilation

If another version of Quartus is used the following instruction will help to build the project from scratch:

- We open Quartus II (here version 13.1)
- We close the project pop-up windows
- File ➔ new project wizard ➔ next
- Project directory text box: We browse to our project folder (we our sources are)
- Project Name text box: We name our project, "Fpga_Soc_Interface" here
- Next ➔ Next
- Device family Cyclone IV E
- Available device: EP4CE115F29C7, Next, Next, Finish
- Project ➔ Add/Remove files in project..
- We browse to our project folder/src/common, we select all files, open
- We browse to src/SRAM_modules, we select all files, open
- We browse to src/pll, we select "pll.v" files, Add, Ok
- In the project navigator windows: right click on top_SRAM.v and set as top level entity
- Assignments ➔ Settings ➔ Libraries
- In "global library names" We add (by browsing) src/common, src/pll, src/SRAM_modules
- Assignments ➔ Import assignments ➔ select « DE2_115_pin_assignements.qsf »
- We click OK
- Assignments ➔ Analysis & Synthesis Settings ➔ More Settings.. ➔ "State Machine Processing" ➔ We select "User-Encoded"
- Processing ➔ Start Compilation


Project Navigator

Files

- src/SRAM_modules/top_SRAM.v
- src/SRAM_modules/fifo_SRAM.v
- src/SRAM_modules/fifo_spi_SRAM.v
- src/common/debounce.v
- src/pll/pll.v
- src/common/timescale.v
- src/common/SSD.sv
- src/common/spi_master_32b.v

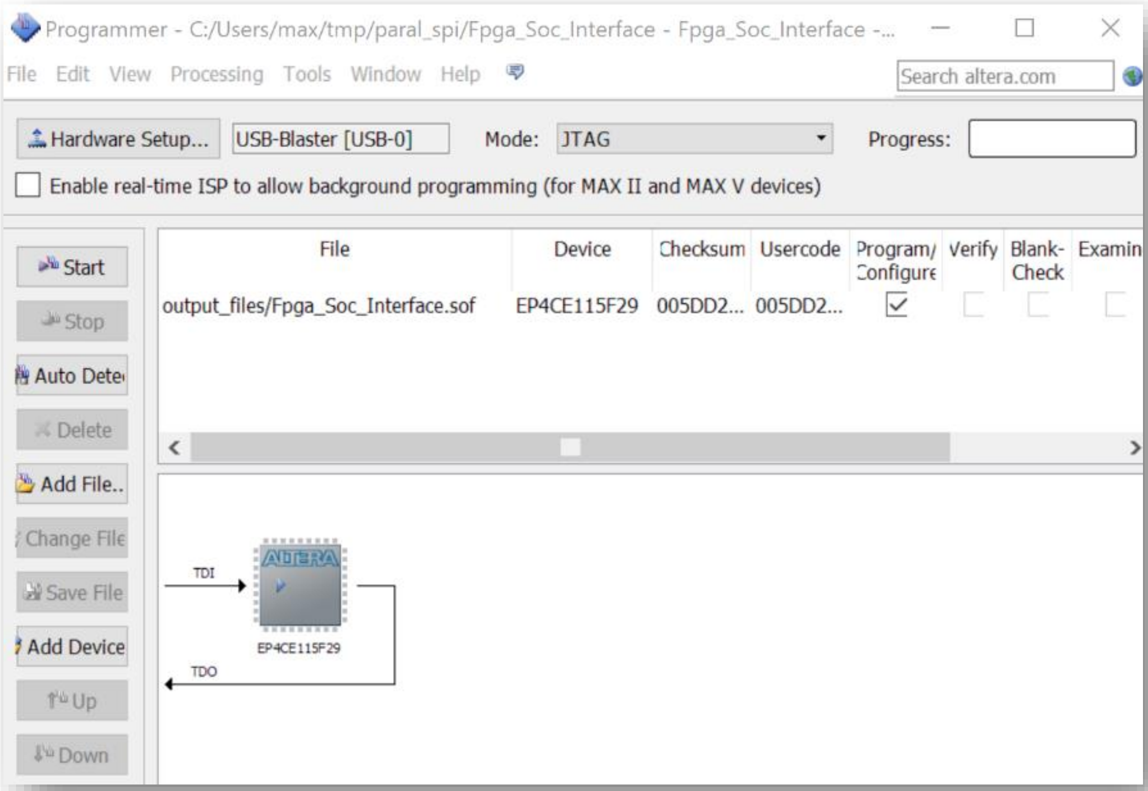
Figure 9: project navigator

We will then program the FPGA:

- We connect the USB cable to the board and power it on
- Set the Switch SW19 to RUN

  In this user manual we write the configuration to the RAM of the FPGA. It is possible it to burn its ROM, to keep the configuration even after a reset of the FPGA (see Additional Resources section).
- Tools ➔ Programmer ➔ Hardware Setup… ➔ Currently selected hardware: USB-Blaster ➔ Close
- Add file : output_files/paral_spi.sof
- Delete all other files if any
- Start
- We wait until successful programming, if it fails, we will retry it. (it usually fails one time the first time we create a programming session)
- We press the reset button. In our case on the DE2-115 we programed the KEY0 button to be the reset, and adjust the SPI throughput with the slide button (switches). Each switch adds delay in the frame transmission.

We may want to use a logic analyser to confirm that the SPI is correctly working. SPI frames by bulk transmission of 256 words packets should be seen on the analyser. In the pictures thereafter, we recorded the CLK, MOSI, ENABLE and MISO of the FPGA SPI pins at different time scales. Since we do not receive data from the MCU to the FPGA, we kept MISO to "0". On the second picture, we can see an increasing 32 bits counter on the MOSI pin (0x00850A71, 0x00850A72, 0x00850A73, etc…). We used a counter here, to later on with the python program, check the integrity of the received data.


Programmer - C:/Users/max/tmp/paral_spi/Fpga_Soc_Interface - Fpga_Soc_Interface -...

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress:

☐ Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examin
output_files/Fpga_Soc_Interface.sof	EP4CE115F29	005DD2...	005DD2...	✓	☐	☐	☐

Figure 10: Programmer

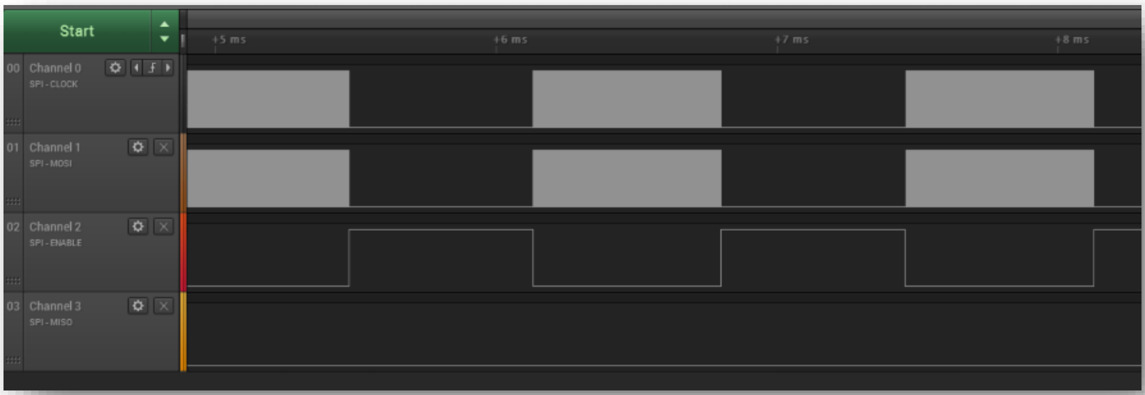


Figure 11: SPI frames (1 ms scale) from logic analiser

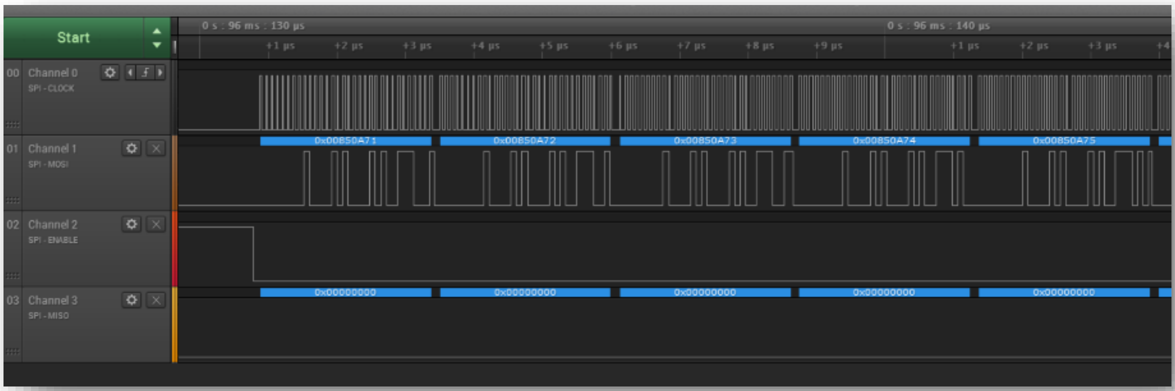


Figure 12: SPI frames (1 µs scale)

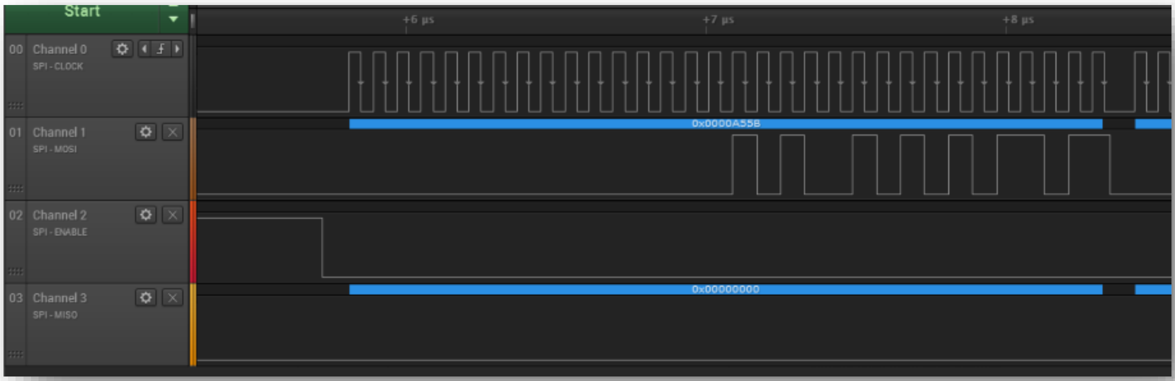


Figure 13: SPI frame (1 µs scale zoomed)

MCU Set-up for the CC3200 Launchpad from TI **(Complete Set-up)**

Remark
We will present here how to build the project from scratch with the sources. Given the complexity due to the linking between the CC3200 SDK and all the parameters that are specific to this particular implementation, a full tutorial is detailed. A zipped file with all project files is provided as well and a second **quicker tutorial will be found after this one.** The **Getting started manual from TI** will be useful to complete the information of this tutorial.

We launch CCS and select a workspace folder. That workspace will contain our project folders as it is the rule with eclipse based IDEs.

Once in the IDE, we select Project ➔ Import CSS Projects
We browse to the root of the CC3200 SDK (in our TI folder) and click OK
We select then

- driverlib
- simplelink

We will leave the 2 checkboxes unchecked as we use a link to the SDK (rather than copying its files to our project folder) in order to easily update the drivers and avoid a complex linkage within CCS.

Now before we compile these two projects, we need to update the compiler.
View ➔ Resource Explorer
We will be prompted to update several tools; one of them is the compiler. Let us update it.
After the download and installation of the update, CCS will restart.

On the Project explorer panel (if not visible: View ➔ Project explorer), we will right click on the each project ➔ Properties ➔ General ➔ Compiler Version: select the most recent TI one, here it is TI v16.9.4.LTS. Then click OK.

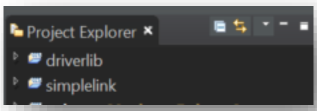
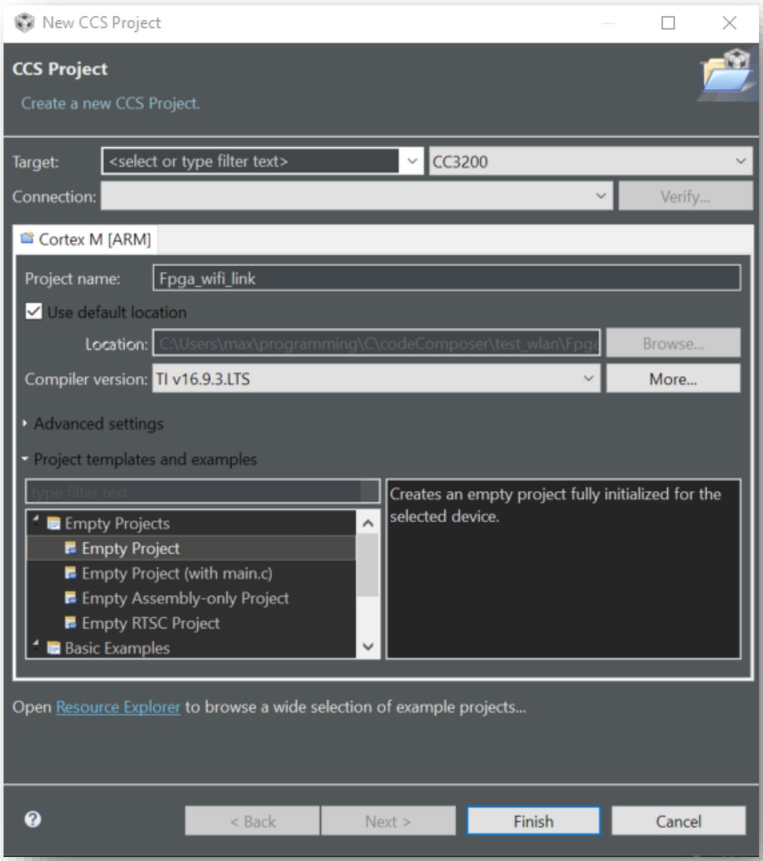


Figure 14: Project explorer

Right click on each project ➔ build.

Now, we will build a new project and incorporate the sources needed by our application.

Project ➔ New CCS project
We select Target: right combo-Box: CC3200
We select the most recent TI compiler, here TI v16.9.3.LTS
Let us give a name to the project, here Fpga_wifi_link
We select empty Project


New CCS Project

CCS Project

Create a new CCS Project.

Target: <select or type filter text> CC3200

Connection:

Cortex M [ARM]

Project name: Fpga_wifi_link

☑ Use default location

Location: C:\Users\max\programming\C\codeComposer\test_wlan\Fpga

Compiler version: TI v16.9.3.LTS

Advanced settings

Project templates and examples

Creates an empty project fully initialized for the selected device.

Open Resource Explorer to browse a wide selection of example projects...

Figure 15: Project creation window

From the MCU project sources spi_tcp folder (.zip file), we will copy/paste the "application" folder to the Fpga_wifi_link project folder (in the CCS workspace).

Now we need to indicate the libraries to be used to compile this project (these libraries were created by the two compiled project "driverlib" and "simplelink")

In the Project explorer we right click on Fpga_wifi_link project ➔ Properties ➔ Build ➔ ARM linker ➔ file search path ➔ Include library file or command file as input ➔ add file (green cross)
We browse and select individual library (.a) files:

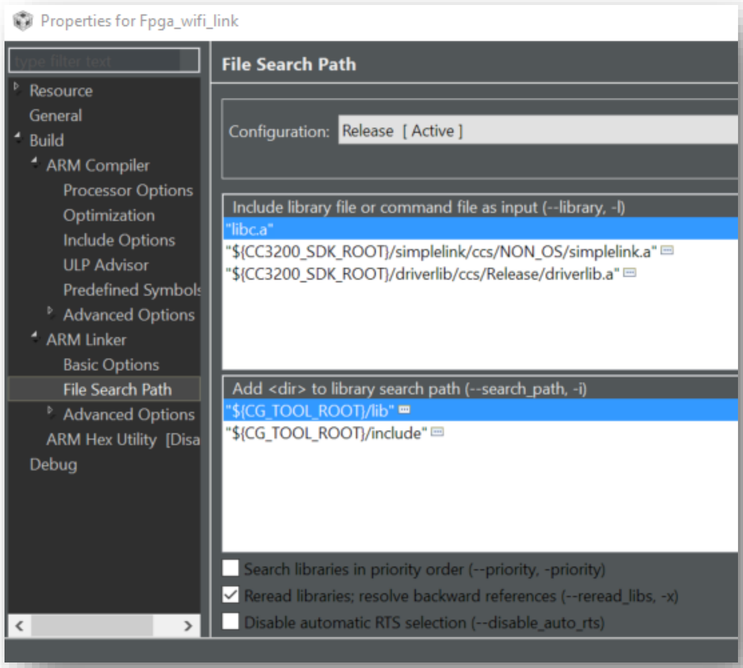
"Our CC3200 SDK ROOT"/simplelink/ccs/NON_OS/simplelink.a

"Our CC3200 SDK ROOT"/driverlib/ccs/Release/driverlib.a

Or we can copy paste in it (more complicated in case of missing CC3200_SDK_ROOT variable):

"${CC3200_SDK_ROOT}/simplelink/ccs/NON_OS/simplelink.a"
"${CC3200_SDK_ROOT}/driverlib/ccs/Release/driverlib.a"

Note: the ${CC3200_SDK_ROOT} variable must be set:
Right click on Fpga_wifi_link project ➔ Properties ➔ Build ➔ Variables ➔ Add ➔
Variable name: CC3200_SDK_ROOT
Value: The path to our SDK, here it is: “C:\ti\cc3200_sdk\cc3200-sdk”



Configuration: Release [Active]

Include library file or command file as input (--library, -l)

"libc.a"
"${CC3200_SDK_ROOT}/simplelink/ccs/NON_OS/simplelink.a"
"${CC3200_SDK_ROOT}/driverlib/ccs/Release/driverlib.a"

Add <dir> to library search path (--search_path, -i)

"${CG_TOOL_ROOT}/lib"
"${CG_TOOL_ROOT}/include"

- [ ] Search libraries in priority order (--priority, -priority)
- [x] Reread libraries; resolve backward references (--reread_libs, -x)
- [ ] Disable automatic RTS selection (--disable_auto_rts)

Figure 16: Project properties, Linker options

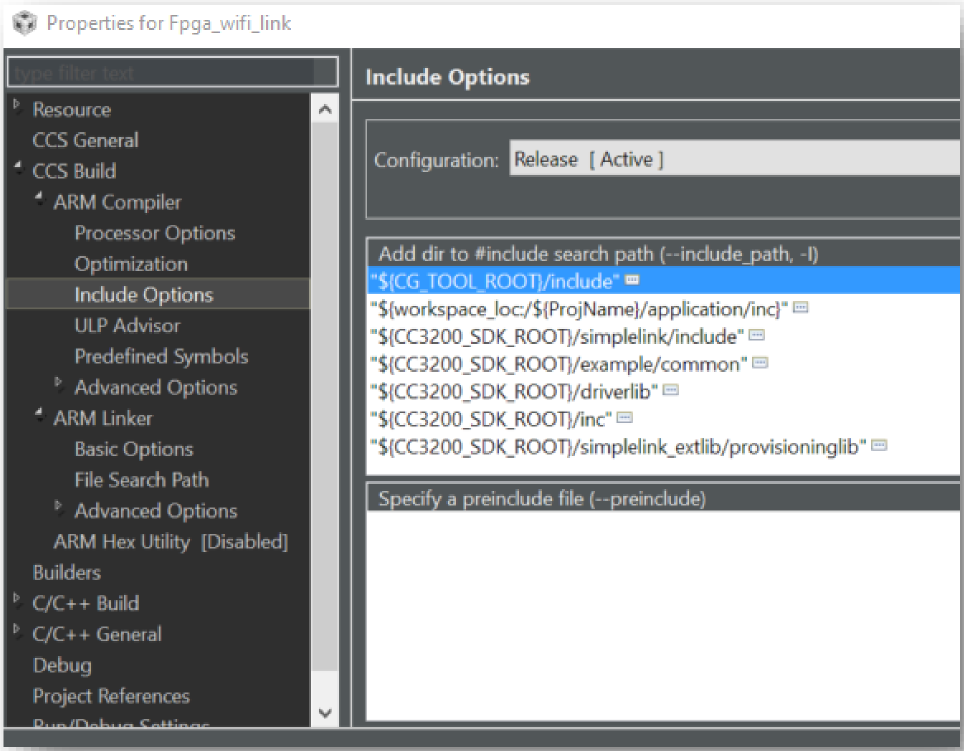
Then we need to indicate the include file path to the compiler:
In the Project explorer right click on Fpga_wifi_link project ➔ Properties ➔ Build ➔ ARM Compiler ➔ Include Options ➔ “Add <dir>” to #include search path ➔ add file (green cross)

We browse and select individual folders:

“Our CC3200 SDK ROOT”/simplelink/include"
“Our CC3200 SDK ROOT”/example/common"
“Our CC3200 SDK ROOT”/driverlib"
“Our CC3200 SDK ROOT”/inc"
“Our CC3200 SDK ROOT”/simplelink_extlib/provisioninglib"

Or copy paste in it (more complicated in case of missing CC3200_SDK_ROOT variable):

```
"${CC3200_SDK_ROOT}/simplelink/include"
"${CC3200_SDK_ROOT}/example/common"
"${CC3200_SDK_ROOT}/driverlib"
"${CC3200_SDK_ROOT}/inc"
"${CC3200_SDK_ROOT}/simplelink_extlib/provisioninglib"
```



Configuration: Release [Active]

Add dir to #include search path (--include_path, -I)

```
"${CG_TOOL_ROOT}/include"
"${workspace_loc:/${ProjName}/application/inc}"
"${CC3200_SDK_ROOT}/simplelink/include"
"${CC3200_SDK_ROOT}/example/common"
"${CC3200_SDK_ROOT}/driverlib"
"${CC3200_SDK_ROOT}/inc"
"${CC3200_SDK_ROOT}/simplelink_extlib/provisioninglib"
```

Figure 17: Project properties, Compiler Options

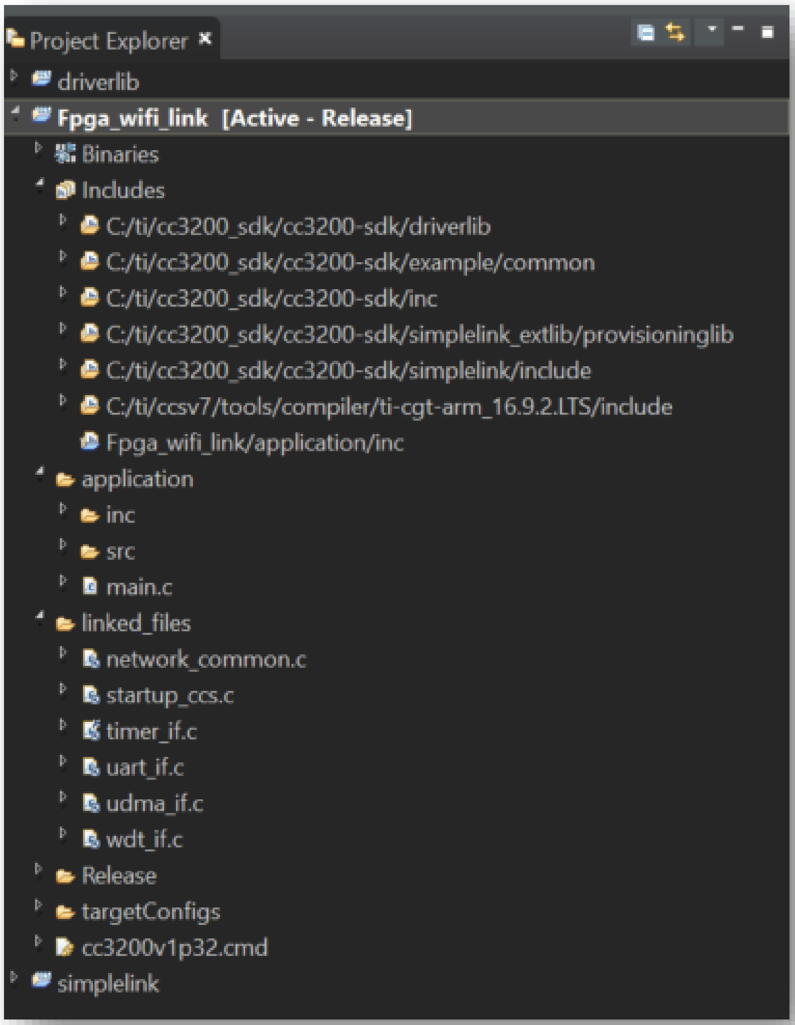
Now we need to add linked resources needed for some peripherals such as the UART, Timer, WatchDog, etc…

In the Project explorer we right click on Fpga_wifi_link project ➔ Add files ➔ We navigate to "CC3200_SDK_ROOT"\example\common ➔ We select the 6 following files:

- network_common.c
- startup_ccs.c
- timer_if.c
- uart_if.c
- uddma_if.c

- wdt_if.c

Then we click OK. For a better file organization, we can create a new folder in the project and drag/drop those linked files in it.
Here is our project tree:

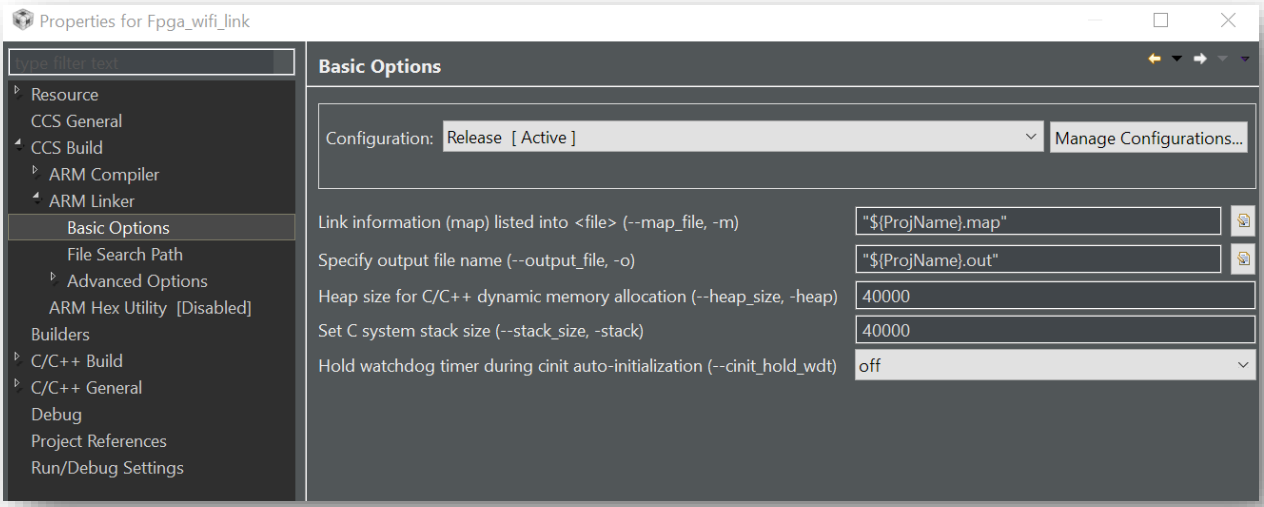


```
Project Explorer
driverlib
Fpga_wifi_link  [Active - Release]
    Binaries
    Includes
        C:/ti/cc3200_sdk/cc3200-sdk/driverlib
        C:/ti/cc3200_sdk/cc3200-sdk/example/common
        C:/ti/cc3200_sdk/cc3200-sdk/inc
        C:/ti/cc3200_sdk/cc3200-sdk/simplelink_extlib/provisioninglib
        C:/ti/cc3200_sdk/cc3200-sdk/simplelink/include
        C:/ti/ccsv7/tools/compiler/ti-cgt-arm_16.9.2.LTS/include
        Fpga_wifi_link/application/inc
    application
        inc
        src
        main.c
    linked_files
        network_common.c
        startup_ccs.c
        timer_if.c
        uart_if.c
        udma_if.c
        wdt_if.c
    Release
    targetConfigs
    cc3200v1p32.cmd
simplelink
```

Figure 18: Project explorer, final files tree

Now we will increase the Stack and Heap size to allow us to store enough SPI data, and we will enable the watchdog. The watchdog is a security process that will reset the MCU in case that it gets stuck stop performing its task.

In the Project explorer right click on Fpga_wifi_link project ➔ Properties ➔ CCS Build ➔ ARM Linker ➔ Basic Options
Modify the Heap and Stack size text boxes to set them to 40,000 and turn the watchdog setting on OFF.



Configuration: Release [Active]

Link information (map) listed into <file> (--map_file, -m): "${ProjName}.map"
Specify output file name (--output_file, -o): "${ProjName}.out"
Heap size for C/C++ dynamic memory allocation (--heap_size, -heap): 40000
Set C system stack size (--stack_size, -stack): 40000
Hold watchdog timer during cinit auto-initialization (--cinit_hold_wdt): off

Figure 19: Project properties, Stack Heap size settings

In “common.h”, we comment the five (5) #define lines as per bellow in the red rectangle. These lines have been implemented in net_app.h and would generate a redefinition error. It is more convenient to keep all the Access point information in the same file (net_app.h).

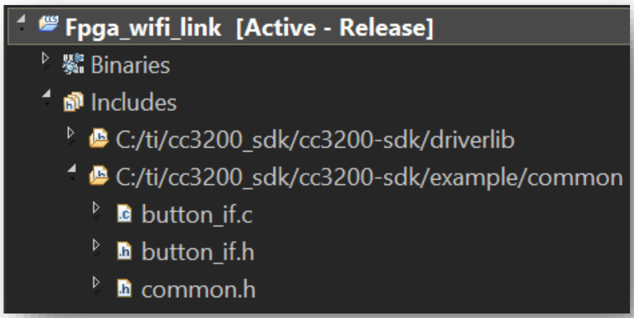
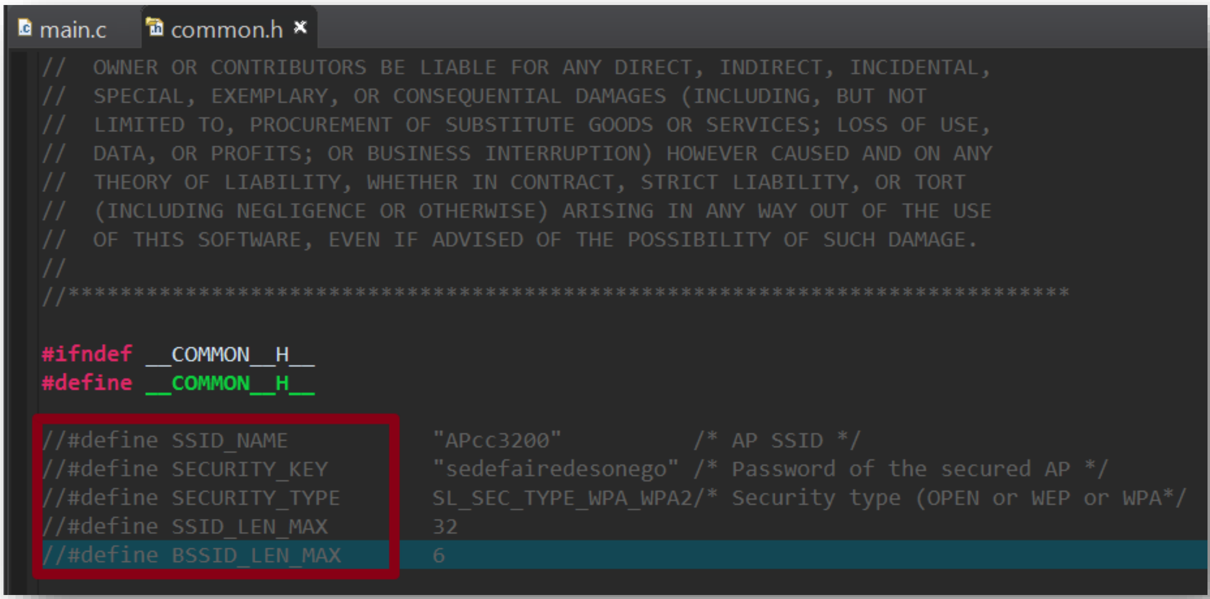

Fpga_wifi_link [Active - Release]
Binaries
Includes
C:/ti/cc3200_sdk/cc3200-sdk/driverlib
C:/ti/cc3200_sdk/cc3200-sdk/example/common
button_if.c
button_if.h
common.h

Figure 20: Location of “common.h”



```
//  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
//  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
//  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
//  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
//  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
//  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
//  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
//*****************************************************************************

#ifndef __COMMON__H__
#define __COMMON__H__

//#define SSID_NAME           "APcc3200"           /* AP SSID */
//#define SECURITY_KEY        "sedefairedesonego" /* Password of the secured AP */
//#define SECURITY_TYPE       SL_SEC_TYPE_WPA_WPA2/* Security type (OPEN or WEP or WPA*/
//#define SSID_LEN_MAX        32
//#define BSSID_LEN_MAX       6
```

Figure 21: Five lines to be commented in “common.h”

On our receiving computer, which runs a Linux OS, we ran the command “ip addr show” in a terminal to get our IP (V4) address and note it. Then we need to translate that IP to a hexadecimal number. We will browse to the following website, enter our IP (under the XXX.XXX.XXX.XXX format) and convert it to a Hex number format. http://www.miniwebtool.com/ip-address-to-hex-converter/
Then in the net_app.h:

- Modify the current IP (hex) define “IP_ADDR” by our new IP (hex) address.
- Replace “SSID_NAME” by the SSID (name) of our access point.
- Replace “SECURITY_KEY” by its password. (Well … we are not working on a cyber-security project)
- Concerning “PORT_NUM” we can keep it as it is, or replace it with our desired port number.

In order to program and debug the CC3200 chip via the FTDI (USB interface chip on the dev board), we need to indicate to CSS how to communicate with it, with a target configuration file.
View ➔ Target Configuration:
We right-click on User Defined, select Import Target Configuration and select the file CC3200.ccxml from “CC3200_SDK_ROOT”\tools\ccs_patch\. We select the Copy files option when prompted.

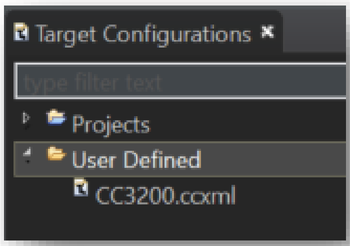


Figure 22: Target configuration widow

We right click on CC3200.ccxml and “Set as default”

We can now compile it and solve the possible compilation errors:
In the Project explorer right click on Fpga_wifi_link project ➔ Build Project

Note:
In order to test the system MCU and Server **without** the FPGA, we can uncomment the line in “**common_include.h**”: **// #define WITHOUT_FPGA 1**
This will disable the SPI and set-up a timer interrupt that will create and load data (same increasing counter as the FPGA data) in the CC3200 Fifo, before sending them to the server. The transmission speed should be about 8 Mbits per second.

We are almost ready to test our program. We need now to set up a serial connexion terminal. We will use putty here. We made it work under Linux mint, but the FTDI driver (with a dual COM port) was working with difficulties, making the debug quite complicated to do in parallel to the debugging in CCS.

The FTSI chip on the CC3200 Launchpad offers debug and terminal capabilities through a dual serial connexion. To know which COM port to use under putty, with a windows OS, we will connect our CC3200 launchpad with the USB cable, then:
Control panel ➔ Hardware and Sound ➔ Device Manager ➔ ports (COM & LPT)
CC3200 LP Dual Port (COM4), in our case, the port number is COM4.

In putty, fill the configuration as indicated in the screenshot bellow (with our own COM port number) and open it.

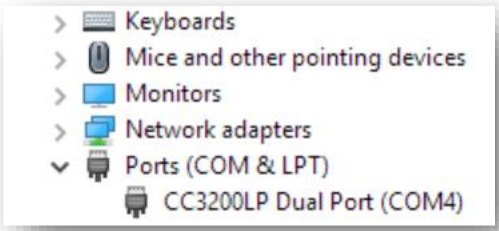
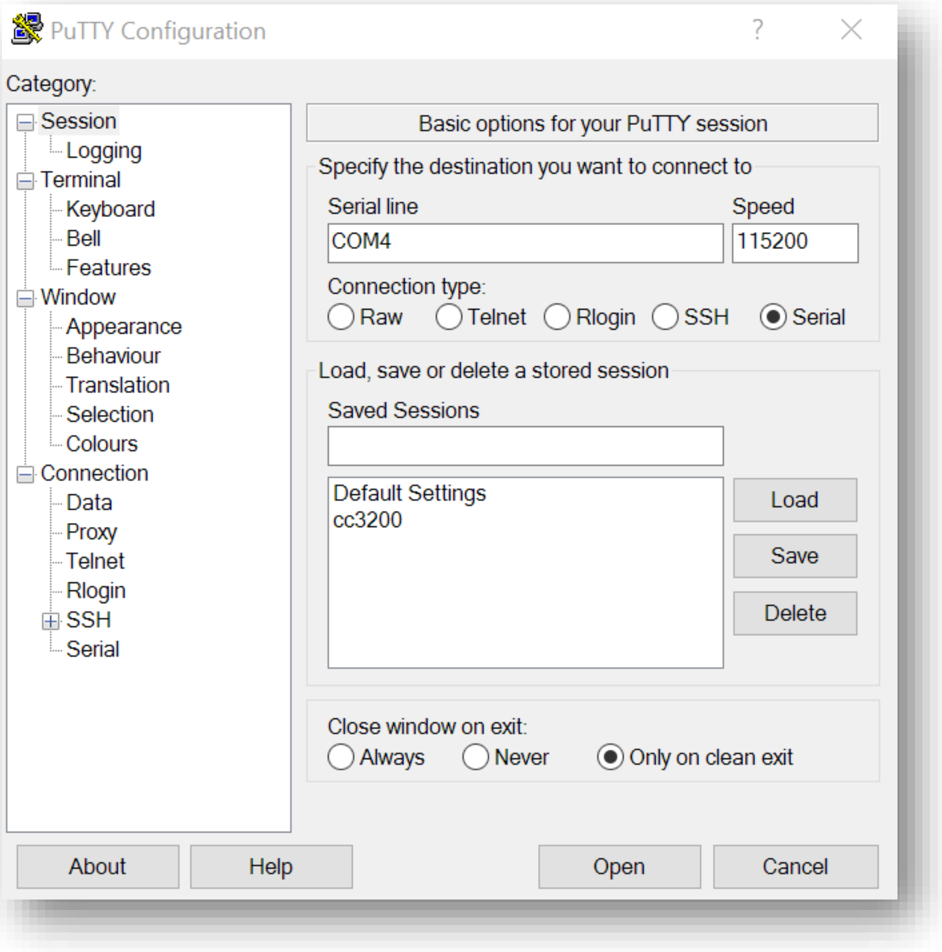


Figure 23: Device Manager (Windows OS)



Basic options for your PuTTY session

Specify the destination you want to connect to

Serial line: COM4

Speed: 115200

Connection type: Raw Telnet Rlogin SSH [✓ Serial]

Load, save or delete a stored session

Saved Sessions:

Default Settings
cc3200

Close window on exit: Always Never [✓ Only on clean exit]

Figure 24: Putty interface

On the CC3200 Launchpad, set the SOP Jumper on “100” with the black jumper provided.

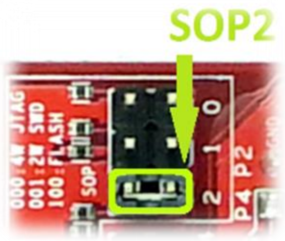


Figure 25: SOP Jumper (here in position “100”)

Back in CCS IDE, Right click on the project ➔ Debug as ➔ Code Composer Debug Session
Once the program loaded in the MCU’s Ram, click on the resume button:

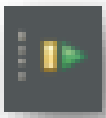
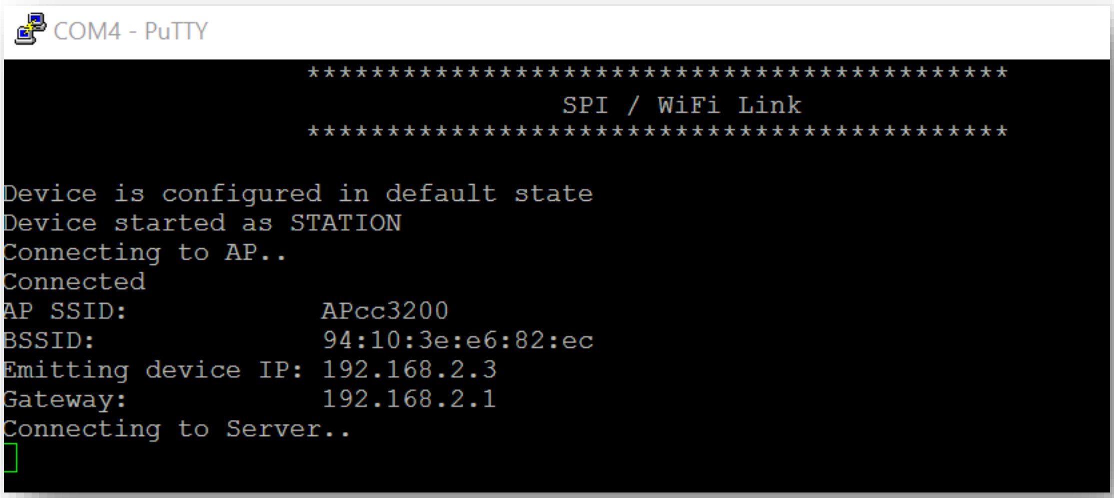


Figure 26: resume button

Since we have not launched the python server, the MCU will stop at “Connecting to Server…” and will probably reset after 5 seconds due the watchdog interrupt (since the program is considered as “stuck”). On the Terminal, we should see something similar to this:



```
COM4 - PuTTY
                   *********************************************
                                   SPI / WiFi Link
                   *********************************************

Device is configured in default state
Device started as STATION
Connecting to AP..
Connected
AP SSID:              APcc3200
BSSID:                94:10:3e:e6:82:ec
Emitting device IP:   192.168.2.3
Gateway:              192.168.2.1
Connecting to Server..
```

Figure 27: Putty terminal with expected connexion info

MCU Set-up for the CC3200 Launchpad from TI **(Quicker set-up)**

This quicker tutorial use a pre-existing workspace and projects (Fpga_Wifi_Link.zip). Still, some linkage needs to be done due to the fact that we are using a 3200 SDK with a path different from the one configured in the zipped file.

In our CCS workspace folder we will copy the content of the Fpga_Wifi_Link.zip into that folder:
Content of the workspace:

```
/Workspace_folder
        .jxbrowser-data
        .metadata
        Fpga_wifi_link
        RemoteSystemsTempFiles
```

We open CCS and select our workspace folder.
If project are automatically imported, right click on each and delete them (**without** deleting them on the disk)

In the IDE, we select Project ➔ Import CSS Projects
We browse to the root of the CC3200 SDK (in our TI folder) and click OK
We select then:

- Driverlib
- simplelink

We will leave the 2 checkboxes unchecked as we use a link to the SDK (rather than copying its files to our project folder) in order to easily update the drivers and avoid a complex linkage within the CCS.

On the Project explorer panel we will right click on each project (simplelink and driverlib) ➔ Properties ➔ General ➔ Compiler Version: select the most recent TI one, here it is TI v16.9.4.LTS. Then click OK.

We can now compile both project: right click on them ➔ Build project
In the IDE, we select Project ➔ Import CSS Projects
We browse to our workspace folder (in our TI folder) and click OK
We select

- Fpga_wifi_link

In the same way than for the simplelink and driverlib project, we will change the compiler to the most recent one.

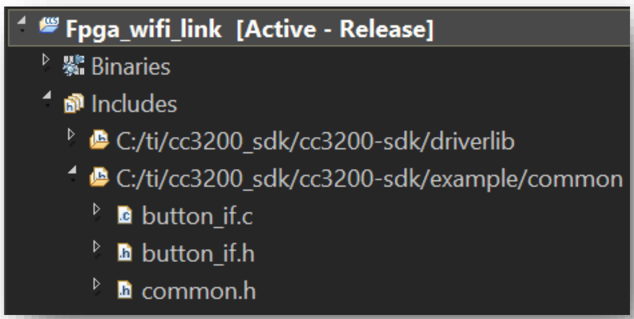
Right click on it ➔ Properties ➔ C/C++ Build ➔ Build Variables ➔ Show system variables ➔ CC3200_SDK_ROOT ➔ Edit
We replace "Value" by the path to our SDK. In our case: "C:\ti\cc3200_sdk\cc3200-sdk"

In our project explorer, remove all files in linked_files.
Right click on our project ➔ add files
We navigate to our SDK/example/common folder, and select:

- network_common.c
- startup_ccs.c
- timer_if.c
- uart_if.c
- uddma_if.c
- wdt_if.c

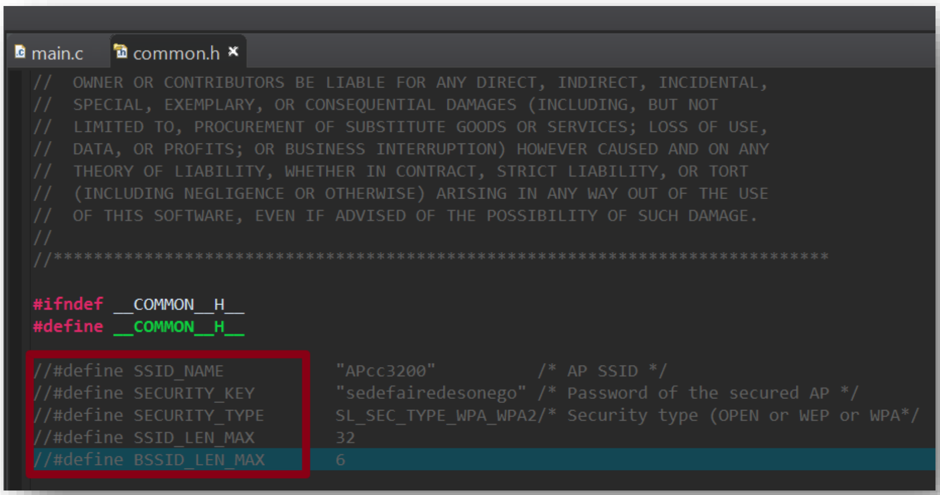
We can move these files (that appeared on our project tree) to the folder linked_files/

In "common.h", we comment the five (5) #define lines as per bellow in the red rectangle. These lines have been already implemented in net_app.h and would generate a redefinition error. It is more convenient to keep all the Access point information in the same file (net_app.h).



```
Fpga_wifi_link  [Active - Release]
    Binaries
    Includes
        C:/ti/cc3200_sdk/cc3200-sdk/driverlib
        C:/ti/cc3200_sdk/cc3200-sdk/example/common
            button_if.c
            button_if.h
            common.h
```

Figure 28: Location of "common.h"



```
//  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
//  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
//  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
//  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
//  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
//  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
//  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
//*****************************************************************************

#ifndef __COMMON__H__
#define __COMMON__H__

//#define SSID_NAME          "APcc3200"          /* AP SSID */
//#define SECURITY_KEY       "sedefairedesonego" /* Password of the secured AP */
//#define SECURITY_TYPE      SL_SEC_TYPE_WPA_WPA2/* Security type (OPEN or WEP or WPA*/
//#define SSID_LEN_MAX       32
//#define BSSID_LEN_MAX      6
```

Figure 29: Five lines to be commented

We can now compile Fpga_wifi_link and debug/run the program as explained in the Complete Set-up version.

Note:
In order to test the system MCU and Server **without** the FPGA, we can uncomment the line in "common_include.h" : **// #define WITHOUT_FPGA 1**
This will disable the SPI and set-up a timer interrupt that will create and load data (same increasing counter as the FPGA data) in the CC3200 Fifo, before sending It to the server. The transmission speed should be about 8 Mbits per second.

Please refer to the complete version (above) for in depth details.

Receiving Computer Side (python program)

We used python 2.7 to develop the (simple) program that will receive the data through TCP/IP protocol.

We connect our computer (hosting the server) to the same network than the MCU. In our case, the MCU was connected to an access point (AP) without internet connexion (closed network configuration), thus we had to connect both the MCU and the Receiving computer to the same AP. If our MCU is connected to an AP with a granted access to internet, we can connect our receiving computer to any AP with internet access.

We will open a terminal and navigate to the src folder of the "Reception" project folder.
We based our code on the fact that we are using static IP address.
We will launch the shell script ./launch_reception.sh to test our system FPGA + wireless_MCU (launch_tests.sh to run tests without the transmitting devices)

After following the direction asking for PORT number, amount of packet to receive and destination file to save the data, the server will wait for client (CC3200) connexions.

Running the Test

1) We will ensure that the FPGA is loaded with configuration thanks to the programmer as explained there before. Since the FPGA configuration is loaded in its RAM, which is a volatile memory, each time we turn OFF and ON the FPGA, we must download the configuration into it. Flashing the FPGA ROM is a solution to avoid download the configuration after each reset. (See additional resources). Once configured, we will turn SW0 to SW5 ON and SW6 to SW17 OFF, to achieve a FIFO input speed of about 7,5 Mbit/sec. Then we can press KEY1 to start the FPGA to MCU SPI transmission. The red LEDs will show the real time filling of the SRAM FIFO.

   Note : In the case the FPGA ROM in already flashed with the configuration program (see additional ressources), no need to pass by Quartus. We will simply turn ON the development board, then we will turn SW0 to SW5 ON and SW6 to SW17 OFF, to achieve a FIFO input speed of about 7,5 Mbit/sec. Finally we will press KEY1 to start the FPGA to MCU SPI transmission.

2) We connect our computer (hosting the server) to the same network than the MCU.

3) We launch the bash script “launch_reception.sh” in the Reception python project folder and follow the instructions.

4) We connect our CC3200 launchpad to CCS and launch our serial terminal program, Putty in our case. We enter in debug mode (see MCU complete set-up). We press resume and observe that the program enter in the infinite TCP loop after the CC3200 get connected to the server. *Between each test in Debug mode, the MCU might get stuck in an infinite loop due to an unexpected and non-cleared interrupt. To avoid this, we will manually reset the MCU between each debug session by push the “reset” button on the CC3200 launchpad.* This issue disappear when the MCU code is flashed in its ROM (rather that in its RAM which is the case in debug mode).

   Note: In the case where the MCU ROM is flashed with the program (see additional ressources), we will simply connect our serial terminal and press the reset button of the launchpad. No need to execute the instruction paragraph above.

5) After a while, the receiving part of the python program will complete and file named Received_data.dat will be found in the /received_data folder. Meanwhile, our MCU will state that the connexion has been lost and will reset until it finds a new connexion.

6) On the receiving side, an integrity test will be prompted at the end of the reception, to verify that there is no missing packet.

7) The received data can be visually checked by using the Hexdump command on a Unix environment:

```
hexdump –e \4 “%08X\n”’ path_to_our_received_data_file.dat
```

Additional Resources

MCU ROM Flashing:

Most of the tests done with the MCU (CC3200 from TI) have been made by writing the compiled program under its binary form to the RAM of the chip. This method enables us to quickly program and debug the chip with the debug tools integrated to Code Composer Studio. However it obviously restrains us to always keep the MCU connected to a computer, since each time we power-up the device we need to reprogram it due to the volatile nature of the RAM.

On way to program durably the chip is to burn the code in its flash memory. For that, we need to use external tools not included to Code Composer Studio. Several tools can be used for that purpose (uniflash, OpenOCD, Serial Programing using the serial boot loader, and others...). In this manual, we will discuss about the Uniflash software from TI, which is the simplest and safest solution.

Uniflash (for CC3200) download link:
http://processors.wiki.ti.com/index.php/CCS_UniFlash_v3.4.1_Release_Notes#Installation_Instructions
Ti Service Pack installer download link:
http://www.ti.com/tool/download/CC3200SDK

Download and install “Service Pack Installer for CC3200SDK” and “Uniflash”.

Uniflash tutorial for the CC3200: (To realise the operations bellow)
http://processors.wiki.ti.com/index.php/CC3100_%26_CC3200_UniFlash_Quick_Start_Guide

To upload the service pack (containing all the “wifi” and “socket” functions):
Run Uniflash
We will close Putty and CodeComposer (or other application using our CC3200 Dev Board)
Verify that the SOP jumper is on “100”
Push the reset button of the CC3200 launchpad
Open uniflash
File ➔ New configuration
Connexion ➔ CC3x Serial(UART) Interface
“Get Version” to be sure the connexion between the program uniflash (via the FTDI chip) and the CC3200 is OK.
Format ➔ Capacity “1 MB” ➔ OK (this will format the chip flash memory, removing consequently all data)
Operation ➔ Service Pack Programming ➔ select “servicepack_1.xx.xx.xx.bin” in the service pack folder created by the service pack installation ➔ OK

To upload our program (.bin):
Run Uniflash
… Same steps as above until …
"Get Version" to check the connexion PC/Launchpad
System Files → /sys/mcuimg.bin (MCU Image binary) → URL (browse) → We will select our "Fpga_wifi_link.bin" or whatever the name of our project, in the Release folder of our project folder.
We tick the boxes: Erase, Update, Verify
Operation → Program
We put the SOP jumper on "000" (we remove the jumper)

Remarks

In order to flash the CC3200 on the development board (CC3200 launchpad), we set the jumper on position "100": flash for the "SOP Jumper" (see CC3200 launchpad user's guide). Once flashed, we will remove the jumper ("000": 4W JTAG) to run the code, and press the hard reset button on the board. We will put the Jumper in the "100" position whenever we wants to place the MCU in debug mode.

FPGA ROM Flashing

The following web page explains in depth how to flash the ROM of our FPGA and is compatible with the DE2-115 Development Board from Altera:
http://www.chroniclesofatechnophile.com/2013/06/generation-of-pof-file-from-sof-file.html

Let's not forget to push the switch SW19 back to "Run" after programming the FPGA Rom, in order to run the configuration.

MCU System Power Consumption

During transmission @ 8 Mbits/sec:	~150 mA	@ 3.3V
Non transmission (Awake WiFi ON):	~50 mA	@ 3.3V
Non transmission (Awake WiFi OFF):	~13 mA	@ 3.3V
Non transmission (Sleep):	~0.28 mA	@ 3.3V

Further Work

The TCP/IP sending gets interrupted sometimes. The connexion however is not lost but due to the fast data rate, during the interruption, the data sent by the SPI are not processed, hence are lost. In details, the socket_send() function take usually few µs. But at some time (randomly every few seconds) it takes up to 0.2 ms. The fact that the send() function get "stuck" for fractions of milliseconds is caused most likely by collisions due to the nature of the WiFi and TCP/IP protocols.

TI affirms that the chip is capable of 12 Mbit/sec and tests made can confirm it. However this is in an ideal case where no other peripherals on the chip are used. In our case, 8/6 Mbits per second is a more reasonable throughput to be expected, although the data rate depending on the quality of the connexion oscillate between 5 and 8 Mbits per second.

An Improvement would be to connect the FPGA to a larger SRAM memory (64 Mbits) in order to be able to buffer up few seconds worth of data (at 8 Mbits/sec) while the TCP connexion get stuck.

Implementing a python interface to transmit data to control the CC3200/FPGA.